TOTALVIEW® REFERENCE GUIDE

Version 8.15.7
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
Contents

About this Guide
Overview ................................................................. 1
TotalView Family Differences ................................. 2
Conventions .............................................................. 3
TotalView Documentation .................................. 4
Contacting Us ......................................................... 6

Part 1: CLI Commands
Chapter 1 CLI Command Summary .............................. 8
Chapter 2 CLI Commands ....................................... 20
  Command Overview .............................................. 20
  General CLI Commands .................................. 20
  CLI Initialization and Termination Commands .......... 20
  Program Information Commands ...................... 21
  Execution Control Commands ......................... 22
  Action Points ............................................... 22
  Platform-Specific CLI Commands ...................... 23
  Other Commands ...................................... 23
  alias ................................................... 24
  capture ................................................. 26
  dactions ............................................... 28
  dassign ............................................... 33
  dattach ............................................... 35
  dbarrier ................................................. 38
  dbreak ................................................ 43
  dcache ................................................. 47
  dcalltree .............................................. 48
  dcheckpoint ............................................ 52
Chapter 3  CLI Namespace Commands  174

Command Overview  ................................................. 174
  Accessor Functions  .......................................... 174
  Helper Functions  .............................................. 175
actionpoint  ....................................................... 176
dec2hex  .......................................................... 179
dll  ................................................................. 180
testCodes  .......................................................... 182
eexpr  ............................................................... 184
focus_groups  ....................................................... 186
focus_processes  ................................................... 187
focus_threads  ....................................................... 188
group  ............................................................... 189
hex2dec  ............................................................ 191
process  ............................................................ 192
read_symbols  ...................................................... 196
respond  ............................................................ 197
scope  ............................................................... 198
source_process_startup  ........................................ 200
symbol  ............................................................. 201
thread  .............................................................. 214
<table>
<thead>
<tr>
<th>Chapter 4</th>
<th>Batch Debugging Using tvscript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>230</td>
</tr>
<tr>
<td>tvscript Command Syntax</td>
<td>231</td>
</tr>
<tr>
<td>tvscript Options</td>
<td>235</td>
</tr>
<tr>
<td>tvscript External Script Files</td>
<td>241</td>
</tr>
<tr>
<td>Logging Functions API</td>
<td>241</td>
</tr>
<tr>
<td>Process Functions API</td>
<td>241</td>
</tr>
<tr>
<td>Thread Functions API</td>
<td>241</td>
</tr>
<tr>
<td>Action Point API</td>
<td>241</td>
</tr>
<tr>
<td>Event API</td>
<td>242</td>
</tr>
<tr>
<td>Example tvscript Script File</td>
<td>243</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5</th>
<th>TotalView Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>247</td>
</tr>
<tr>
<td>Top-Level (::) Namespace</td>
<td>248</td>
</tr>
<tr>
<td>TV:: Namespace</td>
<td>256</td>
</tr>
<tr>
<td>TV::MEMDEBUG:: Namespace</td>
<td>293</td>
</tr>
<tr>
<td>TV::GUI:: Namespace</td>
<td>295</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6</th>
<th>Creating Type Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>303</td>
</tr>
<tr>
<td>Why Type Transformations</td>
<td>304</td>
</tr>
<tr>
<td>Creating Structure and Class Transformations</td>
<td>306</td>
</tr>
<tr>
<td>Transforming Structures</td>
<td>306</td>
</tr>
<tr>
<td>build_struct_transform Function</td>
<td>308</td>
</tr>
<tr>
<td>Type Transformation Expressions</td>
<td>308</td>
</tr>
<tr>
<td>Using Type Transformations</td>
<td>312</td>
</tr>
<tr>
<td>C++View</td>
<td>314</td>
</tr>
<tr>
<td>Writing a Data Display Function</td>
<td>315</td>
</tr>
<tr>
<td>TV_ttf_type_ascii_string</td>
<td>315</td>
</tr>
<tr>
<td>TV_ttf_type_int</td>
<td>315</td>
</tr>
<tr>
<td>Templates</td>
<td>317</td>
</tr>
<tr>
<td>Precedence - Searching for TV_ttf_display_type</td>
<td>318</td>
</tr>
<tr>
<td>TV_ttf_add_row</td>
<td>318</td>
</tr>
<tr>
<td>TV_ttf_ec_ok</td>
<td>319</td>
</tr>
<tr>
<td>TV_ttf_ec_not_active</td>
<td>319</td>
</tr>
</tbody>
</table>
## Part 3: Platforms and Operating Systems

### Chapter 9 Platforms and Compilers 356

**Overview** ................................................................. 356

- Compiling with Debugging Symbols .................................. 357
  - Apple Running Mac OS X ............................................. 357
  - HP-UX for IA-64 ...................................................... 357
  - IBM AIX on RS/6000 Systems ........................................ 358
  - IBM Blue Gene ........................................................ 359
  - IBM Power Linux ..................................................... 359
  - Linux Running on an x86 Platform ................................ 360
  - Linux Running on an x86-64 Platform ............................ 360
  - Linux Running on an Itanium Platform ............................ 361
  - Sun Solaris ............................................................ 361
- Using gnu_debuglink Files ............................................ 362
  - Total View Command-Line Options and CLI State Variables .... 362
  - Searching for the gnu_debug_link File ............................ 363
- Linking with the dbfork Library ...................................... 364
  - Linking with HP-UX ................................................ 364
  - dbfork on IBM AIX on RS/6000 Systems ........................... 364
  - Linking C++ Programs with dbfork ................................. 365
  - Linux or Mac OS X .................................................. 366
  - SunOS 5 SPARC ....................................................... 366

### Chapter 10 Operating Systems 367

**Operating Systems** .................................................. 367

- Supported Operating Systems .................................... 368
- Troubleshooting Mac OS X Installations .......................... 369
  - Problem Description ................................................. 369
  - For All OS X Versions .............................................. 369
  - For OS X Versions 10.8 (Mountain Lion) or Later ................ 369
  - Remotely Debugging without Console Access .................... 370
- Mounting the /proc File System .................................... 371
  - Mounting /proc with SunOS 5 ..................................... 371
- Swap Space .............................................................. 372
  - Swap Space on HP HP-UX .......................................... 372
  - Maximum Data Size ................................................ 372
  - Swap Space on IBM AIX ............................................. 373
Swap Space on Linux ........................................... 373
Swap Space on SunOS 5 ....................................... 374

Shared Libraries .................................................. 375
  Changing Linkage Table Entries and LD_BIND_NOW .......... 376
  Using Shared Libraries on HP-UX ............................ 376

Debugging Your Program's Dynamically Loaded Libraries .... 377
dlopen Options for Scalability ..................................... 379
  Filtering dlopen Events ........................................ 379
  Handling dlopen Events in Parallel ............................ 381
  Known Limitations .............................................. 382

Remapping Keys .................................................. 383

Expression System ............................................... 384
  Expression System on IBM AIX-Power and Blue Gene/Q .... 384

Chapter 11 Architectures 385

Overview .......................................................... 385

AMD and Intel x86-64 ............................................ 386
  x86-64 General Registers ...................................... 386
  x86-64 Floating-Point Registers .............................. 387
  x86-64 FPCR Register ......................................... 388
    Using the x86-64 FPCR Register ............................ 389
  x86-64 FPSR Register ......................................... 389
  x86-64 MXCSR Register ....................................... 390

Power Architectures .............................................. 391
  Power General Registers ..................................... 391
  Blue Gene Power Registers ................................... 393
  Blue Gene/Q QPX Floating-Point Registers ................. 394
  Power MSR Register ......................................... 394
  Power Floating-Point Registers .............................. 395
  Power FPSCR Register ........................................ 395
    Using the Power FPSCR Register ......................... 397

Intel IA-64 ........................................................ 398
  Intel IA-64 General Registers ............................... 398
  IA-64 Processor Status Register Fields (PSR) ............... 400
  Current Frame Marker Register Fields (CFM) ............... 401
  Register Stack Configuration Register Fields (RSC) ....... 401
  Previous Function State Register Fields (PFS) ............ 402
  Floating Point Registers .................................... 402
  Floating Point Status Register Fields ..................... 402
About this Guide

Overview

The information in this guide is organized in four parts:

- **Part I, “CLI Commands,”** on page 7 contains descriptions of all the CLI commands, the variables that you can set using the CLI, and other CLI-related information.

- **Part II, “Running TotalView,”** on page 333 documents all possible command-line options as well as those that customize the behavior of the `tvdsrv`.

- **Part III, “Platforms and Operating Systems,”** on page 355 provides general information on compilers, runtime environments, operating systems, and supported architectures.

- **Part IV, “Appendices,”** on page 413 includes Appendix A which describes how to create startup profiles for environments that TotalView does not define.
TotalView Family Differences

This manual describes the TotalView Enterprise, TotalView Team, and TotalView Individual debuggers. Each of these supports the use of the CLI debugger as well. In all cases, TotalView Enterprise and TotalView Team have the same features, differing only in the way they are licensed. TotalView Individual differs in its feature set.

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

The basic differences are:

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

The following table describes the conventions used in this book:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>Brackets are used when describing optional parts of a command.</td>
</tr>
<tr>
<td>arguments</td>
<td>In a command description, text in italics represents information you enter. Elsewhere, italics is used for emphasis.</td>
</tr>
<tr>
<td>Bold text</td>
<td>In a command description, <strong>bold text</strong> represents keywords or options that must be entered exactly as displayed. Elsewhere, it represents words that are used in a programmatic way rather than their normal way.</td>
</tr>
<tr>
<td>Example text</td>
<td>In program listings, this represents a program or something you'd enter in response to a shell or CLI prompt. Bold text here indicates exactly what you should type. If you're viewing this information online, example text is in color.</td>
</tr>
</tbody>
</table>
# TotalView Documentation

The following table describes all available TotalView documentation:

<table>
<thead>
<tr>
<th>Product</th>
<th>Title</th>
<th>Description</th>
<th>HTML</th>
<th>PDF</th>
<th>Print</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General TotalView Documentation</strong></td>
<td>Getting Started with TotalView Products</td>
<td>Introduces the basic features of TotalView, MemoryScape, and ReplayEngine, with links for more detailed information</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>TotalView Platforms and System Requirements</strong></td>
<td>TotalView Platforms and System Requirements</td>
<td>Defines platform and system requirements for TotalView, MemoryScape, and ReplayEngine</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>TotalView Evaluation Guide</strong></td>
<td>TotalView Evaluation Guide</td>
<td>Brochure that introduces basic TotalView features</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>User Guides</strong></td>
<td>TotalView User Guide</td>
<td>Primary resource for information on using the TotalView GUI and the CLI</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>User Guides</strong></td>
<td>Debugging Memory Problems with MemoryScape</td>
<td>How to debug memory issues, relevant to both TotalView and the MemoryScape standalone product</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>User Guides</strong></td>
<td>Reverse Debugging with Replay Engine</td>
<td>How to perform reverse debugging using the embedded add-on ReplayEngine</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>Reference Guides</strong></td>
<td>TotalView Reference Guide</td>
<td>A reference of CLI commands, how to run TotalView, and platform-specific detail</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>New Features</strong></td>
<td>TotalView New Features</td>
<td>New features in the current release</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>New Features</strong></td>
<td>MemoryScape New Features</td>
<td>New features in the current release</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>Installation Guides</strong></td>
<td>TotalView Install Guide</td>
<td>Installing TotalView and the FLEXlm license manager</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>Installation Guides</strong></td>
<td>MemoryScape Install Guide</td>
<td>Installing MemoryScape as a standalone product</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td>Title</td>
<td>Description</td>
<td>HTML</td>
<td>PDF</td>
<td>Print</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------</td>
<td>---------------------------------------------------------------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>In-Product Help</td>
<td></td>
<td>Help screens launched from within the product's GUI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TotalView Help</strong></td>
<td></td>
<td>Context-sensitive help launched from TotalView</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MemoryScape Help</strong></td>
<td></td>
<td>Context-sensitive help launched from MemoryScape</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Contacting Us

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

**By Email**

- For technical support: support@roguewave.com
- For documentation issues: rwonlinedocs@roguewave.com

**By Phone**

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

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

- The version of TotalView and the platform on which you are running TotalView.
- An example that illustrates the problem.
- A record of the sequence of events that led to the problem.
This part of the *TotalView Reference Guide* contains five chapters that describe the TotalView Command Line Interpreter (CLI).

**Chapter 1, “CLI Command Summary,”** on page 8
Summarizes all CLI commands.

**Chapter 2, “CLI Commands,”** on page 20
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 174
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 230
Discusses how to create batch scripts that run TotalView unattended.

**Chapter 5, “TotalView Variables,”** on page 247
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 6, “Creating Type Transformations,”** on page 303
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.
This chapter contains a summary of all TotalView® Debugger CLI commands. The commands are described in detail in Chapter 2, “CLI Commands,” on page 20 and Chapter 3, “CLI Namespace Commands,” on page 174.

**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 defin-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
[-data pby_data_array] [-show_details] [-sort columns] [-hide_backtrace]
[-save_as_csv filename] [-save_as_dot filename]
**dcheckpoint**

Creates a checkpoint on IBM AIX

```
dcheckpoint [ -delete | -halt ]
```

**dcont**

Continues execution and waits for execution to stop

```
dcont
```

**dcuda**

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

```
dcuda
```

**ddelete**

Deletes some action points

```
ddelete action-point-list
```

Deletes all action points

```
ddelete -a
```

**ddetach**

Detaches from the processes

```
.ddetach
```

**ddisable**

Disables some action points

```
.ddisable action-point-list [ -block number-list ]
```

Disables all action points

```
.ddisable -a
```

**ddlopen**

Loads a shared object library

```
.ddlopen [ -now | -lazy ] [ -local | -global ] [ -mode int ] filespec
```

Displays information about shared object libraries

```
.ddlopen [ -list dll-ids... ]
```

**ddown**

Moves down the call stack

```
.ddown [ num-levels ]
```
**dec2hex**

Converts a decimal number into hexadecimal

`TV::dec2hex number`

**denable**

Enables some action points

`denable action-point-list`

Enables all disabled action points in the current focus

`denable -a`

**dexamine**

Display memory contents

`dexamine [-column_count cnt] [ -count cnt] [ -data_only]`

`[-show_chars] [ -string_length len] [ -format fmt]`

`[-memory_info] [-wordsize size] variable_or_expression`

**dflush**

Removes the top-most suspended expression evaluation

`dflush`

Removes all suspended `dprint` computations

`dflush -all`

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

`dflush susp-eval-id`

**dfocus**

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

`dfocus p/t-set`

Executes a command in this P/T set

`dfocus [ p/t-set command ]`

**dga**

Displays global array variables

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

**dgo**

Resumes execution of target processes

`dgo`

**dgroups**

Adds members to thread and process groups
**dgroups** 
- **-add** [-g gid] [id-list]
  Deletes groups
- **-delete** [-g gid]
  Intersects a group with a list of processes and threads
- **-intersect** [-g gid] [id-list]
  Prints process and thread group information
- **-new** [thread_or_process] [-g gid] [id-list]
  Creates a new thread or process group
- **-remove** [-g gid] [id-list]
  Removes members from thread or process groups

**dhalt**
Suspends execution of processes

**dheap**
Shows Memory Debugger state
- **-status**
  Applies a saved configuration file
- **-apply_config** {default | filename}
  Shows information about a backtrace
- **-backtrace** [subcommands]
  Compares memory states
- **-compare** subcommands [optional_subcommands]
  [process | filename] [process | filename]
  Enables or disables the Memory Debugger
- **-enable | -disable**
  Enables or disables event notification
- **-event_filter** subcommands
  Writes memory information
- **-export** subcommands
  Specifies which filters the Memory Debugger uses
- **-filter** subcommands
  Writes guard blocks (memory before and after an allocation)
Enables and disables the retaining (hoarding) of freed memory blocks
   dheap -hoard [ subcommands ]

Displays Memory Debugger information
   dheap -info [ -backtrace ] [ start_address [ end_address ] ]

Indicates whether an address is within a deallocated block
   dheap -is_dangling address

Locates memory leaks
   dheap -leaks [ -check_interior ]

Enables or disables Memory Debugger event notification
   dheap -[no]notify

Paints memory with a distinct pattern
   dheap -paint [ subcommands ]

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

Enables and disables allocation and reallocation notification
   dheap -tag_alloc subcommand start_address [ end_address ]

Displays the Memory Debugger's version number
   dheap -version

**dhistory**

Displays information about the state of the program as it is being replayed. If you have received a timestamp, you can go back to the line that was executing at that time.
   dhistory [ -info ] [ -get_time ] [ -go_time time ] [ -go_live ] [ -enable ] [ -disable ]

**dhold**

Holds processes
   dhold -process

Holds threads
   dhold -thread

**dkill**

Terminates execution of target processes
   dkill [ -remove ]

**dlappend**

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

dlist
Displays code relative to the current list location
  dlist [-n num-lines ]
Displays code relative to a named location
  dlist breakpoint-expr [-n num-lines ]
Displays code relative to the current execution location
  dlist -e [-n num-lines ]

dll
Manages shared libraries
  TV::dll action [ dll-id-list ] [-all ]

dload
Loads debugging information
  dload [-g gid ] [-mpi starter_value ] [-r hname]
     [-replay | -noreplay ]
     [ -env variable=value ] ... [-e ] executable
     [ -parallel_attach_subset subset_specification ]

dmstat
Displays memory use information
  dmstat

dnext
Steps source lines, stepping over subroutines
  dnext [-back ] [ num-steps ]

dnexti
Steps machine instructions, stepping over subroutines
  dnexti [-back ] [ num-steps ]

dout
Executes until just after the place that called the current routine
  dout [-back ] [ frame-count ]

dprint
Prints the value of a variable or expression
  dprint [-nowait ] [-slice slice_expr ] [-stats [-data ] ] variable_or_expression
dptsets
Shows the status of processes and threads in an array of P/T expressions
  dptsets [ ptset_array ] ...

drerun
Restarts processes
  drerun [ cmd_arguments ] [ < infile ]
  [ > [ > ] [ & ] outfile ]
  [ 2> [ > ] errfile ]

drestart
Restarts a checkpoint on AIX
  drestart [ -halt ] [ -g gid ] [ -r host ] [ -no_same_hosts ]
Restarts a checkpoint on SGI
  drestart [ process-state ] [ -no_unpark ] [ -g gid ] [ -r host ]
  [ -ask_attach_parallel ] [ -no_attached_parallel ]
  [ -no_preserve_ids ] checkpoint-name

drun
Starts or restarts processes
  drun [ cmd_arguments ] [ < infile ]
  [ > [ > ] [ & ] outfile ]
  [ 2> [ > ] errfile ]

dsession
Loads a session
  dsession [ -load session_name ]

dset
Creates or changes a CLI state variable
  dset debugger-var value
Views current CLI state variables
  dset [ debugger-var ]
Sets the default for a CLI state variable
  dset -set_as_default debugger-var value

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 ]
CLI Command Summary

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`
Chapter 2

CLI Commands

Command Overview

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

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:
CLI Commands

Command Overview

• **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**: 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.
- **dstep**: Executes statements, moving into subfunctions if required.
- **:**: 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.
**alias**

Create or views pseudonyms for commands

**Format**

Creates a new user-defined pseudonym for a command

```
alias alias-name defn-body
```

Views previously defined aliases

```
alias [ alias-name ]
```

**Arguments**

*alias-name*

The name of the command pseudonym being defined.

*defn-body*

The text that Tcl substitutes when it encounters *alias-name*.

**Description**

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

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

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

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

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

```
proc dwatch {} {
    puts "The dwatch command is disabled"
}
```

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

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

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

*alias nt dnext*

Defines a command called *nt* that executes the *dnext*-command.

*alias nt*

Displays the definition of the *nt* alias.

*alias*

Displays the definitions of all aliases.

*alias m {dlist main}*

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

*alias step2 {dstep; dstep}*

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

*alias step2 {s ; s}*

Creates an alias that performs the same operations as that in the previous example, differing in that it uses the alias for *dstep*. You could also create the following alias which does the same thing: *alias step2 (s 2).*

*alias step1 {f p1. dstep}*

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

**Initializing TotalView**

*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.

**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**

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

Saves the current process status to a Tcl variable.

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

Saves the printed value of argc into a Tcl variable.

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

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

```
int2_array(1) = -8 (0xffff8)
int2_array(2) = -6 (0xffffa)
int2_array(3) = -4 (0xfffc)
int2_array(4) = -2 (0xfffe)
```

Because the `capture` command records all information sent to it by the commands in the `foreach` loop, you do not have to use a `dlist` command.
exec cat << [ capture help commands ] > cli_help.txt

  Writes the help text for all CLI commands to the cli_help.txt file.
set ofile [open cli_help.txt w]
capture -f $ofile help commands
close $ofile

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

**Displays information, and saves and reloads action points**

**Format**

Displays information about action points.

\[ \text{dactions} \ [ \text{ap-id-list} \] \ [ \text{-at source-loc} \] \ [ \text{-enabled} | \text{-disabled} \] \ [ \text{-enabled_blocks} | \text{-disabled_blocks} \] \ [ \text{-block_images} | \text{-block_lines} \] \]

Saves action points to a file.

\[ \text{dactions} \ -\text{save} \ [ \text{filename} \] \]

Loads previously saved action points.

\[ \text{dactions} \ -\text{load} \ [ \text{filename} \] \]

**Arguments**

**ap-id-list**

A list of action point identifiers. If you specify individual action points, the information that appears is limited to these points.

Do not enclose this list within quotes or braces. See the examples at the end of this section for more information.

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`.

**-enabled**

Shows only enabled action points.

**-disabled**

Shows only disabled action points.

**-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.
-save
   Writes information about action points to a file.

-load
   Restores action point information previously saved in a file.

filename
   The name of the file into which TotalView reads and writes action point information. If you omit this file name, TotalView writes action point information to a file named program_name.TVD.v3breakpoints, where program_name is the name of your program.

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

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.v3breakpoints (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.v3breakpoints.

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 online Help.
**CLI Commands**

---

### 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 81. (This example uses the alias instead of the full command name.) Here is the output from this command:

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

#### dactions 1 3

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

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

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

For example:

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

#### dfocus p1 dactions

Displays information about all action points defined in process 1.

#### dfocus p1 dactions -enabled

Displays information about all enabled action points in process 1.

---

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

1

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

d1.> dactions

1 shared action point for group 3:

1 [bar<std::vector<int, std::allocator<int> > > >::bar(int)] Enabled

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

d1.> dactions 1

1 shared action point for group 3:

1 [bar<std::vector<int, std::allocator<int> > > >::bar(int)] Enabled

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

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
Share in group: true
Stop when hit: process
d1.>

You could use this information, for example, to enabled the currently disabled action point addresses:
d1.> denable -block 2 3

Setting Action Points
Action Point > Enable
Action Point > Disable
Action Point > Load All
Saving Actions Points to a File
Action Point > Save All
Action Point > Save As
TV::auto_save_breakpoints Variable
dassign  

Changes the value of a scalar variable

**Format**

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

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

**Assigning Characters and Strings**

- If you are assigning a character to a \texttt{target}, place the character value within single-quotiation marks; for example, \texttt{'c'}.

- You can use the standard C language escape character sequences; for example, \texttt{
} and \texttt{\t}. These escape sequences can also be in a character or string assignment.

- If you are assigning a string to a \texttt{target}, place the string within quotation marks. However, you must escape the quotation marks so they are not interpreted by Tcl; for example, \texttt{"The quick brown fox\"}.

If \texttt{value} contains an expression, the TotalView expression system evaluates the expression. See Using the Evaluate Window in the \textit{TotalView User Guide} for 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

dassign scalar_y 102

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

dassign i 10*10

Stores the value 100 in variable `i`.

dassign i i*i

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

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

f {p1 p2 p3} as scalar_y 102

Stores the value 102 in each occurrence of variable `scalar_y` contained in processes 1, 2, and 3.

Changing the Value of Variables
Changing a Variables Data Type
**dattach**

Brings currently executing processes under TotalView control

**Format**

dattach [-g *gid*] [-r *hname*]

[-ask_attach_parallel | -no_attach_parallel]
[-replay | -no_replay]
[-go | -halt] [-rank *num*]
[-e] executable [pid-list]
[-c core-file | recording-file]
[-parallel_attach_subset subset-specification]

**Arguments**

- **-g *gid***
  
  Sets the control group for the processes being added to group *gid*. This group must already exist. (The CLI GROUPS variable contains a list of all groups. See “GROUPS” on page 251 for more information.)

- **-r *hname***
  
  The host on which the process is running. The CLI launches a TotalView Server on the host machine if one is not already running. See the Setting Up Parallel Debugging Sessions chapter of the TotalView User Guide for information on the launch command used to start this server.

  Setting a host sets it for all PIDs attached to in this command. If you do not name a host machine, the CLI uses the local host.

- **-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 online Help.

- **-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 online Help.

- **-replay | -no_replay**
  
  Enables or disables the ReplayEngine the next time the program is restarted.

- **-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.

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

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

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

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

If you have not loaded **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>dattach</td>
<td>Brings the process under TotalView control</td>
</tr>
</tbody>
</table>

**Examples**

```bash
dattach mysys 10020
```

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

```bash
dattach -e 123 10020
```

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

```bash
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.

```bash
dattach my_file 51172 52006
```

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

```bash
set new_pid [dattach -e mainprog 123]
ddetach -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.

**Using the Root Window**

- Attaching to Processes
- Examining Core Files
- `ddetach` Command
- `TV::parallel_attach` Variable
- File > New Program Command
**dbARRIER**

Defines a process or thread barrier breakpoint

**Format**

Creates a barrier breakpoint at a source location

```plaintext
dbarrier breakpoint-expr [ -stop_when_hit width ] [ -stop_when_done width ] [ -pending ]
```

Creates a barrier breakpoint at an address

```plaintext
dbarrier -address addr [ -stop_when_hit width ] [ -stop_when_done width ] [ -pending ]
```

**Arguments**

- **breakpoint-expr**
  
  This argument can be entered in more than one way, usually using a line number or a pathname containing a file name, function name, and line number, each separated by `#` characters (for example, `file#line`). For more information, see "Qualifying Symbol Names" in Chapter 13 of the TotalView User Guide.

  For more information on breakpoint expressions, see `dbreak` on page 43, particularly -Breakpoint Expressions.

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

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

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

  The argument `width` may have one of the following three values:

  - **group**
    
    Stops all processes in the control group when the execution reaches the barrier point.

  - **process**
    
    Stops the process that hit the barrier.

  - **none**
    
    Stops only the thread that hit the barrier; that is, the thread is held and all other threads continue running. If you apply this width to a process barrier breakpoint, TotalView stops the process that hit the breakpoint.

- **-stop_when_done width**
  
  After all processes or threads reach the barrier, releases all processes and threads held at the barrier. (Released means that these threads and processes can run.) Setting this option stops additional threads contained in the same `group` or `process`.

  If you do not use this option, the value of BARRIER_STOP_WHEN_DONE indicates any other processes or threads to stop.
Use the `width` argument to indicate 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 43.

**Description**

The `d_barrier` 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 `d_barrier` 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, `d_step`, `d_next`, and `d_go`) 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
   dl.<> 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.

Setting Breakpoints and Barriers

Barrier Points
Using Groups, Processes, and Threads
Creating a Satisfaction Set
Holding and Releasing Processes and Threads
Action Point > Set Barrier Command
dactions Command
dbreak Command
denable Command
ddisable Command
**dbreak**

Defines a breakpoint

**Format**

Creates a breakpoint at a source location

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

Creates a breakpoint at an address

```
  dbreak -address addr [-p | -g | -t] [[ -l lang ] -e expr ] [-pending ]
```

**Arguments**

*breakpoint-expr*

This argument can be entered in more than one way, usually using a line number or a pathname containing a file name, function name, and line number, each separated by # characters (for example, `file#line`). For more information, see “Qualifying Symbol Names” in Chapter 13 of the TotalView User Guide.

Breakpoint expressions are discussed later in this section.

*address addr*

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

*-p*

Stops the process that hit this breakpoint. You can set this option as the default by setting the **STOP_ALL** variable to **process**. See `dset` on page 135 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 135 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 135 for more information.

*-l lang*

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

*-e expr*

When the breakpoint is hit, TotalView evaluates expression `expr` in the context of the thread that hit the breakpoint. The language statements and operators you can use are described in “Chapter 16, Setting Action Points”, of the TotalView 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 checks shared libraries that are subsequently loaded to see if a breakpoint can be set. If a location is found, it is set. Stated in a different way, TotalView normally creates and sets a breakpoint at the same time. The option tells it to separate these two actions.

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

A stop group Breakpoint

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

Default Focus Width

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

Breakpoint Expressions

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

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

A breakpoint expression can evaluate to more than one source line. If the expression evaluates to a function that has multiple overloaded implementations, TotalView sets a breakpoint on each of the overloaded functions.
Set a breakpoint at the line specified by `breakpoint-expr` or the absolute address `addr`. You can enter a breakpoint expression that are sets of addresses at which the breakpoint is placed, and are as follows:

- `[##image#]filename#line_number`
  Indicates all addresses at this line number.

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

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

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

### Command alias

<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 <em>thread of interest</em></td>
</tr>
</tbody>
</table>

### Examples

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

```
dbreak 12
```

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

```
dbreak -address 0x1000764
```

Suspends process 2 when execution reaches address 0x1000764.

```
b 12 -g
```

Suspends all processes in the current control group when execution reaches line 12.

```
dbreak 57 -l f9 -e {goto $63}
```

Causes the thread that reaches the breakpoint to transfer to line 63. The host language for this statement is Fortran 90 or Fortran 95.

```
dfocus p3 b 57 -e {goto $63}
```

In process 3, sets the same evaluation point as the previous example.
Setting Breakpoints and Barriers
Defining Evaluation Points and Conditional Breakpoints
Using Groups, Processes, and Threads
Action Point > Properties Command
Action Point > At Location Command
dactions Command
dbreak Command
denable Command
ddisable Command
**dcache**

Clears the remote library cache

**Format**
dcache -flush

**Arguments**
-flush

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

**Description**

The `dcache -flush` command removes the library files that it places in your cache, located in the `.totalview/lib_cache` subdirectory in your home directory.

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

TotalView automatically deletes cached library files that it hasn't used in the last week. If you need to reclaim additional space at any time, use this command to remove files not currently being used.

**Initializing TotalView**
**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.

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

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

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

**Description**

The TotalView GUI has a Parallel Backtrace View window that displays the state of every process and thread in a parallel job. The dcalltree command makes this same data available either in the console window, or, with the -data switch, as a Tcl associative array.

The associative array has the following format:

```json
{
  {
    Key <value>
    Level <value>
    Processes <value>
    Location <value>
    PC <value>
    Host <value>
    Rank <value>
    ID <value>
    Status <value>
  }
  ...
}
```
The **-show_details** and **-hide_backtrace** switches pull in opposite directions. The **-show_details** switch shows the maximum data, including all processes and threads. The **-hide_backtrace** command hides any intermediate nodes, displaying only the root and leaf nodes. If used together, this results in a display of root and leaf nodes and all threads. This reduction can help to de-clutter the data display if the number of processes and threads is large.

**Examples**

`dfocus group dcalltree`

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

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

`dcalltree -show_details`

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

<table>
<thead>
<tr>
<th>Processes</th>
<th>Location</th>
<th>PC</th>
<th>Host</th>
<th>Rank</th>
<th>ID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>/</td>
<td>...</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>4:12[p1-4.1-3]</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>_start</td>
<td>0x004011b9</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>4:4[p1-4.1]</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>__libc_start_main</td>
<td>0x2b3425358184</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>4:4[p1-4.1]</td>
<td>...</td>
</tr>
</tbody>
</table>
dcalltree -show_details -hide_backtrace

Adding the -hide_backtrace switch reduces the clutter somewhat:

<table>
<thead>
<tr>
<th>Processes</th>
<th>Location</th>
<th>PC</th>
<th>Host</th>
<th>Rank</th>
<th>ID</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>/</td>
<td></td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>4:12[p1-4.1-3]</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>__select_nocancel</td>
<td>0xb342f56e2</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>3.3 - 1090525504</td>
<td>Stopped</td>
</tr>
<tr>
<td>1</td>
<td>__select_nocancel</td>
<td>0xb342f56e2</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>1.1 - 10905264801120 Stopped</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>__select_nocancel</td>
<td>0xb342f56e2</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>3.1 - 10905264801120 Stopped</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>snore#681</td>
<td>0x00401c05</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>1.2 - 1082132800</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>1</td>
<td>snore#681</td>
<td>0x00401c05</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>1.3 - 1090525504</td>
<td>Stopped</td>
</tr>
<tr>
<td>1</td>
<td>snore#705</td>
<td>0x00401c9b</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>2.2 - 1082132800</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>1</td>
<td>snore#681</td>
<td>0x00401c05</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>2.3 - 1090525504</td>
<td>Stopped</td>
</tr>
<tr>
<td>1</td>
<td>snore#681</td>
<td>0x00401c05</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>4.2 - 1082132800</td>
<td>Stopped</td>
</tr>
<tr>
<td>1</td>
<td>snore#681</td>
<td>0x00401c05</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>4.3 - 1090525504</td>
<td>Stopped</td>
</tr>
<tr>
<td>1</td>
<td>wait_a_while</td>
<td></td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>1:2[p3.2-3]</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>__select_nocancel</td>
<td>0xb342f56e2</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>3.3 - 1090525504</td>
<td>Stopped</td>
</tr>
<tr>
<td>1</td>
<td>wait_a_while #580</td>
<td>0x004019e9</td>
<td>&lt;local&gt;</td>
<td>-1</td>
<td>3.2 - 1082132800</td>
<td>Breakpoint</td>
</tr>
</tbody>
</table>

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
    }
}

Parallel Backtrace View
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.

**Tools > Create Checkpoint Command**

**Tools > Restart Checkpoint Command**

drestart Command
**dcont**  
Continues execution and waits for execution to stop

**Format**

dcont

**Arguments**

This command has no arguments

**Description**

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

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 interactively.

---

**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).

**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.

Starting Processes and Threads

dgo Command

dwait Command
**dcuda**  
Manages GPU threads

**Format**

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

**Arguments**

- **Bx,By, Bz**  
  The x, y and z block indices

- **Tx, Ty, Tz**  
  The x,y, and z thread indices

- **D/S/W/L**  
  The coordinates defining the physical space of the hardware:
  
  - **D**: device number
  - **S**: streaming multiprocessor (SM)
  - **W**: warp (WP) number on the SM
  - **L**: lane (LN) number on the warp

**Description**

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

**dcuda block [(Bx,By,Bz)]**

- With no arguments, shows the current CUDA block
- With a block argument of the form `(Bx,By,Bz)`, changes the CUDA focus to that block. Omitted parameters (i.e., Bz) 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. Omitted parameters (i.e., Ty and Tz, or just Tz) are unchanged.

**dcuda kernel**
Displays the logical and hardware coordinates of the current CUDA context.

**dcuda device** [<n>]
- With no arguments, shows the current CUDA device.
- With a numeric argument, changes the CUDA device focus to that device.

**dcuda sm** [<n>]
- With no arguments, shows the current CUDA SM (streaming multiprocessor).
- With a numeric argument, changes the CUDA SM focus to that SM.

**dcuda warp** [<n>]
- With no arguments, shows the current CUDA warp.
- With a numeric argument, changes the CUDA warp focus to that warp.

**dcuda lane** [<n>]
- With no arguments, shows the current CUDA lane.
- With a numeric argument, changes the CUDA lane focus to that lane.

**dcuda info-system**
Displays the CUDA devices in the system.

**dcuda info-device**
Displays currently running SMs in the current device.

**dcuda info-sm**
Displays valid warps in the current SM.

**dcuda info-warp**
Displays valid lanes in the current warp.

**dcuda info-lane**
Displays the current lane.
**dcuda focus (Bx, By, Bz),(Tx,Ty,Tz)**

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

The following abbreviations are also accepted:

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

Angle brackets are optional, but must be balanced.

**dcuda hwfocus <D/S/W/L>**

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

**Command alias**

<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: 0x0000000f/0x0000000f/0x00000000  block: (0,0,0)
```

`dcuda info-warp`

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

dcuda info-lane

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

Displaying the focus
dcuda warp sm

Output:
sm 0 warp 0
dcuda lane device

Output:
device 0 lane 3
dcuda thread

Output:
thread (1,1,0)
dcuda kernel

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

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

dcuda thread (1,1,0)

Changes the CUDA focus to the thread represented by logical coordinates 1,1,0.
New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 3, block (0,0,0), thread (1,1,0)

dcuda lane 2
Changes the CUDA focus to lane 2.
New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 2, block (0,0,0), thread (0,1,0)

dcuda lane 1 sm 0
Changes the CUDA focus to lane 1 and to SM 0.
New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 1, block (0,0,0), thread (1,0,0)

dcuda thread 0,0,0
Changes the CUDA focus to thread 0,0,0.
New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 0, block (0,0,0), thread (0,0,0)

dcuda thread 1
Changes the CUDA focus to thread 1,0,0.
New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 1, block (0,0,0), thread (1,0,0)

Using the CUDA Debugger
**ddelete**

Deletes action points

**Format**

Deletes the specified action points

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

Deletes all action points

\[ \texttt{ddelete -a} \]

**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.

**Deleting Action Points**

**Action Point > Delete All Command**
ddetach

Detaches from processes

Format

ddetach

Arguments

This command has no arguments.

Description

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

You can detach any process controlled by the CLI; the process being detached need not have been loaded with a dattach command.

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

If a single thread serves as the set, the CLI detaches the process that contains the thread. If you do not indicate a focus, the default focus is the process of interest (POI).

Command alias

<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.

Detaching from Processes
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).

**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.

Disabling Action Points
Action Point > Disable Command
**ddlopen**

Dynamically loads shared object libraries

**Format**

Dynamically loads a shared object library

```
ddlopen [-now | -lazy] [-local | -global] [-mode int] filespec
```

Displays information about shared object libraries

```
ddlopen [-list { dll-ids... }]
```

**Arguments**

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

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

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

- **-global**
  
  Includes `RTLD_LOCAL` in the `dlopen` command's mode argument. (Global makes library symbols available to libraries that the program subsequently loads.)

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

- **filespec**
  
  The shared library to load.

- **-list**
  
  Displays information about the listed DLL IDs. If you omit this option or use the `-list` without a DLL ID list, TotalView displays information about all DLL IDs.

- **dll-ids**
  
  A list of one or more DLL IDs.

**Description**

The `ddlopen` command dynamically loads shared object libraries, or lists the shared object libraries loaded using this or the Tools > Dynamic Libraries command.

For a `filespec` argument, TotalView performs a `dlopen` operation on this file in each process in the current P/T set. On the IBM AIX operating system, you can add a parenthesized library module name to the end of the `filespec` argument.
NOTE >> dlopen(3), derror(3), and other related routines are not part of the default runtime libraries on AIX, Solaris, and Red Hat Linux. Instead, they are in the libdl system library. Consequently, you must link your program using the -ldl option if you want to use the ddlopen command.

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

The -local and -global options determine if symbols from the newly loaded library are available to resolve references. If you don't use either option, TotalView uses the target operating system's default. (Linux supports only the -global option; if you don't specify an option, the default is the -local option.)

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

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

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

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

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

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

Examples

    ddlopen "mpistat.so"

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

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

    Loads the module mpistat.o in the AIX DLL library mpistat.so into all members of the current process's control group.
ddlopen -lazy -global "mpistat.so"

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

**ddlopen**

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

**Preloading Shared Libraries**

**Tools > Debugger Loaded Libraries Command**

**TV::dll Command**
ddown

Moves down the call stack

Format

```
ddown [ num-levels ]
```

Arguments

```
um-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><code>ddown</code></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.
denable

Enables action points

**Format**

Enables some action points

```
  denable action-point-list [ -block number-list ]
```

Enables all disabled action points in the current focus

```
  denable -a
```

**Arguments**

`action-point-list`

The identifiers of the action points being enabled.

`-a`

Enables all action points.

`-block number-list`

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

**Description**

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

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

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

**Command alias**

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

**Examples**

```
  denable 3 4
```

Enables two previously identified action points.

```
  dfocus {p1 p2} denable -a
```

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

```
  en -a
```

Enables all action points associated with the current focus.
fa en -a
Enables all actions points in all processes.

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

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

Enabling Action Points
Action Points > Enable Command
ddisable Command
dbarrier Command
dbreak Command
dwatch Command
**dexamine**

Displays memory contents

**Format**


**Arguments**

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

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

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

  - **char**
    - Unsigned character

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

  - **float**
    - Signed float value, either 4 or 8 byte word size

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

  - **instruction**
    - Sequence of instructions
oct
    Unsigned octal value of size 1, 2, 4, or 8 bytes

string
    String

-m
-memory_info
    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 len
-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 size
-wordsize size
    Specifies the “word size” to apply to the format. The default word size is ‘1’ for most formats. For ‘address’ format, the word size is always the size of a target pointer. The values can be 1, 2, 4, 8 or one of the following: b (byte), h (half word), w (word), or g (giant).

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

Description
    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>
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, 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.

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

Thread 1.1 hit breakpoint 4 at line 310 in "nothing2(int)"
   d1.<> dprint -nowait nothing2(9)

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

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

Thread 1.1 hit breakpoint 4 at line 310 in "nothing2(int)"
...

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

# Use the dflush command to remove the last item pushed
# onto the stack. Notice the frame associated with "11"
# is no longer there.
#
   d1.<> dflush
   d1.<> dwhere
   0 nothing2   PC=0x00012520, FP=0xffbef220 [fork.cxx#310]
   1 ***** Eval Function Call (10) ***************
   2 nothing2   PC=0x00012520, FP=0xffbef310 [fork.cxx#310]
   3 ***** Eval Function Call (9) ***************
   4 nothing2   PC=0x00012520, FP=0xffbef400 [fork.cxx#310]
   5 ***** Eval Function Call (8) ***************
   6 nothing2   PC=0x00012520, FP=0xffbef4f0 [fork.cxx#310]
   7 ***** Eval Function Call (7) ***************
   8 forker     PC=0x00013fd8, FP=0xffbef648 [fork.cxx#1120]
   9 fork_wrap  PC=0x00014780, FP=0xffbef6c8 [fork.cxx#1278]
# Use the dflush command with a suspended ID argument to remove all frames up to and including the one associated with suspended ID 9. This means that IDs 7 and 8 remain.

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

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


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

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

Format

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

    dfocus [ p/t-set ]

Executes a command in this P/T set

    dfocus p/t-set command

Arguments

p/t-set

A set of processes and threads to be the target of subsequent CLI commands.

command

A CLI command that operates on its own local focus.

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.

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

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

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

NOTE >> Instead of a P/T set, you can enter a P/T set expression. These expressions are described in “Using P/T Set Operators” in Chapter 12 of the TotalView 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

Sets and then resets command focus. A focus command that includes a dot and omits the process value uses the current process. Thus, this sequence of commands changes the focus to process 3, thread 5 (d3.5).

dfocus g dstep

Steps the current group. Although the *thread of interest* (TOI) is determined by the current focus, this command acts on the entire group that contains that thread.

dfocus {p2 p3} {dwhere ; dgo}

Performs a backtrace on all threads in processes 2 and 3, and then tells these processes to execute.

f 2.3 {f p w; f t s; g}

Executes a backtrace (dwhere) on all the threads in process 2, steps thread 3 in process 2 (without running any other threads in the process), and continues the process.

dfocus p1

Changes the current focus to include just those threads currently in process 1. The width is set to process. The CLI sets the prompt to p1.<.

dfocus a

Changes the current set to include all threads in all processes. After you execute this command, your prompt changes to a1.<. This command alters CLI behavior so that actions that previously operated on a thread now apply to all threads in all processes.

dfocus gW dstatus

Displays the status of all worker threads in the control group. The width is group level and the target is the workers group.

dfocus pW dstatus

Displays the status of all worker threads in the current focus process. The width is process level and the target is the workers group.

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

Shows all threads that are stopped at breakpoints or watchpoints.

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

Shows all stopped threads that are not stopped at breakpoints.

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

**Using Groups, Processes, and Threads**
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:

```
1b_dist
  Handle        -1000
  Ghosts        yes
  C type        $double[129][129][27]
  Fortran Type  \$double_precision(27,129,129)

bc_mask
  Handle        -999
  Ghosts        yes
  C type        long[129][129]
  Fortran Type  $integer(129,129)
```
dga bc_mask (:2,:2)

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

```
bc_mask(:,2) = {
    (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.
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.

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

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

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

### Command alias

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>dgo</td>
<td>Resumes execution</td>
</tr>
<tr>
<td>G</td>
<td>{dfocus g dgo}</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 q 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).
Continues all threads in the current process that are at the same PC as the TOI.

Continues a single thread.

Starting Processes and Threads
Process > Go Command
Thread > Go Command
dgroups

Manipulates and manages groups

**Format**

Adds members to thread and process groups

\texttt{dgroups -add [-g gid] [id-list]}

Deletes groups

\texttt{dgroups -delete [-g gid]}

Intersects a group with a list of processes and threads

\texttt{dgroups -intersect [-g gid] [id-list]}

Prints process and thread group information

\texttt{dgroups [-list] [pattern-list]}

Creates a new thread or process group

\texttt{dgroups -new [thread_or_process] [-g gid] [id-list]}

Removes members from thread or process groups

\texttt{dgroups -remove [-g gid] [id-list]}

**Arguments**

-\texttt{g gid}

The group ID on which the command operates. The \textit{gid} value can be an existing numeric group ID, an existing group name, or, if you are using the \texttt{-new} option, a new group name.

-\texttt{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 \textit{pid},\textit{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 \texttt{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 \texttt{dset} command’s discussion of the \texttt{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.

-\texttt{pattern-list}

A pattern to be matched against group names. The pattern is a Tcl regular expression.

-\texttt{thread_or_process}

Keywords that create a new process or thread group. You can specify one of the following arguments: \texttt{t}, \texttt{thread}, \texttt{p}, or \texttt{process}.

**Description**

The \texttt{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

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

• Process to a thread group, the CLI adds all of its threads.
• Thread to a process group, the CLI adds the thread’s parent process.

You can abbreviate the -add option to -a.

The CLI returns the ID of this group.

You can explicitly name the items being added by using an id-list argument. Without an id-list argument, the CLI adds the threads and processes in the current focus. Similarly, you can name the group using the -g option. Without the -g option, the CLI uses the groups in the current focus.

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

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

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

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

    dset CGROUP($mypid) $new_group_id

dgroups -delete

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

dgroups -intersect

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

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

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

The CLI returns the ID of the newly created group.

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

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

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

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

You can abbreviate the -new option to -n.


dgroups -remove

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**

```bash
dgroups -add
f tW gr -add
```

Adds the focus thread to its workers group.

```bash
dgroups -add
```

Adds the current focus thread to the current focus group.

```bash
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.

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

Adds process 9 to a user-defined group.

```bash
dgroups -delete
gr -delete -g mygroup
```

Deletes the mygroup group.

```bash
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.

```bash
f tW gr -i
```

Intersects the focus thread with the threads in its workers group.

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

Removes all nonworker threads from the mygroup group.

```bash
dgroups -list
dgroups -list
```

Displays information about all named groups; for example:

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

```bash
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
dgr -nt -g mygroup GROUP(${GROUP(1)})
Creates a new thread group named mygroup that contains all threads from all processes in the control group for process 1.
set mygroup [dgroups -new]
Creates a new process group that contains the current focus process.
dgroups -remove
dgroups -remove -g 3 3.2
Removes thread 3.2 from group 3.
f W dgroups -add
Marks the current thread as being a worker thread.
f W dgroups -r
Indicates that the current thread is not a worker thread.
dhalt

Suspends execution of processes

Format

dhalt

Arguments

This command has no arguments

Description

The `dhalt` command stops all processes and threads in the current focus.

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

Stopping Processes and Threads

Updating Process Information

Group > Halt Command

Process > Halt Command

Thread > Halt Command
**dheap**

Controls heap debugging

**Format**

Shows Memory Debugger state

```
dheap [-status ]
```

Applies a saved configuration file

```
dheap -apply_config { default | filename }
```

Shows information about a backtrace

```
dheap -backtrace [ subcommands ]
```

Compares memory states

```
dheap -compare subcommands [ optional_subcommands ]
[ process | filename [ process | filename ] ]
```

Enables or disables the Memory Debugger

```
dheap {-enable | -disable }
```

Enables or disables event notification

```
dheap -event_filter subcommands
```

Writes memory information

```
dheap -export subcommands
```

Specifies the filters the Memory Debugger uses

```
dheap -filter subcommands
```

Writes guard blocks (memory before and after an allocation)

```
dheap -guard [ subcommands ]
```

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

```
dheap -hoard [ subcommands ]
```

Displays Memory Debugger information

```
dheap -info [ subcommands ]
```

Indicates whether an address is in a deallocated block

```
dheap -is_dangling address
```

Locates memory leaks

```
dheap -leaks [ -check_interior ]
```

Enables or disables Memory Debugger event notification

```
dheap {-no}notify
```

Paints memory with a distinct pattern

```
dheap -paint [ subcommands ]
```
Enables or disables the ability to catch bounds errors and use-after-free errors retaining freed memory blocks

`dheap -red_zones [ subcommands ]`

Enables or disables allocation and reallocation notification

`dheap -tag_alloc subcommand [ start_address [ end_address ] ]`

Displays the Memory Debugger version number

`dheap -version`

**Description**

The `dheap` command is described in the Batch Scripting section of “Locating Memory Problems” in the MemoryScape documentation.
**dhistory**

Performs actions upon ReplayEngine

**Format**


**Arguments**

- **-info**
  Writes ReplayEngine information including the current time, the live time, and whether the process is in replay or record mode. If you do not enter any arguments, **-info** is the default.

- **-get_time**
  Displays the current time. The output of this command shows an integer value followed by an address. The first integer value is a virtual timestamp. This virtual timestamp does not refer to the exact point in time; it has a granularity that is typically a few lines of code. The address value is a PC value that corresponds to a precise point within that block of code.

- **-go_time time**
  Places the process back to the virtual time specified by the *time* integer argument. The *time* argument is a virtual timestamp as reported by **dhistory -get_time**. You cannot use this command to move to a specific instruction but you can use it to get to within a small block of code (usually within a few lines of your intended point in execution history). This command is typically used either for roughly bookmarking a point in code or for searching execution history. It may need to be combined with stepping and **duntil** commands to return to an exact position.

- **-go_live**
  Places the process back to the PC and back into record mode. You can resume your “regular” debugging session.

- **-enable**
  Enables ReplayEngine, starting the history

- **-disable**
  Disables ReplayEngine, ending the history.

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

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

  To reload a recording at startup:

  ```
  totalview executable recording-file
  ```

  To reload a recording file when TotalView is running:
**dattach filename -c recording-file**

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

**Description**

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

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

**Command alias**

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

**Examples**

`dhistory`

Typing this command displays the following information:

History info for process 1

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

Holds threads or processes

**Format**

Holds processes

`dhold -process`

Holds threads

`dhold -thread`

**Arguments**

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

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

**Description**

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

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

**Examples**

`f W HT`

Holds all worker threads in the focus group.

`f s HP`

Holds all processes in the share group.

`f $mygroup/ HP`

Holds all processes in the group identified by the contents of `mygroup`.
Holding and Releasing Processes and Threads
Group > Hold Command
Process > Hold Command
Thread > Hold Command
Group > Release Command
Process > Release Threads Command
dunhold Command
**dkill**

Terminates execution of processes

**Format**

dkill [-remove ]

**Arguments**

- remove

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

**Description**

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

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

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

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

**Command alias**

<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.

**Starting Your Program**

**Restarting Programs**

**Deleting Programs**

**Group > Delete Command**

**Group > Restart Command**
**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**:

```
dlappend foo 1 3 5
```

In contrast, the CLI command displays an error message:
```
dlappend foo 1 3 5
```

**Examples**  
dlappend TV::process_load_callbacks my_load_callback

  Adds the **my_load_callback** function to the list of functions in the TV::process_load_callbacks variable.
dlist Displays source code lines

Format
Displays source code relative to the current list location
   dlist [ -n num-lines ]
Displays source code relative to a named place
   dlist breakpoint-expr [ -n num-lines ]
Displays source code relative to the current execution location
   dlist -e [ -n num-lines ]

Arguments
   -n num-lines
      Displays this number of lines rather than the default number. (The default is the value of the MAX_LIST variable.)
      If num-lines is negative, the CLI displays lines before the current location, and additional dlist commands show
      preceding lines in the file rather than following lines.
      This option also sets the value of the MAX_LIST variable to num-lines.

   breakpoint-expr
      The location at which the CLI begins displaying information. In most cases, specify this location as a line number
      or as a string that contains a file name, function name, and line number, each separated by # characters; for ex-
      ample: file#func#line. (For more information, see “Qualifying Symbol Names” in Chapter 12 of the TotalView
      User Guide.) The CLI creates defaults if you omit parts of this specification.
      If you enter a different file, it is used for future display. This means that if you want to display information relative
      to the current thread's execution point, use the -e option to dlist.
      If the breakpoint expression evaluates to more than one location, TotalView chooses one.
      For other ways to enter these expressions, see “Breakpoint Expressions” on page 44. If you name more than
      one address, TotalView picks one.

   -e
      Sets the display location to include the current execution point of the thread of interest (TOI). If you use dup and
      ddown commands to select a buried stack frame, this location includes the PC (program counter) for that stack
      frame.

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

If you enter a file or procedure name, the listing begins at the file or procedure's first line.
The default focus for this command is thread level. If your focus is at process level, TotalView acts on each thread in the process.

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

Tabs in the source file are expanded as blanks in the output. The `TAB_WIDTH` variable controls the tab stop width, which defaults to 8. If `TAB_WIDTH` is set to -1, no tab processing is performed, and the CLI displays tabs using their ASCII value.

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

- `@`  
  An action point is set at this line.

- `>`  
  The PC for the current stack frame is at the indicated line and this is the leaf frame.

- `=`  
  The PC for the current stack frame is at the indicated line and this is a buried frame; this frame has called another function so that this frame is not the active frame.

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

Here are some general rules:

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

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

### Command alias

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlist</td>
<td></td>
<td>Displays lines</td>
</tr>
</tbody>
</table>

### Examples

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

```
dlist
```

Displays 20 lines of source code, beginning at the current list location. The list location is incremented by 20 when the command completes.
dlist 10
Displays 20 lines, starting with line 10 of the file that corresponds to the current list location. Because this uses an explicit value, the CLI ignores the previous command. The list location is changed to line 30.

dlist -n 10
Displays 10 lines, starting with the current list location. The value of the list location is incremented by 10.

dlist -n -50
Displays source code preceding the current list location; shows 50 lines, ending with the current source code location. The list location is decremented by 50.

dlist do_it
Displays 20 lines in procedure do_it. Changes the list location to be the 20th line of the procedure.

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

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

f 1.2 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 *hostname* ]

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

**Arguments**

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

- **-r *hostname***
  
  The host on which the process will run. The CLI launches a TotalView Debugger Server on the host machine if one is not already running there. (See Chapter 5 and Chapter 6 in the TotalView User Guide for information on the server launch commands.)

{ -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.

- **-mpi *starter***
  
  Names the starter configuration, either a configuration supplied by TotalView or one created and named using the TV::parallel_configs variable. For information on defining configurations, see .

- **-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.

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

**-parallel_attach_subset subset_specification**
Defines a list of MPI ranks to attach to when an MPI job is created or attached to. The list is space-separated; each element can have one of three forms:

- **rank**: specifies that rank only
- **rank1-rank2**: specifies all ranks between rank1 and rank2, inclusive
- **rank1-rank2:stride**: specifies every strideth rank between rank1 and rank2

A rank must be either a positive decimal integer or **max** (the last rank in the MPI job).

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

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

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

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

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

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

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

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

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

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

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

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

---

**Loading Executables**

**File > New Program Command**

**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` is sensitive to the focus. Note this four-process program:

```
<table>
<thead>
<tr>
<th>process</th>
<th>text</th>
<th>data</th>
<th>heap</th>
<th>stack</th>
<th>[stack_vm]</th>
<th>vm_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(9271):</td>
<td>1128.54K</td>
<td>16.15M</td>
<td>9976</td>
<td>10432</td>
<td>[16384]</td>
</tr>
</tbody>
</table>
```

**image information:**

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

**Examples**

`dmstat` is sensitive to the focus. Note this four-process program:

```
<table>
<thead>
<tr>
<th>process</th>
<th>text</th>
<th>data</th>
<th>heap</th>
<th>stack</th>
<th>[stack_vm]</th>
<th>vm_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(9271):</td>
<td>1128.54K</td>
<td>16.15M</td>
<td>9976</td>
<td>10432</td>
<td>[16384]</td>
</tr>
</tbody>
</table>
```

**image information:**

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

`dfocus a dmstat`

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

```
<table>
<thead>
<tr>
<th>process</th>
<th>text</th>
<th>data</th>
<th>heap</th>
<th>stack</th>
<th>[stack_vm]</th>
<th>vm_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(9271):</td>
<td>1128.54K</td>
<td>16.15M</td>
<td>9976</td>
<td>10432</td>
<td>[16384]</td>
</tr>
</tbody>
</table>
```

**image information:**

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

**Column Description**

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

- **vm_size**: The sum of the sizes of the mappings in the process's address space.
MemoryUsagePage
Opening MemoryScape to examine memory usage
CLI Commands

**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 141.
**CLI Commands**

<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.
Creating a Process by Single Stepping
Stepping and Setting Breakpoints
Using Stepping Commands
dnexti Command
dstep Command
dfocus Command
Group > Next Command
Process > Next Command
Thread > Next Command
**dnexiti**

Steps machine instructions, stepping over subroutines

**Format**

`dnexiti [-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 `dnexiti` command executes machine-level instructions; that is, it advances the program by a single instruction. However, if the instruction invokes a subfunction, the `dnexiti` 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 `dnexiti` operations to perform. If you do not specify `num-steps`, the default is 1.

The `dnexiti` 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 141.
## 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>NI</td>
<td>{dfocus g dnexti}</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 instruction. The rest of the control group runs freely.</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.

f t n

Steps the thread one machine-level instruction.

dfocus 3. dnexti
Steps process 3 one machine-level instruction.

Creating a Process by Single Stepping
Stepping and Setting Breakpoints
Using Stepping Commands

dnext Command
dstep Command
dfocus Command

Process > Next Instruction Command
Thread > Next Instruction Command
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 141 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.

**Executing to the Completion of a Function**

**Group > Out Command**

**Process > Out Command**
Thread > Out Command
dprint

Evaluates and displays information

Format

Prints the value of a variable

\texttt{dprint [-nowait] [-slice "slice_expr"] variable}

Prints the value of an expression

\texttt{dprint [-nowait] [-slice "slice_expr"] [-stats [-data]] expression}

Arguments

- \texttt{-nowait}
  Tells TotalView to evaluate the expression in the background. Use \texttt{TV::expr} to obtain the results, as they are not displayed.

- \texttt{-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, \texttt{\[\texttt{100:110}\]}.

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

- \texttt{-data}
  Returns the results of \texttt{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.

  \textbf{Note:} This switch can be used \textit{only} in conjunction with the \texttt{--stats} switch.

\texttt{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,

\texttt{expression}

A source-language expression to evaluate and print. Because \texttt{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 \texttt{dprint} command evaluates and displays a variable or an expression. The CLI interprets the expression by looking up the values associated with each symbol and applying the operators. The result of an expression can be a scalar value or an aggregate (array, array slice, or structure).

If an event such as a $stop, SEGV, breakpoint occurs, the \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:
dfocus tdpid.dtid TV::expr get susp-eval-id

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

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

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

Since the dprint command can generate a considerable amount of output, you might want to use the capture on page 26 command described on to save the output to a variable.

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

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

Arrays print in a similar manner; for example:

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

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

```
d.l<> p -slice "\[10:20\]" random
random slice: (10:30) = {
   (10) = 0.479426
   (11) = 0.877583
   (12) = 0.564642
   (13) = 0.825336
   (14) = 0.644218
   (15) = 0.764842
   (16) = 0.717356
   (17) = 0.696707
```
CLI Commands

dprint

(18) = 0.783327
(19) = 0.62161
(20) = 0.841471

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

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

The following example illustrates the output from dprint -stats command:

    d1.<> dprint -stats twod_array

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

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

    Checksum: 41071

By adding the -data switch,

    d1.<> dprint -stats -data twod_array

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

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

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

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

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

- **first process or thread:**
  - expression result

- **second process or thread:**
  - expression result

...  

- **last process or thread:**
  - expression result

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

```
float bint;
```

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

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

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

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

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

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

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

```
dprint \$r1
```
You must precede the dollar sign (\$) with a backslash to escape it since the register's name includes the \$. This \$ is not the standard indicator that tells Tcl to fetch a variable's value. Chapter 11, “Architectures,” on page 385 lists the mnemonic names assigned to -registers.

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

Command alias

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

Examples

dprint scalar_y
  Displays the values of variable scalar_y in all processes and threads in the current focus.
p argc
  Displays the value of argc.
p argv
  Displays the value of argv, along with the first string to which it points.
p (argv[argc-1])
  Prints the value of argv[argc-1]. If the execution point is in main(), this is the last argument passed to main().
dfocus p1 dprint scalar_y
  Displays the values of variable scalar_y for the threads in process 1.
f 1.2 p arrayx
  Displays the values of the array arrayx for the second thread in process 1.
for {set i 0} {$i < 100} {incr i} {p argv\[$i\]}
  If main() is in the current scope, prints the program's arguments followed by the program's environment strings.
f {t1.1 t2.1 t3.1} dprint {f()}
  Evaluates a function contained in three threads. Each thread is in a different process:
  Thread 1.1:
  f(): 2
  Thread 2.1:
  f(): 3
  Thread 3.1:
  f(): 5

f {t1.1 t2.1 t3.1} dprint -nowait {f()}
1
Evaluates a function without waiting. Later, you can obtain the results using `TV::expr`. The number displayed immediately after the command, which is “1”, is the `susp-eval-id`. The following example shows how to get this result:

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

Examining and Changing Data

Examining Arrays

Evaluating Expressions

Tools > Evaluate Command

TV::errorCodes Command

TV::expr Command
CLI Commands

**dptsets**

Shows the status of processes and threads

**Format**

*dptsets* [*ptset_array*] ...

**Arguments**

*ptset_array*

An optional array that indicates the P/T sets to show. An element of the array can be a number or it can be a more complicated P/T expression. (For more information, see “Using P/T Set Operators” in Chapter 12 of the *TotalView 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 12 of the *TotalView User Guide*), and the elements’ array indices are strings that label each element’s section in the output.

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

**Examples**

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

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

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

watchpoint:

running:

held:

error:

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

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

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

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

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

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

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

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

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

d.1<> set mystat(system) a-gW
d.1<> set mystat(reallystopped) \stopped(a)-breakpoint(a)-watchpoint(a)
d.1<> dptsets t mystat
system:
Threads in process 1 [regress/fork_loop]:
1.-1: 21587.[-1] Running PC=0x3ff805c6998
Threads in process 2 [regress/fork_loop.1]:
2.-1: 15224.[-1] Stopped PC=0x3ff805c6998
2.-2: 15224.[-2] Stopped PC=0x3ff805c669c

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

Using Groups, Processes, and Threads
drerun

Reverts procedures

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 131 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.
Starting Processes and Threads
Command Arguments
Process > Startup Parameters
  drun 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.

- **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.
**dcalltree Command**

Tools > Create Checkpoint Command

Tools > Restart Checkpoint Command
drun

Starts or restarts processes

Format

\texttt{drun [ cmd\_arguments ][ in\_operation infile ] [ out\_operations outfile ] [ error\_operations errfile ]}

Arguments

\texttt{cmd\_arguments}

The argument list passed to the process.

\texttt{in\_operation}

Names the file from which the CLI reads input.

\texttt{< infile}

Reads from \texttt{infile} instead of \texttt{stdin}. \texttt{infile} indicates a file from which the launched process reads information.

\texttt{out\_operations}

Names the file to which the CLI writes output. In the following, \texttt{outfile} indicates the file into which the launched processes writes information.

\texttt{> outfile}

Sends output to \texttt{outfile} instead of \texttt{stdout}.

\texttt{>& outfile}

Sends output and error messages to \texttt{outfile} instead of \texttt{stdout} and \texttt{stderr}.

\texttt{>>> outfile}

Appends output and error messages to \texttt{outfile}.

\texttt{>> outfile}

Appends output to \texttt{outfile}.

\texttt{error\_operations}

Names the file to which the CLI writes error output. In the following, \texttt{errfile} indicates the file into which the launched processes writes error information.

\texttt{2> errfile}

Sends error messages to \texttt{errfile} instead of \texttt{stderr}.

\texttt{2>>errfile}

Appends error messages to \texttt{errfile}.

Description

The \texttt{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 \texttt{drerun} command to restart the program.
The arguments to this command are similar to the arguments used in the Bourne shell.

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

**ARGS_DEFAULT**

The CLI sets this variable if you use the *-a* command-line option when you started the CLI or TotalView. (This option passes command-line arguments that TotalView uses when it invokes a process.) This variable holds the default arguments that TotalView passes to a process when the process has no default arguments of its own.

**ARGS(dpmid)**

An array variable that contains the command-line arguments. The index *dpid* is the process ID. This variable holds a process's default arguments. It is always set by the *drun* command, and it also contains any arguments you used when executing a *drerun* command.

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

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

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

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

**Issues When Using Starter Programs**

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

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

```
drun < in.txt
```
Command alias

<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.

Starting Processes and Threads
Command Arguments
Process > Startup Parameters
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.

**Loading a Session Using the Sessions Manager**

**Managing Sessions**

`dattach`
**dset**

Changes or views CLI variables

**Format**

Changes a CLI variable

```
dset debugger-var value
```

Views current CLI variables

```
dset [ debugger-var ]
```

Sets the default for a CLI variable

```
dset -set_as_default debugger-var value
```

**Arguments**

`debugger-var`

Name of a CLI variable.

`value`

Value to be assigned to `debugger-var`.

`-set_as_default`

Sets the value to use as the variable's default. This option is most often used by system administrators to set site-specific defaults in the global `.tvdrc` startup script. Values set using this option replace the CLI built-in default.

**Description**

The `dset` command sets the value of CLI debugger variables. CLI and TotalView variables are described in Chapter 5, “TotalView Variables,” on page 247.

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

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

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

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

```
dset TV::
```

or:

```
dset TV::GUI::
```
You need to type the double colons at the end of this example when obtaining listings for a namespace. Without them, Tcl assumes that you are requesting information on a variable. For example, `dset TV::GUI` looks for a variable named GUI in the TV namespace.

**Using -set_as_default**

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

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

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

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

**Examples**

```
dset PROMPT "Fixme\% 
Sets the prompt to Fixme% followed by a space.
dset *
Displays all CLI variables and their current settings.
dset VERBOSE
Displays the current setting for output verbosity.
dset EXECUTABLE_PATH ..\test_dir;$EXECUTABLE_PATH
Places ..\test_dir at the beginning of the previous value for the executable path.
dset -set_as_default TV::server_launch_string  
   {/use/this/one/tvdsvr}
Sets the default value of the TV::server_launch_string. If you change this value, you can later select the Defaults button within the File > Preferences Launch String Page to reset it to this value.
dset TV::GUI::fixed_font_size 12
Sets the TotalView GUI to display information using a 12-point, fixed-width font. Commands such as this are often found in a startup file.
```

**TotalView Variables**

dlappend Command
**dstatus**

Shows current status of processes and threads

**Format**


dstatus

dstatus [-g]
dstatus [-group_by process_state | replay | pheld | thread_state | pc, | function | line | apid | theld | stop_reason ]

**Arguments**

-g

Alias for -group_by.

-group_by

Reduces the display based on the following process-level or thread-level arguments. The reduction 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:**

process_state

Limits the display to the state of the process.

replay

Groups by replay mode. A process can be in three replay states: Replay Unavailable, Replay, or -Record.

pheld

Groups the processes as either Held or UnHeld.

**Thread level:**

thread_state

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

apid

The action point identifier that the thread's pc is on. If the thread is not at an action point, it will be grouped as ap (none).

theld

Threads grouped as either Held or UnHeld.
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 compressed list that is shown in a reduced dstatus display. If a reduction results in exceeding the ptlist_element_count, an ellipsis is appended. For instance, if ptlist_element_count is set to 5:

[p1-4.1, p2.2, p3-4.3, p5.4, p6.1-2, ...]

To change the default value, use the TotalView State variable ptlist_element_threshold. For example:

dset TV::ptlist_element_threshold 10

-levels

The number of levels to show for a set of properties. If no levels are specified, then each property is reduced on a new line with indentation. If the number of levels is less than the number of specified properties, then the remaining properties are shown in a single reduction on one line.

-v

Show verbose output in the reduced display. Without -v, full paths of filenames and line numbers are not displayed.

-detail

Force full detailed information for the current state of each process and thread in the current focus. This option affects the amount of information displayed from grouping by function.

Description

With the -group_by option, the dstatus command displays an aggregated view of the process and thread state in the current focus. To make the display more useful, you can reduce it based on specific properties, provided as arguments as described above. The full detail shows the current state of each process and thread in the current focus. ST is aliased to dfocus g dstatus and acts as a group-status command. Type help ptset for more information.

If you have not changed the focus, the default is process. In this case, the dstatus command shows the status for each thread in process 1. In contrast, if you set the focus to g1.<, the CLI displays the status for every thread in the control group. When you limit thread state display by certain properties, the output is displayed as a compressed thread list, or ptlist.

Compressed List Syntax (ptlist)

A compressed ptlist consists of a process and thread count, followed by square-bracket-enclosed list of process and thread ranges separated by dot (.). If the thread range is missing, it's merely a compressed list of processes and it is referred to as a plist.

If the process range starts with the letter p, the process IDs are TotalView DPIDs (debugger unique process identifiers); otherwise, they are the MPI rank for the process, MPI_COMM_WORLD.
The thread IDs are always TotalView DTIDs (debugger unique thread identifiers). For example, the compressed `ptlist 5:13[0-3.1-3, p1.1]` indicates that there are five processes and 13 threads in the list. The process and thread range `0-3.1-3` indicates MPI rank processes 0 through 3, each with DTIDs 1 through 3. The process range `p1.1` indicates process DPID 1 and thread DTID 1, normally the MPI starter process named `mpirun`.

### Command alias

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>st</td>
<td>dstatus</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]
```

```
fa st
```

Displays the status for all threads in all processes.

```
f p1 st
```

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

```
ST
```

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

```
1:  773686  Stopped  [fork_loop_64]
  1.1:773686.1  Stopped  PC=0x0d9cae64
  1.2:773686.2  Stopped  PC=0x0d9cae64
  1.3:773686.3  Stopped  PC=0x0d9cae64
  1.4:773686.4  Stopped  PC=0x0d9cae6
2:  779490  Stopped  [fork_loop_64.1]
  2.1:779490.1  Stopped  PC=0x0d9cae64
  2.2:779490.2  Stopped  PC=0x0d9cae64
  2.3:779490.3  Stopped  PC=0x0d9cae64
  2.4:779490.4  Stopped  PC=0x0d9cae64
```

```
f W st
```

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

```
f W ST
```
CLI Commands

**dstatus**

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).

**dl.<> 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]
```

**Using the Root Window**

**Viewing Process and Thread State**

**The Root Window**

**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.

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.
Creating a Process by Single Stepping
Stepping and Setting Breakpoints
Using Stepping Commands
Command
`dnext` Command
`dfocus` Command
Group > Step Command
Process > Step Command
Thread > Step Command

Examples

`dstepi`
Executes the next machine instruction, stepping into any procedure call it encounters. Although the CLI only steps the current thread, other threads in the process run.

`si 15`
Executes the next 15 instructions.

`f p1.2 dstepi`
Steps thread 2 in process 1 by one instruction, and resumes execution of all other threads in process 1; they halt as soon as thread 2 in process 1 executes its instruction.

`f t1.2 si`
Steps thread 2 in process 1 by one instruction. No other threads in process 1 execute.
**dunhold**

Releases a held process or thread

**Format**

Releases a process

```
dunhold -process
```

Releases a thread

```
dunhold -thread
```

**Arguments**

- `-process`
  - 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 process of interest (POI)</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 thread of interest (TOI)</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.

```
http; uht
```

Holds all threads in the focus process except the TOI.

**Holding and Releasing Processes and Threads**

**Starting Processes and Threads**

**Group > Release Command**

**Process > Release Threads Command**

**Thread > Hold Command**

**dhold Command**
**dunset**

Restores default settings for variables

**Format**
Restores a CLI variable to its default value

```
dunset debugger-var
```

Restores all CLI variables to their default values

```
dunset -all
```

**Arguments**
- **debugger-var**
  
  Name of the CLI variable whose default setting is being restored.
- **-all**
  
  Restores the default settings of all CLI variables.

**Description**

The **dunset** command reverses the effects of any previous **dset** commands, restoring CLI variables to their default settings. See **Chapter 5, “TotalView Variables,”** on page 247 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.
```
**duntil**

Runs the process until a target place is reached

**Format**

Runs to a line

```
duntil [ -back ] line-number
```

Runs to an address

```
duntil [ -back ] -address addr
```

Runs into a function

```
duntil [ -back ] proc-name
```

**Arguments**

- **-back**
  
  (ReplayEngine only) Steps to the previous instruction, moving into subroutines that called the current function. This option can be abbreviated to **-b**.

- **line-number**
  
  A line number in your program.

- **-address addr**
  
  An address in your program.

- **proc-name**
  
  The name of a procedure, function, or subroutine in your program.

**Description**

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

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

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 breakpoint. This lets you sync all the processes in a group in preparation for group-stepping them.

**Thread group**

Runs the process (for **p** width) or the control group (for **g** width) and waits until all the running threads in the group of interest arrive at the goal.

There are some differences in the way processes and threads run using the **duntil** command and other stepping commands:
• **Process Group Operation**: TotalView examines the TOI to see if it is already at the goal. If it is, TotalView does not run the POI. Similarly, TotalView examines all other processes in the share group, and runs only processes without a thread at the goal. It also runs members of the control group not in the share group.

• **Group-Width Thread Group Operation**: TotalView identifies all threads in the entire control group that are not at the goal. Only those threads run. Although TotalView runs share group members in which all worker threads are already at the goal, it does not run the workers. TotalView also runs processes in the control group outside the share group. The `duntil` command operation ends when all members of the focus thread group are at the goal.

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

### Command alias

<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.
Executing to a Selected Line
Using_groups, Processes, and Threads
Using Run To and duntil Commands
Group > Run To Command
Process > Run to Command
Thread > Run To Command
CLI Commands

dup

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>dup</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 p1 u 5
Moves up five levels in the call stack for each thread involved in process 1. If fewer than five levels exist, the CLI moves up as far as it can.
**dwait**

Blocks command input until the target processes stop

**Format**

dwait

**Arguments**

This command has no arguments

**Description**

The `dwait` command 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 | -t] [[-l lang] -e expr] [-t type]
```

Defines a watchpoint for an address

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

**Arguments**

**variable**

A symbol name corresponding to a scalar or aggregate identifier, an element of an aggregate, or a dereferenced pointer.

**-address addr**

An absolute address in the file.

**-length byte-count**

The number of bytes to watch. If you enter a variable, the default is the variable's byte length.

If you are watching a variable, you need to specify only the amount of storage to watch if you want to override the default value.

**-g**

Stops all processes in the process's control group when the watchpoint triggers.

**-p**

Stops the process that hit this watchpoint.

**-t**

Stops the thread that hit this watchpoint.

**-l lang**

Specifies the language in which you are writing an expression. The values you can use for `lang` are `c`, `c++`, `f7`, `f9`, and `asm`, for C, C++, FORTRAN 77, Fortran-9x, and assembler, respectively. If you do not use a language code, TotalView picks one based on the variable's type. If you specify only an address, TotalView uses the C language.

Not all languages are supported on all systems.

**-e expr**

When the watchpoint is triggered, evaluates `expr` in the context of the thread that hit the watchpoint. In most cases, you need to enclose the expression in braces (`{}`).

**-t type**

The data type of `$oldval/$newval` in the expression. If you do not use this option, TotalView uses the variable's datatypes. 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, IBM PowerPC computers running Linux Power, and Hewlett Packard (HP) computers running or HP-UX.

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

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

To obtain a variable’s address if your application demands that you specify a watchpoint with an address instead of a variable name:

- `dprint &variable`
- `dwhat variable`

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

See “Using Watchpoints” in the *TotalView User Guide* for additional information on watchpoints.

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

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

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

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

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

<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.

**dwatch *ptr**

Watches the address stored in pointer `ptr` at the time the watchpoint is defined, for changes made by process 2. Only process 2 is stopped. The watchpoint location does not change when the value of `ptr` changes.

**dwatch {*ptr}**

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

**dfocus {p2 p3} wa *ptr**

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

**dfocus {p2 p3 p4} dwatch -p *ptr**

Watches the address pointed to by `ptr` in processes 2, 3, and 4. The `-p` option indicates that TotalView only stops the process triggering the watchpoint.

**wa * aString -length 30 -e {goto $447}**

Watches 30 bytes of data beginning at the location pointed to by `aString`, if any of these bytes change, execution control transfers to line 447.

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

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

**wa my_vbl -e {if (my_vbl == 0x11ffff38) stop;}**

Performs the same function as the previous example. This example tests the variable directly rather than by using the `$newval` variable.

---

**Using Watchpoints**

**Writing Code Fragments**

**Tools > Watchpoint Command**

**dactions Command**
dwhat

Determines what a name refers to

Format

dwhat symbol-name

Arguments

symbol-name

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

Description

The dwhat command displays the name and description of a named entity in a program.

NOTE >> To view information on CLI variables or aliases, use the dset or alias -commands.

The focus constrains the query to a particular context.

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

Command alias

<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.cxx#snore 
  (Scope class: Any) Address class: auto_var 
  (Local variable)

wh timeval

In process 1: Type name: struct timeval; Size: 8 bytes; 
  Category: Structure
  Fields in type:
  { tv_sectime_t(32 bits)
```plaintext
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)
```

---

CLI Commands / dwhat 157

---

RogueWave
field_sub = {
    field1_int = 0xc0002980 (-1073731200)
    <padding> = '\377' (0xff, or -1)
    <padding> = '\003' (0x03, or 3)
    <padding> = '\000' (0x00, or 0)
    <padding> = '\000' (0x00, or 0)
    field2_long = 0x0000000000000000 (0)
}

wh s_vbl

In thread 2.3
Name: s_vbl; Type: struct check_struct; 
    Size: 80 bytes; Addr: 0x11ffff240
    Scope: #check_structs.cxx#main 
    Scope class: Any)
    Address class: auto_var (Local variable)

wh check_struct

In process 2:
Type name: struct check_struct; 
    Size: 80 bytes; Category: Structure
    Fields in type:
    {
        field1_int(int(32 bits)
        field2_char(char(8 bits)
        field2_chars$string[2](16 bits)
        <padding>$char(8 bits)
        field3_int(int(32 bits)
        field2_uchar(unsigned char(8 bits)
        <padding>$char[3](24 bits)
        field_substruct sub_st(320 bits){
            field1_int(int(32 bits)
            <padding>$char[4](32 bits)
            field2_longlong(64 bits)
            field2_ulong(unsigned long(64 bits)
            field3_uint(unsigned int(32 bits)
            enum en_check (32 bits)
            field3_doubledouble(64 bits)
        }
    }

    dstatus Command
    dwhere Command
View > Lookup Variable Command
dwhere

Displays the current execution location and call stack

Format

Displays locations in the call stack

```
dwhere [ -level level-num ] [ num-levels ] [ -args ] [ -locals ] [ -registers ] [ -noshow_pc ] [ -noshow_fp ] [ -show_image ] [ -group_by property ]
```

Displays all locations in the call stack

```
dwhere -all [ -args ] [ -locals ] [ -registers ] [ -noshow_pc ] [ -noshow_fp ] [ -show_image ]
```

Arguments

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

- **-level level-num**
  - Sets the level at which dwhere starts displaying information.

- **num-levels**
  - Restricts output to this number of levels of the call stack. By default, all levels are shown.

- **-args**
  - Displays argument names and values in addition to program location information. By default, the arguments are not shown.

- **-locals**
  - Displays each frame’s local variables. By default, the local variable information is not shown.

- **-noshow_pc**
  - Does not show the PC. By default, the PC value is shown.

- **-noshow_fp**
  - Does not show the FP. By default, the FP value is shown.

- **-registers**
  - Displays each frame’s registers. By default, the register information is not shown.

- **-show_image**
  - Shows the executable name as well as the file name. By default, dwhere displays the associated image information if the source line cannot be found.

- **-group_by 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).

This option requires a property argument to control the “equivalence” relationship of stack frames across the threads. See The -group_by Option 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.

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.
- `function+pc (f+pc)`: Equivalence based on the PC value 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.

The `dwhere` command displays the current execution location(s) and the backtrace(s) for the threads in the current focus. 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`.

**Command alias**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>dwhere</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 $fp1 w 5$ because specifying a process width tells the $dwhere$ command to ignore the thread indicator.

$w 1 -a$

Displays the current execution locations (one level only) of threads in the current focus, together with the names and values of any arguments that were passed into the current process.
**dworker**

Adds or removes a thread from a workers group

**Format**

```dworker { number | boolean }```

**Arguments**

- **number**
  - If positive, marks the thread of interest (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, 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><code>{dworker false}</code></td>
<td>Removes the focus thread from the workers group</td>
</tr>
<tr>
<td>wot</td>
<td><code>{dworker true}</code></td>
<td>Inserts the focus thread into the workers group</td>
</tr>
</tbody>
</table>

Organizing Chaos
Creating Groups
Setting Group Focus
dgroups Command
exit

Terminates the debugging session

Format
exit [-force]

Arguments
-force

Exits without asking permission. This is most often used in scripts.

Description
The exit command ends the debugging session.

After you enter this command, the CLI confirms that you wish to exit, then exits. If you entered the CLI from the TotalView GUI, this command also closes the GUI window.

NOTE >> If you invoked the CLI from within the TotalView GUI, pressing Ctrl+D closes the CLI window without exiting from TotalView.

TotalView destroys all processes and threads that it makes. Any processes that existed prior to the debugging session (that is, TotalView attached to them because you used the 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.

Exiting from TotalView
File > Exit Command
quit Command
help

**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:

```
dl.\> he a
```

Ambiguous help topic "a". Possible matches:

```
- alias accessors arguments addressing_expressions
```

```
dl.\> he ac
```

"ac" has been aliased to "dactions":

```
dactions [ bp-ids ... ] [ -at <source-loc> ] [ -disabled | \n- enabled ]
```

Default alias: ac

```
...
dl.\> 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 `exit` 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.

**Exiting from TotalView**

- **File > Exit Command**
  
  **exit Command**
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

```
spurs break [ 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 | lfqueue 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 43.

breakpoint-id

The action point ID to delete
**eaSpurs**
- The kernel context at this PPU address

**eaBarrier**
- The barrier object at this PPU address

**eaEventFlagSet**
- The event flag object at this PPU address

**eaLFQueue**
- The lfqueue object at this PPU address

**eaQueue**
- The queue object at this PPU address

**eaSemaphore**
- The semaphore object at this PPU address

**eaTaskset**
- The taskset at this PPU address

**taskID**
- The task at this index

**Description**

Modeled after the GDB “spurs” command, the **spurs** command was created so that developers who are familiar with 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-expexpr*

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

`cell_spurs_print_kernel` is an alias for this command.

spurs print 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.
Chapter 3

CLI Namespace Commands

Command Overview

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

Accessor Functions

The following functions, all within the TV:: namespace, access and set TotalView properties:

- **actionpoint**: Accesses and sets action point properties.
- **expr**: Manipulates values created by the `dprint -nowait` command.
- **focus_groups**: Returns a list containing the groups in the current focus.
- **focus_processes**: Returns a list of processes in the current focus.
- **focus_threads**: Returns a list of threads in the current focus.
- **group**: Accesses and sets group properties.
- **process**: Accesses and sets process properties.
- **scope**: Accesses and sets scope properties.
- **symbol**: Accesses and sets symbol properties.
- **thread**: Accesses and sets thread properties.
- **type**: Accesses and sets data type properties.
- **type_transformation**: Accesses and defines type transformations.
Helper Functions

The following functions, all within the TV:: namespace, are most often used in scripts:

- **dec2hex**: Converts a decimal number into hexadecimal format.
- **dll**: Manages shared libraries.
- **errorCodes**: Returns or raises TotalView error information.
- **hex2dec**: Converts a hexadecimal number into decimal format.
- **read_symbols**: Reads shared library symbols.
- **respond**: Sends a response to a command.
- **source_process_startup**: Reads and executes a .tvd file when TotalView loads a process.
**actionpoint**

Sets and gets action point properties

**Format**

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

**Arguments**

*action*

The action to perform, as follows:

**commands**

Displays the subcommands that you can use. The CLI responds by displaying these four *action* subcommands. There are no arguments to this subcommand.

**get**

Retrieves the values of one or more action point properties. The *other-args* argument can include one or more property names. The CLI returns values for these properties in a list whose order is the same as the names you enter.

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

**properties**

Lists the action point properties that TotalView can access. There are no arguments to this subcommand.

**set**

Sets the values of one or more properties. The *other-args* argument contains property name and value pairs.

*object-id*

An identifier for the action point.

*other-args*

Arguments that the get and set actions use.

**Description**

The TV::actionpoint command lets you examine and set the following action point properties and states:

**address**

The address of the action point.

**block_count**

The number of addresses associated with an actionpoint.

A single line of code can generate multiple instruction sequences. For example, there may be several entry points to a subroutine, depending on where the caller is. This means that an actionpoint can be set at many addresses even if you are placing it on a single line.

Internally, a block represents one of these addresses.
block_enabled

Each individual actionpoint block is an instruction that TotalView may replace with a trap instruction. (When a trap instruction is encountered, the operating system passes control to the debugger.)

Each block can be enabled or disabled separately. This property type returns a list with in which 1 indicates if the block is enabled and 0 if it is disabled.

This is the only property that can be set from within TotalView. All others are read-only.

conflicted

Indicates that another action point shares at least one of the action point blocks. If this condition exists, the block is conflicted. If a block is conflicted, TotalView completely disables the action point.

The conflicted property is 1 if the actionpoint is conflicted, and 0 if it is not.

context

A string that totally identifies an action point.

The location of every action point is represented by a string. Even action points set by clicking on a line number are represented by strings. (In this case, the string is the line number.)

Sometimes, this string is all that is needed. Usually, however, more context is needed. For example, a line number needs a file name.

enabled

A value (either 1 or 0) indicating if the action point is enabled. A value of 1 means enabled. (settable)

expression

The expression to execute at an action point. (settable)

id

The ID of the action point.

language

The language in which the action point expression is written.

length

The length in bytes of a watched area. This property is only valid for watchpoints. (settable)

line

The source line at which the action point is set. This property is not valid for watchpoints.

satisfaction_group

The group that must arrive at a barrier for the barrier to be satisfied. (settable)

share

A value (either 1 or 0) indicating if the action point is active in the entire share group. A value of 1 means that it is. (settable)

stop_when_done

A value that indicates what is stopped when a barrier is satisfied (in addition to the satisfaction set). Values are process, group, or none. (settable)
**stop_when_hit**

A value that indicates what is stopped when an action point is hit (in addition to the thread that hit the action point). Values are `process`, `group`, or `none`. (settable)

**type**

The object’s type. (See `type_values` for a list of possible types.)

**type_values**

Lists values that can TotalView can assign to the `type` property: `break`, `eval`, `process_barrier`, `thread_barrier`, and `watch`.

**Examples**

```python
TV::actionpoint set 5 share 1 enable 1
```

Shares and enables action point 5.

```python
f p3 TV::actionpoint set -all enable 0
```

Disables all the action points in process 3.

```python
foreach p [TV::actionpoint properties] {
  puts [format "%-20s %s" $p:  
    [TV::actionpoint get 1 $p]]
}
```

Dumps all the properties for action point 1. Here is what your output might look like:

```
address: 0x1200019a8
enabled: 0
expression: id: 1
language:
length:
line: /temp/arrays.F#84
satisfaction_group:
satisfaction_process:
satisfaction_width:
share: 1
stop_when_done:
stop_when_hit: group
type:
type_values: break eval process_barrier thread_barrier watch
```
Dec2hex

Converts a decimal number into hexadecimal

Format
TV::dec2hex number

Arguments
number
A decimal number to convert.

Description
The TV::dec2hex command converts a decimal number into hexadecimal. This command correctly manipulates 64-bit values, regardless of the size of a long value on the host system.
**CLI Namespace Commands**

**dll**

Manages shared libraries

**Format**

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

**Arguments**

*action*

The action to perform, as follows:

*close*

Dynamically unloads the shared object libraries that were dynamically loaded by the `ddlopen` commands corresponding to the list of `dll-ids`.

If you use the `-all` option, TotalView closes all of the libraries that it opened.

*commands*

Displays the subcommands that you can use. The CLI responds by displaying these four `action` subcommands. There are no arguments to this subcommand.

*get*

Retrieves the values of one or more `TV::dll` properties. The `other-args` argument can include one or more property names.

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

*properties*

Lists the `TV::dll` properties that TotalView can access. There are no arguments to this subcommand.

*resolution_urgency_values*

Returns a list of values that this property can take. This list is operating-system specific, but always includes `{lazy now}`.

*symbol_availability_values*

Returns a list of values that this property can take. This list is operating system specific, but always includes `{lazy now}`.

*dll-id-list*

A list of one or more dll-ids. There are the IDs returned by the `ddlopen` command.

-`all`

Closes all shared libraries that you opened using the `ddlopen` command.

**Description**

The `TV::dll` command either closes shared libraries that were dynamically loaded with the `ddlopen` command or obtains information about loaded shared libraries.
Examples

TV::dll close 1

Closes the first shared library that you opened.
errorCodes

Returns or raises TotalView error information

Format

Returns a list of all error code tags

TV::errorCodes

Returns or raises error information

TV::errorCodes number_or_tag [-raise [message]]

Arguments

number_or_tag
An error code mnemonic tag or its numeric value.

-raise

Raises the corresponding error. If you append a message, TotalView returns this string. Otherwise, TotalView uses the human-readable string for the error.

message
An optional string used when raising an error.

Description

The TV::errorCodes command lets you manipulate the TotalView error code information placed in the Tcl errorCodes variable. The CLI sets this variable after every command error. Its value is intended to be easy to parse in a Tcl script.

When the CLI or TotalView returns an error, errorCodes is set to a list with the following format:

TOTALVIEW error-code subcodes... string

where:

- The first list element is always TOTALVIEW.
- The second list element is always the error code.
- The subcodes argument is not used at this time.
- The last list element is a string describing the error.

With a tag or number, this command returns a list containing the mnemonic tag, the numeric value of the tag, and the string associated with the error.

The -raise option raises an error. If you add a message, that message is used as the return value; otherwise, the CLI uses its textual explanation for the error code. This provides an easy way to return errors from a script.
Examples

```
foreach e [TV::errorCodes] {
    puts [eval format {"%20s %2d %s"} \ 
         [TV::errorCodes $e]]
}
```

Displays a list of all TotalView error codes.
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.

suspend-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` is 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.<>
```
focus_groups

Returns a list of groups in the current focus

Format
TV::focus_groups

Arguments
This command has no arguments

Description
The TV::focus_groups command returns a list of all groups in the current focus.

Examples
f d1.< TV::focus_groups
Returns a list containing one entry, which is the ID of the control group for process 1.

focus_processes Command
focus_threads Command
dfocus Command
Using Groups, Processes, and Threads
focus_processes

Returns a list of processes in the current focus

Format
TV::focus_processes [ -all | -group | -process | -thread ]

Arguments
- all
  Changes the default width to all.
- group
  Changes the default width to group.
- process
  Changes the default width to process.
- thread
  Changes the default width to thread.

Description
The TV::focus_processes command returns a list of all processes in the current focus. If the focus width is something other than d (default), the focus width determines the set of processes returned. If the focus width is d, the TV::focus_processes command returns process width. Using any of the options changes the default width.

Examples
f g1.< TV::focus_processes
Returns a list containing all processes in the same control as process 1.

focus_groups Command
focus_threads Command
dfocus Command
Using Groups, Processes, and Threads
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 pl.< TV::focus_threads
  Returns a list containing all threads in process 1.
group

Sets and gets group properties

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

Arguments
action
The action to perform, as follows:

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

get
Gets the values of one or more group properties. The other-args argument can include one or more property names. The CLI returns the values for these properties in a list in the same order as you entered the property names.

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

properties
Displays the properties that the CLI can access. Do not use additional arguments with this option.

set
Sets the values of one or more properties. The other-args argument is a sequence of property name and value pairs.

object-id
The group ID. If you use the -all option, TotalView executes this operation on all groups in the current focus.

other-args
Arguments required by the get and set subcommands.

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

actionpoint_count
The number of shared action points planted in the group. This is only valid for share groups and shared action points that are associated with the share group containing the process, rather than with the process itself.

When you obtain the results of this read-only value, the number may not look correct as this number also includes “magic breakpoints”. These are breakpoints that TotalView sets behind the scene; they are not usually visible. In addition, these magic breakpoints seldom appear when you use the dactions command.

canonical_execution_name
The absolute file name of the program being debugged. If you had entered a relative name, TotalView finds this absolute name.
count
   The number of members in a group.

executable
   Like canonical_execution_name, this is the absolute file name of the program being debugged. It differs in that it contains symbolic links and the like that exist for the program.

id
   The ID of the object.

member_type
   The type of the group's members, either process or thread.

member_type_values
   A list of all possible values for the member_type -property. For all groups, this is a two-item list with the first being the number of proess groups and the second being the number of thread groups. In many ways, this is related to the type_values property, which is a list values the type property may take.

members
   A list of a group's processes or threads.

type
   The group's type. Possible values are control, lockstep, share, user, and workers.

type_values
   A list of all possible values for the type property.

Examples

TV::group get 1 count

Returns the number of objects in group 1.
hex2dec

Converts a hexadecimal number to decimal

Format

TV::hex2dec number

Arguments

number

A hexadecimal number to convert.

Description

The TV::hex2dec command converts a hexadecimal number to decimal. You can type 0x before this value. The CLI correctly manipulates 64-bit values, regardless of the size of a long value.

dec2hex
process

Sets and gets process properties

Format

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

Arguments

The action to perform, as follows:

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

get
Gets the values of one or more process properties. The other-args argument can include one or more property names. The CLI returns these property values in a list whose order is the same as the names you enter. If you use the -all option instead of object-id, the CLI returns a list containing one (sublist) element for each object.

properties
Displays the properties that the CLI can access. Do not use other arguments with this subcommand.

set
Sets the values of one or more properties. The other-args arguments contains pairs of property names and values.

object-id
An identifier for a process. For example, 1 represents process 1. If you use the -all option, the operation executes upon all objects of this class in the current focus.

other-args
Arguments required by the get and set subcommands.

Description

The TV::process command lets you examine and set process properties and states, as the following list describes:

cannonical_executable_name
The full pathname of the current executable.

clusterid
The ID of the cluster containing the process. This is a number uniquely identifying the TotalView server that owns the process. The ID for the cluster TotalView is running in is always 0 (zero).

data_size
The size of the process's data segment.
duid
   The internal unique ID associated with an object.

executable
   Like canonical_execution_name, this is the absolute file name of the program being debugged. It differs in that it contains an symbolic links and the like that exist for the program.

heap_size
   The amount of memory currently being used for data created at runtime. Stated in a different way, the heap is an area of memory that your program uses when it needs to dynamically allocate memory. For example, calls to the malloc() function allocate space on the heap while the free() function releases the space.

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

hia_guard_max_size
   The value set for the maximum size for guard blocks that surround a memory allocation. See the “Debugging Memory Problems Using TotalView Guide” for information on what this size represents.

hia_guard_payload_alignment
   The number of bits the guard block is aligned to.

hia_guard_pre_pattern
   The numerical value of the bit pattern written into the guard block preceding an allocated memory block.

hia_guard_pre_size
   The number of bits into which the guard block preceding an allocated memory block is written.

hia_guard_post_pattern
   The numerical value of the bit pattern written into the guard block following an allocated memory block.

hia_guard_post_size
   The number of bits into which the guard block following an allocated memory block is written.

hia_paint_pattern_width
   Deprecated

hostname
   A name of the process's host computer and operating system (if needed); for example, linux-x86 would be returned if the program is running on a 32-bit linux system.

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

id
   The process ID.

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

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

parallel_attach_subset
   Contains the specification for MPI ranks to be attached to when an MPI job is created or attached to. See `-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**

```
fg 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
```

**Using Groups, Processes, and Threads**

**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 Users Guide.
respond

Provides responses to commands

Format

TV::respond response command

Arguments

response

The response to one or more commands. If you include more than one response, separate the responses with newline characters.

command

One or more commands that the CLI executes.

Description

The TV::respond command executes a command. The command argument can be a single command or a list of commands. In most cases, you place this information in braces ({}). If the CLI asks questions while command is executing, you are not asked for the answer. Instead, the CLI uses the characters in the response string for the argument. If more than one question is asked and strings within the response argument have all been used, the TV::respond command starts over at the beginning of the response string. If response does not end with a newline, the TV::respond command appends one.

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

dset LINES_PER_SCREEN 0

The most common values for response are y and n.

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.

**code_unit_by_soid**

Look up loader symbols by using a regular expression to match the base name. For example:

```
TV::scope lookup $scope_id \
loader_sym_by_regexp \
"regular expression"
```

**commands**

Displays the subcommands that you can use. The CLI responds by displaying the subcommands shown here. Do not use additional arguments with this subcommand.

**create**

Allows you to create blocks, enum_type, named_constant, typedef, upc_shared_type, and variable symbols. The type of symbol determines the properties you 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**

Dump all properties of all symbols in the scope and in the enclosed scope.

**get**

Returns properties of the symbols whose soids are specified. Specify the kinds of properties using the `other-args` argument.

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

**lookup**

Look up a symbol by name. Specify the kind of lookup using the `other-args` argument. The values you can enter are:

- **by_language_rules**: Use the language rules of the language of the scope to find a single name.
- **by_path**: Look up a symbol using a pathname.
- **by_properties [proptery_regexp_pair]**: TotalView recurses down the scope tree after it visits a symbol. This means TotalView will search for matching symbols in the specified scope and any nested scope. The `walk` property shows an example.
- **by_type_index**: Look up a symbol using a type index.
in_scope: Look up a name in the given scope and in all enclosing scopes, and in the global scope.

lookup_keys
Displays the kinds of lookup operations that you can perform.

properties
Displays the properties that the CLI can access. Do not use additional arguments with this option. The arguments displayed are those that are displayed for the scope of all types. Additional properties also exist but are not shown. (Only the ones used by all are visible.) For more information, see TV::symbol.

walk
Walk the scope, calling Tcl commands at particular points in the walk. The commands are named using the following options:

by_properties [proptery_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. 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] \n [required_property_regex_pair]... \n [non-required_property_regex_pair]... This is the general specification for creating a symbol

TV::scope create 1|31 kind block location {ldam 0x12}

Create a block. A block should have a length. However, you can set the length later using the set property.
source_process_startup  
Reads, then executes a .tvd file when a process is loaded

Format
TV::source_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.

Initializing TotalView
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-consts 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_kindartificial external name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>full_pathnameidkindscope_ownerscope_owner</td>
</tr>
<tr>
<td>array_type</td>
<td>X</td>
<td>X</td>
<td>artificialdata_addressingelement_addressingexternal_name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>index_type_indexkindlogical_scope_ownerslower_boundscope_ownerstride_bound</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>submembertarget_type_indexupper_boundvalidator</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>block</td>
<td>X</td>
<td></td>
<td>address_class id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>artificial kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>full_pathname length</td>
</tr>
<tr>
<td>char_type</td>
<td>X</td>
<td>X</td>
<td>artificial id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>external_name kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>full_pathname location</td>
</tr>
<tr>
<td>code_type</td>
<td>X</td>
<td>X</td>
<td>artificial id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>external_name kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>full_pathname</td>
</tr>
<tr>
<td>ds undiscovered type</td>
<td>X</td>
<td>X</td>
<td>artificial id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>external_name kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>full_pathname</td>
</tr>
<tr>
<td>enum_type</td>
<td>X</td>
<td>X</td>
<td>artificial id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>external_name kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>enumerators full_pathname</td>
</tr>
<tr>
<td>file</td>
<td>X</td>
<td></td>
<td>artificial full_pathname idkind language</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>compiler_kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>delayed_symbol demangler</td>
</tr>
<tr>
<td>float_type</td>
<td>X</td>
<td>X</td>
<td>artificial full_pathname idkind length</td>
</tr>
<tr>
<td>function_type</td>
<td>X</td>
<td>X</td>
<td>artificial full_pathname idkind length</td>
</tr>
<tr>
<td>image</td>
<td>X</td>
<td></td>
<td>artificial full_pathname id</td>
</tr>
<tr>
<td>int_type</td>
<td>X</td>
<td>X</td>
<td>artificial full_pathname idkind length</td>
</tr>
<tr>
<td>label</td>
<td>X</td>
<td></td>
<td>address_class full_pathname idkind 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</td>
</tr>
<tr>
<td></td>
<td>artificial</td>
<td>kind</td>
<td>scope_owner</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</td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>length</td>
<td>scope_owner</td>
</tr>
<tr>
<td>member</td>
<td>address_class</td>
<td>inheritance</td>
<td>ordinal</td>
</tr>
<tr>
<td></td>
<td>artificial</td>
<td>kind</td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>location</td>
<td>type_index</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>module</td>
<td>artificial</td>
<td>id</td>
<td>logical_scope_owner</td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>kind</td>
<td>scope_owner</td>
</tr>
<tr>
<td>named_constant</td>
<td>artificial</td>
<td>kind</td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>length</td>
<td>type_index</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>namespace</td>
<td>artificial</td>
<td>id</td>
<td>logical_scope_owner</td>
</tr>
<tr>
<td></td>
<td>full_pathname</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>opaque_type</td>
<td>artificial</td>
<td>id</td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td>external_name</td>
<td>kind</td>
<td></td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>pathname_reference_symbol</td>
<td>artificial</td>
<td>kind</td>
<td>resolved_symbol_pathname</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>lookup_scope</td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>pointer_type</td>
<td>artificial</td>
<td>kind</td>
<td>target_type_index</td>
</tr>
<tr>
<td></td>
<td>external_name</td>
<td>length</td>
<td>validator</td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>scope_owner</td>
<td></td>
</tr>
<tr>
<td>qualified_type</td>
<td>artificial</td>
<td>id</td>
<td>qualification</td>
</tr>
<tr>
<td></td>
<td>external_name</td>
<td>kind</td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>target_type_index</td>
<td></td>
</tr>
<tr>
<td>soid_reference_symbol</td>
<td>artificial</td>
<td>kind</td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td></td>
<td>id</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>
<tbody>
<tr>
<td>stringchar_type</td>
<td>X</td>
<td>X</td>
<td>artificial, external_name, full_pathname, id, kind, logical_scope_owner</td>
</tr>
<tr>
<td>subroutine</td>
<td>X</td>
<td></td>
<td>address_class, artificial, full_pathname, kind, length, location, logical_scope_owner, return_type_index, scope_owner, static_chain, static_chain_height</td>
</tr>
<tr>
<td>typedef</td>
<td>X</td>
<td>X</td>
<td>artificial, external_name, full_pathname, id, kind, length, logical_scope_owner, scope_owner, target_type_index</td>
</tr>
<tr>
<td>variable</td>
<td>X</td>
<td></td>
<td>address_class, artificial, full_pathname, kind, location, logical_scope_owner, ordinal, scope_owner, type_index</td>
</tr>
<tr>
<td>void_type</td>
<td>X</td>
<td>X</td>
<td>artificial, external_name, full_pathname, id, kind, length, logical_scope_owner, scope_owner</td>
</tr>
<tr>
<td>wchar_type</td>
<td>X</td>
<td>X</td>
<td>artificial, external_name, full_pathname, id, kind, logical_scope_owner, scope_owner, target_type_index</td>
</tr>
</tbody>
</table>

Figure 1 on page 206 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 showing data addressing concept]

**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 211 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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>type_index</td>
</tr>
<tr>
<td>cplus_global_symname</td>
<td>base_name</td>
</tr>
<tr>
<td></td>
<td>cplus_template_types</td>
</tr>
<tr>
<td></td>
<td>cplus_class_name</td>
</tr>
<tr>
<td></td>
<td>cplus_type_name</td>
</tr>
<tr>
<td></td>
<td>cplus_local_name</td>
</tr>
<tr>
<td></td>
<td>loader_file_path</td>
</tr>
<tr>
<td></td>
<td>loader_name</td>
</tr>
<tr>
<td>cplus_local_symname</td>
<td>base_name</td>
</tr>
<tr>
<td></td>
<td>cplus_overload_list</td>
</tr>
<tr>
<td></td>
<td>cplus_class_name</td>
</tr>
<tr>
<td></td>
<td>cplus_template_types</td>
</tr>
<tr>
<td></td>
<td>cplus_local_name</td>
</tr>
<tr>
<td></td>
<td>cplus_type_name</td>
</tr>
<tr>
<td>cplus_type_symname</td>
<td>base_name</td>
</tr>
<tr>
<td></td>
<td>cplus_template_types</td>
</tr>
<tr>
<td></td>
<td>cplus_class_name</td>
</tr>
<tr>
<td></td>
<td>cplus_type_name</td>
</tr>
<tr>
<td></td>
<td>cplus_local_name</td>
</tr>
<tr>
<td></td>
<td>type_index</td>
</tr>
<tr>
<td></td>
<td>cplus_overload_list</td>
</tr>
<tr>
<td>file_symname</td>
<td>base_name</td>
</tr>
<tr>
<td></td>
<td>directory_path</td>
</tr>
<tr>
<td></td>
<td>directory_hint</td>
</tr>
<tr>
<td>fortran_global_symname</td>
<td>base_name</td>
</tr>
<tr>
<td></td>
<td>loader_file_path</td>
</tr>
<tr>
<td></td>
<td>fortran_module_name</td>
</tr>
<tr>
<td></td>
<td>loader_name</td>
</tr>
<tr>
<td></td>
<td>fortran_parent_function_name</td>
</tr>
<tr>
<td>fortran_local_symname</td>
<td>base_name</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td>fortran_module_name</td>
<td>type_index</td>
</tr>
<tr>
<td>image_symname</td>
<td>base_name</td>
</tr>
<tr>
<td></td>
<td>member_name</td>
</tr>
<tr>
<td></td>
<td>directory_path</td>
</tr>
<tr>
<td></td>
<td>node_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</td>
</tr>
<tr>
<td></td>
<td>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 manager thread. (1 means that it is a system manager thread.)

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

sp
The value of the stack pointer.

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

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

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

systid
The system thread ID.

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

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

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

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

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

```
proc show_signal {} {
    foreach th [TV::focus_threads] {
        puts "Continue_sig ($th): \ 
            [TV::thread get $th continue_sig]";
    }
}
```

Show all starting signals

**Using Groups, Processes, and Threads**

- **focus_threads Command**
- **group Command**
- **process Command**
**type**

**Gets and sets type properties**

**Format**

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

**Arguments**

*action*

The action to perform, as follows:

- **commands**
  
  Displays the subcommands that you can use. The CLI responds by displaying these four *action* 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:

```
print_type 1|6
```

```markdown
<table>
<thead>
<tr>
<th>enum_values</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>image_id</td>
</tr>
<tr>
<td>language</td>
</tr>
<tr>
<td>length</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>prototype</td>
</tr>
<tr>
<td>rank</td>
</tr>
<tr>
<td>struct_fields</td>
</tr>
<tr>
<td>target</td>
</tr>
<tr>
<td>type</td>
</tr>
<tr>
<td>type_values</td>
</tr>
</tbody>
</table>
```
**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 no longer change them.
    - **get**
      - Gets the values of one or more transformation properties. The *other-args* argument can include one or more property names. The CLI returns these property values in a list whose order is the same as the property names you entered.
      - If you use the **-all** option instead of *object-id*, the CLI returns a list containing one (sublist) element for the object.
    - **properties**
      - Displays the properties that the CLI can access. Do not use additional arguments with this option. These properties are discussed later in this section.
    - **set**
      - Sets the values of one or more properties. The *other-args* argument consists of pairs of property names and values. The argument pairs that you can set are listed later in this section.
    - **object-id**
      - The type transformation ID. This value is returned when you create a new transformation; for example, 1 represents process 1. If you use the **-all** option, the operation executes upon all objects of this class in the current focus.
    - **other-args**
      - Arguments required by get and set subcommands.

**Description**

The TV::type_transformation command lets you define and examine properties of a type transformation. The states and properties you can set are:
**Common Properties**

**id**

The type transformation ID returned from a `create` operation.

**language**

The language property specifies source language for the code of the aggregate type (class) to transform. This is always C++.

**name**

Contains a regular expression that checks to see if a symbol is eligible for type transformation. This regular expression must match the definition of the aggregate type (class) being transformed.

**type_callback**

The `type_callback` property is used in two ways.

1. When it is used within a list or vector transformation, it names the procedure that determines the type of the list or vector element. The callback procedure takes one parameter, the symbol ID of the symbol that was validated during the callback to the procedure specified by the `validate_callback`. The call structure for this callback is:

   ```
   type_callback id
   ```

   where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

2. When it is used within a struct transformation, it names the procedure that specifies the data type to be used when displaying the struct.

**type_transformation_description**

A string containing a description of what is being transformed; for example, you might enter “GNU Vector”.

**validate_callback**

Names a procedure that is called when a data type matches the regular expression specified in the `name` property. The call structure for this callback is:

```
validate_callback id
```

where `id` is the symbol ID of the symbol being validated.

Your callback procedure should check the symbol’s structure to insure that it should be transformed. While not required, most users will extract symbol information such as its type and its data members while validating the datatype. The callback procedure must return a Boolean value, where `true` means the symbol is valid and can be transformed.

**compiler**

Reserved for future use.

**Array Properties**

**addressing_callback**

Names the procedure that locates the address of the start of an array. The call structure for this callback is:

```
addressing_callback id
```
where $id$ is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

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

**lower_bounds_callback**
Names the procedure that obtains a lower bound value for the array type being transformed. For C/C++ arrays, this value is always 0. The call structure for this callback is:

`lower_bounds_callback id`

where $id$ is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

**upper_bounds_callback**
Names the procedure that defines an addressing expression that computes the extent (number of elements) in an array. The call structure for this callback is:

`upper_bounds_callback id`

where $id$ is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

**List Properties**

**list_element_count_addressing_callback**
Names the procedure that determines the total number of elements in a list. The call structure for this callback is:

`list_element_count_addressing_callback id`

where $id$ is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

This callback defines an addressing expression that specifies how to get to the member of the symbol that specifies the number of elements in the list.

If your data structure does not have this element, you still must use this callback. In this case, simply return `{nop}` as the addressing expression and the transformation will count the elements by following all the pointers. This can be very time consuming.

**list_element_data_addressing_callback**
Names the procedure that defines an addressing expression that specifies how to access the data member of a list element. The call structure for this callback is:

`list_element_data_addressing_callback id`

where $id$ is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

**list_element_next_addressing_callback**
Names the procedure that defines an addressing expression that specifies how to access the next element of a list. The call structure for this callback is:

`list_element_next_addressing_callback id`

where $id$ is the symbol ID of the symbol that was validated using the `validate_callback` procedure.
**list_element_prev_addressing_callback**
Names the procedure that defines an addressing expression that specifies how to access the previous element of a list. The call structure for this callback is:

`list_element_prev_addressing_callback id`

where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

This property is optional. For example, you would not use it in a singly linked list.

**list_end_value**
Specifies if a list is terminated by NULL or the head of the list. Enter one of the following: **NULL** or **ListHead**

**list_first_element_addressing_callback**
Names the procedure that defines an addressing expression that specifies how to go from the head element of the list to the first element of the list. It is not always the case that the head element of the list is the first element of the list. The call structure for this callback is:

`list_first_element_addressing_callback id`

where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

**list_head_addressing_callback**
Names the procedure that defines an addressing expression to obtain the head element of the linked list. The call structure for this callback is:

`list_head_addressing_callback id`

where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

**Struct Properties**

**struct_member_count_callback**
Names the procedure that obtains the total number of members in a struct. The call structure for this callback is:

`struct_member_count_callback id`

where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

**struct_member_addressing_callback**
Names the procedure that defines an addressing expression that specifies how to access the specified member of a struct. The call structure for this callback is:

`struct_member_addressing_callback id index`

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

**struct_member_type_callback**
Names the procedure that obtains the type id of the specified member of a struct. The call structure for this callback is:

`struct_member_type_callback id index`
where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure and \( index \) specifies the zero-based position of the member within the struct.

**struct_member_name_callback**

Names the procedure that obtains the name of the specified member of a struct. The call structure for this callback is:

\[
\text{struct_member_name_callback} \ id \ index
\]

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

**Red/Black Tree Properties**

The implementation of map/multimap and set/multiset STL types uses red/black trees. These properties are common to all these types.

**rbtree_head_addressing_callback**

Names the procedure that defines an addressing expression to obtain the head element of the map. The call structure for this callback is:

\[
\text{rbtree_head_addressing_callback} \ id
\]

where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.

**rbtree_head_type_callback**

Names the procedure that obtains the type id of the head of a map. The call structure for this callback is:

\[
\text{rbtree_head_type_callback} \ id
\]

where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.

**rbtree_element_left_addressing_callback**

Names the procedure that defines an addressing expression that specifies how to access the left sub-tree of the current element of a map. The call structure for this callback is:

\[
\text{rbtree_element_left_addressing_callback} \ id
\]

where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.

**rbtree_element_right_addressing_callback**

Names the procedure that defines an addressing expression that specifies how to access the right sub-tree of the current element of a map. The call structure for this callback is:

\[
\text{rbtree_element_right_addressing_callback} \ id
\]

where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.

**rbtree_element_parent_addressing_callback**

Names the procedure that defines an addressing expression that specifies how to access the parent of the current element of a map. The call structure for this callback is:

\[
\text{rbtree_element_parent_addressing_callback} \ id
\]

where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.
rbtree_element_count_addressing_callback
Names the procedure that determines the total number of elements in a map. The call structure for this callback is:

```
rbtree_element_count_addressing_callback id
```

where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

This callback defines an addressing expression that specifies how to get to the member of the symbol that specifies the number of elements in the map.

If your data structure does not have this element, you still must use this callback. In this case, simply return `{nop}` as the addressing expression and the transformation will count the elements by following all the pointers. Unfortunately, this can be very time consuming.

rbtree_element_count_type_callback
Names the procedure that obtains the type ID of the member that specifies the number of elements in the map. The call structure for this callback is:

```
rbtree_element_count_type_callback id
```

where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

If your data structure does not have a count element, this property is not required.

rbtree_left_most_addressing_callback
Names the procedure that defines an addressing expression to obtain the left-most element of the map. The call structure for this callback is:

```
rbtree_left_most_addressing_callback id
```

where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

Map/Multimap Properties
For map and multimap STL types these properties are used in combination with those for red/black trees above.

map_element_key_data_addressing_callback
Names the procedure that defines an addressing expression that specifies how to access the key of an element of a map. The call structure for this callback is:

```
map_element_key_data_addressing_callback id
```

where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

map_element_key_type_callback
Names the procedure that obtains the type ID of the key of a map. The call structure for this callback is:

```
map_element_key_type_callback id
```

where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.

map_element_type_callback
Names the procedure that obtains the type ID of the element in the red/black tree that contains the key/value pair. The call structure for this callback is:
map_element_type_callback id
where id is the symbol ID of the symbol that was validated using the validate_callback procedure.

map_element_value_data_addressing_callback
Names the procedure that defines an addressing expression that specifies how to access the value of an element of a map. The call structure for this callback is:

map_element_value_data_addressing_callback id
where id is the symbol ID of the symbol that was validated using the validate_callback procedure.

map_element_value_type_callback
Names the procedure that obtains the type id of the value of a map. The call structure for this callback is:

map_element_value_type_callback id
where id is the symbol ID of the symbol that was validated using the validate_callback procedure.

map_iterator_end_value
Specifies if a map is terminated by NULL or the head of the map. Enter one of the following: NULL or MapHead

Set/Multiset Properties
For set and multiset STL types these properties are used in combination with those for red/black trees above.

set_element_data_addressing_callback
Names the procedure that defines an addressing expression that specifies how to access an element of a set. The call structure for this callback is:

set_element_data_addressing_callback id
where id is the symbol ID of the symbol that was validated using the validate_callback procedure.

set_element_type_callback
Names the procedure that obtains the type id of an element in the set. The call structure for this callback is:

set_element_type_callback id
where id is the symbol ID of the symbol that was validated using the validate_callback procedure.

set_iterator_end_value
Specifies if a set is terminated by NULL or the head of the set. Enter one of the following: NULL or SetHead

Hashtable Properties
The implementations of unordered map/multimap and unordered set/multiset STL types use hash tables. These properties are common to all these types.

hashtable_head_addressing_callback
Names the procedure that defines an addressing expression to obtain the head element of the map. Depending on the implementation, this element may be the address of the bucket list or the beginning element of a forward list. The call structure for this callback is:

hashtable_head_addressing_callback id
where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.

hashtable_element_count_addressing_callback

Names the procedure that determines the total number of elements in a hashtable. The call structure for this callback is:

\[
\text{hashtable_element_count_addressing_callback}(id)
\]

where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.

This callback defines an addressing expression that specifies how to get to the member of the symbol that specifies the number of elements in the map.

hashtable_element_count_type_callback

Names the procedure that obtains the type id of the member that specifies the number of elements in the map. The call structure for this callback is:

\[
\text{hashtable_element_count_type_callback}(id)
\]

where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.

hashtable_element_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the next element. The call structure for this callback is:

\[
\text{hashtable_element_addressing_callback}(id)
\]

where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.

hashtable_begin_index_addressing_callback

Names the procedure that determines the index of the first used bucket in a hashtable. The call structure for this callback is:

\[
\text{hashtable_begin_index_addressing_callback}(id)
\]

where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.

This callback defines an addressing expression that specifies how to get to the member of the symbol that specifies the first used bucket in the hashtable. This allows a small optimization since the transformation can skip empty buckets at the start of the bucket table. If your data does not supply this value you can use \{nop\}.

hashtable_begin_index_type_callback

Names the procedure that determines the type of the value that contains the index of the first used bucket in a hashtable. The call structure for this callback is:

\[
\text{hashtable_begin_index_type_callback}(id)
\]

where \( id \) is the symbol ID of the symbol that was validated using the validate_callback procedure.

hashtable_bucket_count_addressing_callback

Names the procedure that determines the total number of buckets in a hash table. The call structure for this callback is:

\[
\text{hashtable_bucket_count_addressing_callback}(id)
\]
where \( id \) 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 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.

\textbf{hashtable\textunderscore bucket\textunderscore count\textunderscore type\textunderscore callback}

Names the procedure that obtains the type id of the member that specifies the number of buckets in a hashtable. The call structure for this callback is:

\texttt{hashtable\textunderscore bucket\textunderscore count\textunderscore type\textunderscore callback \( id \)}

where \( id \) is the symbol ID of the symbol that was validated using the \texttt{validate\textunderscore callback} procedure.

If you are not scanning the bucket list for the hashed values, this property is not required.

\textbf{Unordered Map/Multimap Properties}

For unordered map and unordered multimap STL types these properties are used in combination with those for hash tables above.

\textbf{umap\textunderscore element\textunderscore key\textunderscore data\textunderscore addressing\textunderscore 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:

\texttt{umap\textunderscore element\textunderscore key\textunderscore data\textunderscore addressing\textunderscore callback \( id \)}

where \( id \) is the symbol ID of the symbol that was validated using the \texttt{validate\textunderscore callback} procedure.

\textbf{umap\textunderscore element\textunderscore key\textunderscore type\textunderscore callback}

Names the procedure that obtains the type id of the key of a map. The call structure for this callback is:

\texttt{umap\textunderscore element\textunderscore key\textunderscore type\textunderscore callback \( id \)}

where \( id \) is the symbol ID of the symbol that was validated using the \texttt{validate\textunderscore callback} procedure.

\textbf{umap\textunderscore element\textunderscore type\textunderscore 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:

\texttt{umap\textunderscore element\textunderscore type\textunderscore callback \( id \)}

where \( id \) is the symbol ID of the symbol that was validated using the \texttt{validate\textunderscore callback} procedure.

\textbf{umap\textunderscore element\textunderscore value\textunderscore data\textunderscore addressing\textunderscore 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:

\texttt{umap\textunderscore element\textunderscore value\textunderscore data\textunderscore addressing\textunderscore callback \( id \)}

where \( id \) is the symbol ID of the symbol that was validated using the \texttt{validate\textunderscore 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:

```plaintext
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:

```plaintext
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:

```plaintext
uset_element_key_type_callback id
```

where `id` is the symbol ID of the symbol that was validated using the `validate_callback` procedure.
Chapter 4

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:

```bash
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:

```bash
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 334. tvscript command-line options are described in the next section.

**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 334.

**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:

```
tvscript [ options ] -mpi BlueGene -np number-of-processes -starter_args "filename [ mpi-arguments ] [-args program_args]" mpirun
```

- **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.
**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 "[srun-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.

**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

```
```

- **-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:

\[ \text{tvscript } [ \text{options } ] -\text{mpi BlueGeneQ-LoadLeveler} -\text{np number-of-processes} \quad \text{-starter_args "[runjob-arguments]"
--exe filename } [\text{program_args}]\] 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:

\[ \text{tvscript } [ \text{options } ] -\text{mpi CrayKNC-aprun} -\text{np number-of-processes} \quad \text{-starter_args "[aprun-arguments] filename
[program_args]" aprun
}

- `np`
  The number of processes or tasks that the starter program will create.

- `starter_args`
  Required, with the arguments following enclosed in quotes; the application executable (`filename`) to be debugged must follow the arguments for `aprun`.

`aprun-arguments`
  The command arguments for `aprun` (except the `-k` argument).

`filename`
  The program being debugged.

`program_args`
  The arguments for the program being debugged.
aprun
   Required; the executable at the end of the command line.

For example:

    tvscript \
    -create_actionpoint "tx_basic_mpi.c#98=>display_backtrace \
        -show_arguments, print myid" \
    -mpi CrayKNC-aprun -np 16 \
    -starter_args "tx_basic_mpi" \
    aprun

Cray XK7

The syntax for using tvscript on Cray XK7 is:

    tvscript [ options ] -mpi CrayXK7-aprun -np number-of-processes -starter_args \
        "[aprun-arguments] filename [program_args]" aprun

    -np
    The number of processes or tasks that the starter program will create.

    -starter_args
    Required, with the arguments following enclosed in quotes; the application executable (filename) to be debugged must follow the arguments for aprun.

    aprun-arguments
    The command arguments for aprun.

    filename
    The program being debugged.

    program_args
    The arguments for the program being debugged.

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

For example:

    tvscript \
    -create_actionpoint "tx_basic_mpi.c#98=>display_backtrace \
        -show_arguments, print myid" \
    -mpi CrayXK7-aprun -np 16 \
    -starter_args "tx_basic_mpi" \
    aprun
**tvscript Options**

- **-create_actionpoint** "source_location_expr [ =>action1, action2]"
  
  Creates an action point at a source location using an expression. (See “Action Point API” on page 241 for writing expressions.) When the action point is hit, tvscript can trigger one or more actions. Add one -create_watchpoint command-line option for each action point.

  See -event_action for information about actions.

- **-event_action** "event_action_list"
  
  Performs an action when an event occurs. Events represent an unanticipated condition, such as free_not_allocated in the Memory Debugger. You can use more than one -event_action command-line option when invoking tvscript.

  Here is how you enter an event_action_list:

  event1=action1,event2=action2

  or

  event1=>action1,action2,action3

**Table 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>any_event</td>
<td>A generated event occurred.</td>
</tr>
<tr>
<td>Memory debugging event</td>
<td>addr_not_at_start</td>
<td>Program attempted to free a block using an incorrect address.</td>
</tr>
<tr>
<td></td>
<td>alloc_not_in_heap</td>
<td>The memory allocator returned a block not in the heap; the heap may be corrupt.</td>
</tr>
<tr>
<td></td>
<td>alloc_null</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>alloc_returned_bad_alignment</td>
<td>The memory allocator returned a misaligned block; the heap may be corrupt.</td>
</tr>
<tr>
<td></td>
<td>any_memory_event</td>
<td>A memory event occurred.</td>
</tr>
<tr>
<td></td>
<td>bad_alignment_argument</td>
<td>Program supplied an invalid alignment argument to the heap manager.</td>
</tr>
<tr>
<td></td>
<td>double_alloc</td>
<td>The memory allocator returned a block currently being used; the heap may be corrupt.</td>
</tr>
<tr>
<td></td>
<td>double_dealloc</td>
<td>Program attempted to free an already freed block.</td>
</tr>
<tr>
<td></td>
<td>free_not_allocated</td>
<td>Program attempted to free an address that is not in the heap.</td>
</tr>
<tr>
<td></td>
<td>guard_corruption</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></td>
<td>hoard_low_memory_threshold</td>
<td>Hoard low memory threshold crossed.</td>
</tr>
<tr>
<td></td>
<td>realloc_not_allocated</td>
<td>Program attempted to reallocate an address that is not in the heap.</td>
</tr>
<tr>
<td></td>
<td>rz_overrun</td>
<td>Program attempted to access memory beyond the end of an allocated block.</td>
</tr>
<tr>
<td></td>
<td>rz_underrun</td>
<td>Program attempted to access memory before the start of an allocated block.</td>
</tr>
<tr>
<td></td>
<td>rz_use_after_free</td>
<td>Program attempted to access a block of memory after it has been deallocated.</td>
</tr>
<tr>
<td></td>
<td>rz_use_after_free_overrun</td>
<td>Program attempted to access memory beyond the end of a deallocated block.</td>
</tr>
<tr>
<td></td>
<td>rz_use_after_free_underrun</td>
<td>Program attempted to access memory before the start of a deallocated block.</td>
</tr>
<tr>
<td></td>
<td>termination_notification</td>
<td>The target is terminating.</td>
</tr>
<tr>
<td>Source code debugging event</td>
<td>actionpoint</td>
<td>A thread hit an action point.</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>An error occurred.</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><strong>Action Type</strong></td>
<td><strong>Action</strong></td>
<td><strong>Definition</strong></td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Source code debugging actions</td>
<td><code>display_backtrace</code></td>
<td>Writes the current stack backtrace into the log file.</td>
</tr>
<tr>
<td></td>
<td><code>-level level-num</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>[ num_levels ]</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>[ options ]</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>-level level-num</strong> sets the level at which information starts being logged.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>num_levels</strong> restricts output to this number of levels in the call stack.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If you do not set a level, <code>tvscript</code> displays all levels in the call stack.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>options</strong> is one or more of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-[no]show_arguments</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-[no]show_fp</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-[no]show_fp_registers</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-[no]show_image</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-[no]show_locals</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-[no]show_pc</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-[no]show_registers</code></td>
</tr>
<tr>
<td></td>
<td><code>print [-slice {slice_exp}]</code></td>
<td>Writes the value of a variable or an expression into the log file. If the</td>
</tr>
<tr>
<td></td>
<td>`{variable</td>
<td>exp}`</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by slice_exp. A slice expression is a way to define the slice, such as <code>var[100:130]</code> in C and C++. (This displays all values from <code>var[100]</code> to <code>var[130].</code>) To display every fourth value, add an additional argument; for example, <code>var[100:130:4]</code>. For additional information, see Chapter 14 “Examining Arrays” in the <code>TotalView User Guide</code>.</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 specifiers</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>
<tr>
<td>Memory debugging display specifiers</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>

**-memory_debugging**

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

**-mem_detect_leaks**

Performs leak detection before generating memory information.

**-mem_detect_use_after_free**

Tests for use after memory is freed.

**-mem_guard_blocks**

Adds guard blocks to an allocated memory block.
-**mem_hoard_freed_memory**
  Holds onto freed memory rather than returning it to the heap.

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

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

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

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

-**mem_red_zones_overruns**
  Turns on testing for Red Zone overruns.

-**mem_red_zones_size_ranges min:max,min:max,...**
  Defines the memory allocations ranges for which Red Zones are in effect. Ranges can be specified as follows:
  x:y allocations from x to y
  :y allocations from 1 to y
  x: allocations of x and higher
  x allocation of x

-**mem_red_zones_underruns**
  Turns on testing for Red Zone underruns.

-**maxruntime "hh:mm:ss"**
  Specifies how long the script can run.

-**script_file script_file**
  Names a file containing tvscript API calls and Tcl callback procedures that you create.

-**script_log_filename logFilename**
  Overrides the name of the TVScript log file.
  WARNING: Previous log files of the same name are overwritten.

-**script_summary_log_filename summaryLogFilename**
  Overrides the name of the TVScript summary log file.
  WARNING: Previous summary log files with the same name are overwritten.

**tvscript Example:**

The following example is similar to that shown in “Batch Debugging Using tvscript” on page 230.

```bash
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 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 192 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 214 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
**Batch Debugging Using tvscript**

**tvscript External Script Files**

- **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 information is passed to tvscript as follows:

```
tvscript -script_file script_file
```

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. Similarly, when an action point is hit, it passes an array of information to the action point handler. These arrays are described in “Event API” on page 242.

```
# 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:
  #   process_id
  #   thread_id
  #   error_message
  array set error_data_array $error_data

  # Get the process so we have some information about it
  temp = [ tvscript_get_process_property \n```

```
$error_data_array(process_id) "syspid"

tvscript_log "Process ID: $temp"

temp = [tvscript_get_thread_property \\
    $error_data_array(thread_id) "systid"

tvscript_log "Thread ID: $temp"

temp = $error_data_array(error_message)

tvscript_log "Error Message: $temp"
}

# Action point handlers
proc l1_actionpoint_handler {event_data} {
    tvscript_log "Inside l1_actionpnt_handler: $event_data"
    tvscript_slog "Inside l1_actionpnt_handler: $event_data"

    array set event_data_array $event_data

    temp = [tvscript_get_process_property \\
        $event_data_array(process_id) "syspid"
    tvscript_log "Process ID: $temp"

    temp = [tvscript_get_thread_property \\
        $event_data_array(thread_id) "systid"
    tvscript_log "Thread ID: $temp"

    temp = [tvscript_get_process_property \\
        $event_data_array(process_id) "state"
    tvscript_log "Status: $temp"

    temp = [tvscript_get_process_property \\
        $event_data_array(process_id) "executable"
    tvscript_log "Executable: $temp"

    temp = $event_data_array(actionpoint_source_loc_expr)
    tvscript_log "Action point Expression: $temp"
tvscript_log "Value of i:
set output [capture "dprint i"]
tvscript_log $output
}

# Event handlers

proc generic_actionpoint_event_handler {actionpoint_data} {
  tvscript_log "Inside generic_actionpoint_event_handler: 
  tvscript_log $actionpoint_data
  tvscript_slog "Inside generic_actionpoint_event_handler: 
  tvscript_slog $actionpoint_data

  # Change the incoming list into an array.
  # It contains the following indices:
  #   actionpoint_id
  #   actionpoint_source_loc_expr
  #   event
  #   process_id
  #   thread_id
  array set actionpnt_data_array $actionpoint_data

  temp = $actionpnt_data_array(process_id)
tvscript_log "            Process ID: $temp"

  temp = $actionpnt_data_array(thread_id)
tvscript_log "            Thread ID: $temp"

  temp = $actionpnt_data_array(actionpoint_id)
tvscript_log "       Action Point ID: $temp"

  temp = $actionpnt_data_array(actionpoint_source_loc_expr)
tvscript_log "Action Point Expression: 
}

# Add event handlers

# Setup action points and action point handlers
set actionpoint_id [tvscript_create_actionpoint "l1"]
tvscript_add_actionpoint_handler $actionpoint_id "l1_actionpoint_handler"

# Setup a generic actionpoint handler
tvscript_add_event_handler "actionpoint" "generic_actionpoint_event_handler"
# Add error handler
tvscript_add_event_handler "error" "error_handler"
Chapter 5

TotalView Variables

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(dpid)**
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(dpid)` variable is defined.

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

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

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

    dset CGROUP(3) $CGROUP(1)

Permitted Values: A number

Default: None

COMMAND_EDITING

Enables some Emacs-like commands for use when editing text in the CLI. These editing commands are always available in the CLI window of the TotalView GUI. However, they are available only in the stand-alone CLI if the terminal in which it is running supports cursor positioning and clear-to-end-of-line. The commands you can use are:

^A: Moves the cursor to the beginning of the line

^B: Moves the cursor one character backward

^D: Deletes the character to the right of cursor

^E: Moves the cursor to the end of the line

^F: Moves the cursor one character forward

^K: Deletes all text to the end of line

^N: Retrieves the next entered command (only works after ^P)

^P: Retrieves the previously entered command

^R or ^L: Redraws the line

^U: Deletes all text from the cursor to the beginning of the line

Rubout or Backspace: Deletes the character to the left of the cursor

Permitted Values: true or false

Default: false
EXECUTABLE_PATH
Contains a colon-separated list of the directories searched for source and executable files.

Permitted Values: Any directory or directory path. To include the current setting, use `$EXECUTABLE_PATH`.
Default: . (dot)

EXECUTABLE_SEARCH_MAPPINGS
Contains pairs of regular expressions and replacement and replacement strings—these replacements are called mappings—separated by colons. TotalView applies these mappings to the search paths before it looks for source, object, and program files.

The syntax for mapping strings is:

```
+regular_exp+=+replacement+ :+regular_exp+=+replacement+
```

This example shows two pairs, each delimited by a colon (";"). Each element within a pair is delimited by any character except a colon. The first character entered is the delimiter. This example uses a "+" as a delimiter. (Traditionally, forward slashes are used as delimiters but are not used here, as a forward slash is also used to separate components of a pathname. For example, `/home/my_dir` contains forward slashes.)

Be aware that special characters must follow standard Tcl rules and conventions, for example:

```
dset EXECUTABLE_SEARCH_MAPPINGS {+^/nfs/compiled/u2/(.*)$+ = +/nfs/host/u2/\1+ }
```

This expression applies a mapping so that a directory named `/nfs/compiled/u2/project/src1` in the expanded search path becomes `/nfs/host/u2/project/src1`.

Default: `{}`

EXECUTABLE_SEARCH_PATH
Contains a list of paths, separated by a colon, to search for executables. For information, see “Setting Search Paths Using TotalView Variables” in the online help.

Permitted Values: Any directory or directory path.
Default: `${EXECUTABLE_PATH};${PATH}:.`

GROUP(gid)
Contains a list of the TotalView IDs for all members in group `gid`.

The first element indicates the type of group:

control
The group of all processes in a program

lockstep
A group of threads that share the same PC

process
A user-created process group
share
The group of processes in one program that share the same executable image

thread
A user-created thread group

workers
The group of worker threads in a program

Elements that follow are either 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 online help.

- **Permitted Values:** Any directory or directory path.
- **Default:** ${COMPILATION_DIRECTORY}: ${EXECUTABLE_PATH}: ${EXECUTABLE_DIRECTORY}: $links${EXECUTABLE_DIRECTORY}: ..:${TOTALVIEW_SRC}

**PROCESS(dpid)**
Contains a list of information associated with a dpid. This is a read-only value and cannot be set.

- **Permitted Values:** An integer
- **Default:** None

**PROMPT**
Defines the CLI prompt. 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
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 online help.

Permitted Values: Any directory or directory path.
Default: ${COMPILEDATION_DIRECTORY}: ${EXECUTABLE_PATH}: ${EXECUTABLE_DIRECTORY}: ${links${EXECUTABLE_DIRECTORY}): ${TOTALVIEW_SRC}

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.v3breakpoints 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 **Action Point > Load All** or 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.v3breakpoints, where *filename* is the name of the file being debugged. Information about watchpoints is not saved.

TotalView writes this information when you exit from TotalView. If you set this variable to false, you can explicitly save this information by using the **Action Point > Save All** or the **dactions -save** command.

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

Contains the value 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

**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 tvdsrv 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 (tvdsrv) 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 (tvdsrv) 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 (tvdsrv) using the bulk server feature.

Permitted Values: A number from 1 to 3600 (1 hour)
Default: 10

TV::bulk\_launch\_string

Defines the command used to launch the -TotalView Debugger Server (tvdsrv) when remote processes are created. For information on this launch string, see “Replacement Characters” on page 352.

Permitted Values: A string, usually contained within braces \{}

---

**RogueWave**

TotalView Variables / TV:: Namespace 261
**Default:**

The default value depends upon the platform; use the `dset` command to see the default.

**TV::bulk_launch_tmpfile1_header_line**

Defines the header line used in the first temporary file for a bulk server launch operation. For information on this launch string, see “Replacement Characters” on page 352.

*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. For information on this launch string, see “Replacement Characters” on page 352.

*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. For information on this launch string, see “Replacement Characters” on page 352.

*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. For information on this launch string, see “Replacement Characters” on page 352.

*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. For information on this launch string, see “Replacement Characters” on page 352.

*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. For information on this launch string, see “Replacement Characters” on page 352.

*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  
*Default:* {}  

**TV::cell_spu_images_stop_regexp**
If set to a non-empty string and TV::ask_on_cell_spu_image_load is true, TotalView matches the SPU image's name with the regular expression. 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 or the file named by this variable.

*Permitted Values:* An ELF file name  
*Default:* spurs_jm.elf

**TV::cell_spurs_kernel_dll_regexp**
Defines a regular expression that matches the image path component name of the SPURS kernel SPU ELF image embedded in the libspurs.so DLL.
When TotalView sees a new image loaded into an SPU thread by `libspe` or `libspe2`, it checks if the image path component name matches this variable. If 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` or 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` or the file named by this variable.

**Permitted Values:** An ELF file name

**Default:** `spurs_tm.elf`

### TV::checksum_libraries

**Permitted Values:**

**Default:** `auto`

### TV::comline_patch_area_base

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

**Permitted Values:** A hexadecimal value indicating space accessible to TotalView

**Default:** `0xffffffffffffffff`

### TV::comline_patch_area_length

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

**Permitted Values:** true or false

**Default:** false

**TV::compiler_vars**

(HP and SGI only) When true, TotalView shows variables created by your Fortran compiler as well as the variables in your program. When false (which is the default), TotalView does not show the variables created by your compiler.
Some Fortran compilers (HP f90/f77, SGI 7.2 compilers) write debugging information that describes variables the compiler created to assist in some operations. For example, it could create a variable used to pass the length of character*(*) variables. You might want to set this variable to true if you are looking for a corrupted runtime descriptor.

You can override the value set to this variable in a startup file with these command-line options:

- **-compiler_vars**: sets this variable to true
- **-no_compiler_vars**: sets this variable to false

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

**TV::control_c_quick_shutdown**

When true, 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 314.

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

**TV::cray_xt_mrnet_filterlib_dir**

The path to the directory that contains MRNet filter libraries on a Cray compute node. This string is expanded before use. See the section “Replacement Characters” on page 352 for more information. This value is only used when TotalView uses MRNet on Cray systems.

  **Permitted Values**: A string indicating a path
  **Default**: /var/spool/alps/%A/toolhelper%A

**TV::cuda_debugger**

Indicates whether cuda debugging is currently enabled. This is a read-only variable.

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

Setting this variable overrides the C++ demangler used by default. Note that this value is ignored unless you also set the value of the **TV::force_default_cplus_demangler** variable. The following values are supported:

- **compaq**: HP cxx on running Linux-Alpha
- **gnu**: GNU C++ on Linux Alpha
- **gnu_dot**: GNU C++ Linux x86
- **gnu_v3**: GNU C++ Linux x86
- **hp**: HP aCC compiler
- **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:

- **xlf90**: IBM Fortran
- **fujitsu_f9x**: Fujitsu Fortran 9x
- **hpux11_64_f9x**: HP Fortran 9x
- **intel**: Intel Fortran 9x
- **sunpro_f9x_4**: Sun ProFortran 4
- **sunpro_f9x_5**: Sun ProFortran 5
  - **Permitted Values**: A string naming the compiler
  - **Default**: Derived from your platform and information within your program
TV::data_format_double

Defines the format 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 forms.

0 (zero)
   Pads a value with zeros. This is ignored if the number is left-justified. If you omit this character, TotalView pads the value with spaces.

-(hyphen)
   Left-justifies the value within the field's width.

**Permitted Values:** A value in the described format

**Default:** {auto %-1.15 %-1.15 %-20.2}

**TV::data_format_ext**

Defines the format to use when displaying extended floating point values such as long doubles. For a description of the contents of this variable, see **TV::data_format_double**.

**Permitted Values:** A value in the described format

**Default:** {auto %-1.15 %-1.15 %-1.15}

**TV::data_format_int8**

Defines the format to use when displaying 8-bit integer values. For a description of the contents of this variable, see **TV::data_format_double**.

**Permitted Values:** A value in the described format

**Default:** {auto %1.1 %#4.2 %#4.3}

**TV::data_format_int16**

Defines the format to use when displaying 16-bit integer values. For a description of the contents of this variable, see **TV::data_format_double**.

**Permitted Values:** A value in the described format

**Default:** {auto %1.1 %#6.4 %#7.6}

**TV::data_format_int32**

Defines the format to use when displaying 32-bit integer values. For a description of the contents of this variable, see **TV::data_format_double**.

**Permitted Values:** A value in the described format

**Default:** {auto %1.1 %#10.8 %#12.11}
**TV::data_format_int64**

Defines the format to use when displaying 64-bit integer values. For a description of the contents of this variable, see TV::data_format_double.

- **Permitted Values:** A value in the described format
- **Default:** {auto %1.1 %#18.16 %#23.22}

**TV::data_format_int128**

Defines the format to use when displaying 128-bit integer values. For a description of the contents of this variable, see TV::data_format_double.

- **Permitted Values:** A value of the described format.
- **Default:** {auto %1.1 %#34.32 %#44.43}

**TV::data_format_long_stringlen**

Defines the number of characters allowed in a long string.

- **Permitted Values:** A positive integer number
- **Default:** 8000

**TV::data_format_single**

Defines the format to use when displaying single precision, floating-point values. For a description of the contents of this variable, see TV::data_format_double.

- **Permitted Values:** A value in the described format
- **Default:** {auto %-1.6 %-1.6 %-1.6}

**TV::data_format_stringlen**

Defines the maximum number of characters displayed for a string.

- **Permitted Values:** A positive integer number
- **Default:** 100

**TV::dbfork**

When true, TotalView catches the fork(), vfork(), and execve() system calls if your executable is linked with the dbfork library. See “Linking with the dbfork Library” on page 364.

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

**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_stderr_is_stdout**
When true, TotalView writes the target program’s stderr information to the same location as stdout.

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

**TV::default_stdin_filename**
Names the file from which the target program reads stdin information.

- **Permitted Values:** A string indicating a pathname
- **Default:** None

**TV::default_stdout_append**
When true, TotalView appends the target program’s stdout information to the file set in the GUI, by the `-stdout` command-line option, or in the `TV::default_stdout_filename` variable. If no pathname is set, the value of this variable is ignored. If the file does not exist, TotalView creates it.

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

**TV::default_stdout_filename**
Names the file to which to write the target program’s stdout information. If the file exists, TotalView overwrites it. If the file does not exist, TotalView creates it.
**Permitted Values:** A string indicating a pathname

**Default:** None

**TV::display_assembler_symbolically**
When **true**, TotalView displays assembler locations as `label+offset`. When **false**, these locations are displayed as hexadecimal addresses.

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

**Default:** **false**

**TV::dll_ignore_prefix**
Defines a list of library files that will not result in a query to stop the process when loaded. This list contains a colon-separated list of prefixes. Also, TotalView will not ask if you would like to stop a process if:

- You also set the **TV::ask_on_dlopen** variable to **true**.
- The suffix of the library being loaded does **not** match a suffix contained in the **TV::dll_stop_suffix** variable.
- One or more of the prefixes in this list match the name of the library being loaded.

**Permitted Values:** A list of path names, each item of which is separated from another by a colon

**Default:** `/lib/::usr/lib/::usr/lpp/::usr/ccs/lib/::usr/dt/lib/::tmp/`

**TV::dll_read_all_symbols**
Always reads loader and debugging symbols of libraries named within this variable.

This variable is set to a colon-separated list of library names. A name can contain the * (asterisk) and ? (question mark) wildcard characters, which have their usual meaning:

- ***:** zero or more characters.
- **?:** a single character.

Because this is the default behavior, include only library names here that would be excluded because they are selected by a wildcard match within the **TV::dll_read_loader_symbols_only** and **TV::dll_read_no_symbols** variables.

**Permitted Values:** One or more library names separated by colons

**Default:** None

**TV::dll_read_loader_symbols_only**
When 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:

- *
- ?

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:

- *
- ?

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 (see “dlopen Options for Scalability” on page 379).
**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".

For a more complete explanation of **dlopen** event filtering, including use-case examples, please refer to “**dlopen Options for Scalability**” on page 379.

**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 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::dwhere_qualification_level**

Controls the amount of information displayed when you use the `dwhere` command. Here are three examples:

- `dset TV::dwhere_qualification_level +overload_list`
- `dset TV::dwhere_qualification_level -class_name`
- `dset TV::dwhere_qualification_level -parent_function`

You could combine these arguments into one command. For example:

- `dset TV::dwhere_qualification_level +overload_list \ -class_name -parent_function`

In these examples “+” means that the information should be displayed and “-” means the information should not be displayed.

The arguments to this command are:

- all
- class_name
- file_directory
- hint
- image_directory
- loader_directory
- member
• module
• node
• overload_list
• parent_function
• template_args
• type_name

The all argument is often used as follows:

```c
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 201.

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

Default: {xterm -e %E +%N %S}

**TV::env**

Names a variable that is already contained within your program's environment. This is a read-only variable and is set by using the -env command-line option. For more information, see -env variable=value on page 339.
To set this variable from within TotalView, use the **File > New Program** or **Process > Startup** dialog boxes.

**Permitted Values:** None. The variable is read-only.

**Default:** None

**TV::follow_clone**

When 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::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 `.gnu_debuglink` section within your process. If it is found, it looks for the file named in this section. If **false**, TotalView ignores the contents of this section. This means that a gnu_debuglink file will not be loaded. For more information, see “Using gnu_debuglink Files” on page 362.

**Permitted Values:** true or false

**Default:** true

**TV::gnu_debuglink_checksum**

When **true**, TotalView compares the checksum of the gnu_debuglink file against the checksum contained within the `.gnu_debuglink` section. TotalView will only load the information from the gnu_debuglink file when the checksums match. For more information, see “Using gnu_debuglink Files” on page 362.

**Permitted Values:** true or false

**Default:** true

**TV::gnu_debuglink_global_directory**

Names the directory to store gnu_debuglink files. For more information, see “Using gnu_debuglink Files” on page 362.

**Permitted Values:** A pathname within your file system. While this path can be relative, it is usually a full pathname.

**Default:** /usr/lib/debug

**TV::gnu_debuglink_global_search_path**

Defines the search path to use when searching for `.gnu_debuglink` files. You can use two substituting variables when assigning values:

- **%D**: The directory containing the `.gnu_debuglink` file.
- **%G**: The contents of the **TV::gnu_debuglink_global_directory** variable.
- **%/**: The target directory delimiter; for example “/”. 
For more information, see “Using gnu_debuglink Files” on page 362.

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::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:** tvdsrv

**TV::local_server_launch_string**
(Sun only) If TotalView will not be using the server contained in the same working directory as the TotalView executable, the contents of this string indicate the shell command that TotalView uses to launch this alternate server. For information on this launch string, see “Replacement Characters” on page 352.

- **Permitted Values:** A string enclosed with {} (braces) if it has embedded spaces
- **Default:** {%-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_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 `.tvdrc` file:

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

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

---

**TV::parallel**

When **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_configs**

Defines a new parallel configuration or overwrites an existing one. You can define this variable in a global `.tvdrc` to add new configurations or overwrite those provided by Rogue Wave. You can also define this variable in a `tvdrc` file contained within your `.totalview` directory.

For more information, see .

---

**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 alpha or 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.

  • **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_string**

Names the command string that TotalView uses to automatically launch the TotalView Debugger Server (tvdsvr) when debugging a remote process. This command string is executed by /bin/sh. By default, TotalView uses the command ssh -x to start the server (rsh on Sun SPARC), but you can use any other command that can invoke tvdsvr on a remote host. If no command is available for invoking a remote process, you can't automatically launch the server; therefore, you should set this variable to /bin/false. If you cannot automatically launch a server, you should also set the TV::server_launch_enabled variable to false. For information on this launch string, see “Replacement Characters” on page 352.

*Permitted Values:* A string  
*Default:* {%C %R -n "%B/tvdsvr -working_directory %D -callback %L -set_pw %P -verbosity %V %F"}  

**TV::server_launch_timeout**

Specifies the number of seconds to wait for a response from the TotalView Debugger Server (tvdsvr) that it has launched.

*Permitted Values:* An integer from 1 to 3600 (1 hour)  
*Default:* 30

**TV::server_response_wait_timeout**

Specifies how long to wait for a response from the TotalView Debugger Server (tvdsvr). Using a higher value may help avoid server timeouts if you are debugging across multiple nodes that are heavily loaded.

*Permitted Values:* An integer from 1 to 3600 (1 hour)  
*Default:* 30

**TV::share_action_point**

Indicates the scope in which TotalView places newly created action points. In the CLI, this is the dbarrier, dbreak, and dwatch commands. If true, newly created action points are shared across the group. If false, a newly created action point is active only in the process in which it is set.

As an alternative to setting this variable, you can select the Plant in share group check box in the Action Points Page in the File > Preferences dialog box. You can override this value in the GUI by selecting the Plant in share group checkbox in the Action Point > Properties dialog box.

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

A list that modifies the way in which TotalView handles signals. This list consists of a list of `signal_action` descriptions, separated by spaces:

```
signal_action[signal_action] ...
```

A `signal_action` description consists of an action, an equal sign (=), and a list of signals:

```
action=signal_list
```

An action can be one of the following: **Error**, **Stop**, **Resend**, or **Discard**.

A `signal_list` is a list of one or more signal specifiers, separated by commas:

```
signal_specifier[,signal_specifier] ...
```

A `signal_specifier` can be a signal name (such as **SIGSEGV**), a signal number (such as **11**), or a star (**_***), which specifies all signals. We recommend using the signal name rather than the number because number assignments vary across UNIX versions.

The following rules apply when you are specifying an `action_list`:

- If you specify an action for a signal in an `action_list`, TotalView changes the default action for that signal.
- If you do not specify a signal in the `action_list`, TotalView does not change its default action for the signal.
- If you specify a signal that does not exist for the platform, TotalView ignores it.
- If you specify an action for a signal twice, TotalView uses the last action specified. In other words, TotalView applies the actions from left to right.

If you need to revert the settings for signal handling to built-in defaults, use the **Defaults** button in the **File > Signals** dialog box.

For example, to set the default action for the **SIGTERM** signal to **Resend**, you specify the following action list:

```
{Resend=SIGTERM}
```

As another example, to set the action for **SIGSEGV** and **SIGBUS** to **Error**, the action for **SIGHUP** and **SIGTERM** to **Resend**, and all remaining signals to **Stop**, you specify the following action list:

```
{Stop=,* Error=SIGSEGV,SIGBUS Resend=SIGHUP,SIGTERM}
```

This action list shows how TotalView applies the actions from left to right.

1. Sets the action for all signals to **Stop**.
2. Changes the action for **SIGSEGV** and **SIGBUS** from **Stop** to **Error**.
3. Changes the action for **SIGHUP** and **SIGTERM** from **Stop** to **Resend**.

**Permitted Values:** A list of signals, as was just described
Default: This differs from platform to platform; type dset TV::signal_handling_mode to see what a platform's default values are

**TV::source_pane_tab_width**

Sets the width of the tab character that is displayed in the Process Window's Source Pane. You may want to set this value to the same value as you use in your text editor.

*Permitted Values:* An integer

*Default:* 8

**TV::spell_correction**

When you use the View > Lookup Function or View > Lookup Variable commands in the Process Window or edit a type string in a Variable Window, TotalView checks the spelling of your entries. By default (verbose), TotalView displays a dialog box before it corrects spelling. You can set this resource to brief to run the spelling corrector silently. (TotalView makes the spelling correction without displaying it in a dialog box first.) You can also set this resource to none to disable the spelling corrector.

*Permitted Values:* verbose, brief, or none

*Default:* verbose

**TV::stack_trace_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:** `{:c:include s:asm S:asm c:c include lex:c:include y:c:include bmap:c:include f:f77 F:f77 f90:f9x F90:F9x hpf:hpfp HPF:hpfp C:x:c++ c++:c++ C:+:c++ C+:c++ C:c++ h:x:c++:include hpp:c++:include h+h++:c++:include HXX:c++:include HPP:c++:include HH:c++:include H:c++:include ih:c++:include th:c++}`

**TV::target_platform**

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

**Permitted Values:** A list of platform names

**Default:** Platform-dependent

**TV::ttf**

When `true`, TotalView uses registered type transformations to change the appearance of data types that have been registered using the `TV::type_transformation` command.

**Permitted Values:** `true` or `false`

**Default:** `true`

**TV::ttf_max_length**

When transforming STL structures, TotalView must chase through pointers to obtain values. This number indicates how many of these pointers it should follow.

**Permitted Values:** an integer number

**Default:** `10000`
**TV::use_fast_trap**

Controls TotalView's use of the target operating system's support of the fast trap mechanism for compiled conditional breakpoints, also known as EVAL points. As of TotalView 8.7, when this variable was introduced, only AIX supported the fast trap mechanism for breakpoints, but we anticipate other operating systems adding support. You cannot interactively use this variable. Instead, you must set it within a TotalView startup file; for example, set its value with a .tvdrc file.

Your operating system may not be configured correctly to support this option. See the TotalView Release Notes on our web site for more information.

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

**TV::use_fast_wp**

Controls TotalView's use of the target operating system's support of the fast trap mechanism for compiled conditional watchpoints, also known as CDWP points. As of TotalView 8.7, when this variable was introduced, only AIX supported the fast trap mechanism for watchpoints, but we anticipate other operating systems adding support. You cannot interactively use this variable. Instead, you must set it within a TotalView startup file; for example, set its value with a .tvdrc file.

Your operating system may not be configured correctly to support this option. See the TotalView Release Notes on our web site for more information.

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

**TV::use_interface**

This variable is a synonym for TV::local_interface.

**TV::user_threads**

When true, it enables TotalView support for handling user-level (M:N) thread packages on systems that support two-level (kernel and user) thread scheduling.

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

**TV::version**

Indicates the current TotalView version. This is a read-only variable.

- **Permitted Values:** A string
- **Default:** Varies from release to release
**TV::visualizer_launch_enabled**
When true, TotalView automatically launches the Visualizer when you first visualize something. If you set this variable to false, TotalView disables visualization. This is most often used to stop evaluation points containing a $visualize directive from invoking the Visualizer.

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

**TV::visualizer_launch_string**
Specifies the command string that TotalView uses when it launches a visualizer. Because the text is actually used as a shell command, you can use a shell redirection command to write visualization datasets to a file (for example, “cat > your_file”).

*Permitted Values:* A string  
*Default:* %B/visualize

**TV::visualizer_max_rank**
Specifies the default value used in the Maximum permissible rank field in the Launch Strings Page of the File > Preferences dialog box. This field sets the maximum rank of the array that TotalView will export to a visualizer. The Visualizer cannot visualize arrays of rank greater than 2. If you are using another visualizer or just dumping binary data, you can set this value to a larger number.

*Permitted Values:* An integer  
*Default:* 2

**TV::warn_step_throw**
If this is set to true and your program throws an exception during a single-step operation, TotalView asks if you wish to stop the step operation. The process will be left stopped at the C++ run-time library's “throw” routine. If this is set to false, TotalView will not catch C++ exception throws during single-step operations. Setting it to false may mean that TotalView will lose control of the process, and you may not be able to control the program.

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

**TV::wrap_on_search**
When true, 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 only used when TotalView uses MRNet.

- **Permitted Values:** a remote launch command
- **Default:** `{}`

**TV::xplat_rsh_args**

A list of arguments that need to be given to the remote launch command. This string is passed to MRNet instead of using the XPLAT_RSH_ARGS environment variable. This value is only used when TotalView uses MRNet.

- **Permitted Values:** a space-separated list of remote launch arguments
- **Default:** `{}`

**TV::xterm_name**

The name of the program that TotalView should use when spawning the CLI. In most cases, you will set this using the `-xterm_name` command-line option.

- **Permitted Values:** a string
- **Default:** `xterm`
TV::MEMDEBUG:: Namespace

TV::MEMDEBUG::default_snippet_extent
Defines the number of code lines above and below point of allocation that the Memory Debugger saves when it is adding code snippets to saved output.

You can also set this value using a Memory Debugger preference.

Permitted Values: A positive integer number
Default: 5

TV::MEMDEBUG::do_not_apply_hia_defaults
If set to true, tells the Memory Debugger that it should use settings it finds in a default .hiarc file. Otherwise, the Memory Debuggers sets all options to off.

You can also set this value using a Memory Debugger preference.

Permitted Values: true or false
Default: false

TV::MEMDEBUG::hia_allow_ibm_poe
Tells the Memory Debugger if you can enable memory debugging on poe. As the default value is false, set this variable if you want memory debugging to be on by default. This variable is hardly ever used.

Permitted Values: true or false
Default: false

TV::MEMDEBUG::ignore_snippets
When true, the Memory Debugger ignores code snippets that it saved and instead locates the information from your program's files.

You can also set this value using Memory Debugger preference.

Permitted Values: true or false
Default: false

TV::MEMDEBUG::leak_check_interior_pointers
When true, the Memory Debugger considers a block as being referenced if a pointer is pointing anywhere within the block instead of just at the block's starting location. In most programs, the code should be keeping track of the block's boundary. However, if your C++ program is using multiple inheritance, you may be pointing into the middle of the block without knowing it.

Permitted Values: true or false
Default: true
**TV::MEMDEBUG::leak_detection_alignment**

Specifies the alignment and stride TotalView uses as it steps through memory looking for pointers during leak detection. If 0 (the default value), then TotalView defaults to using the size of a pointer, which varies according to platform and programming model. In normal circumstances you should not need to adjust the alignment.

*Permitted Values:* A non-negative integer number

*Default:* 0

**TV::MEMDEBUG::leak_max_cache**

Sets the size of the Memory Debugger’s cache. We urge you not to change this value unless your program is exceptionally large or are asked to make the change by someone on the TotalView support team.

*Permitted Values:* A positive integer number

*Default:* 4194304

**TV::MEMDEBUG::leak_max_chunk**

Tells the Memory Debugger how much memory it should obtain when it obtains memory from your operating system. You shouldn't change this value unless asked to by someone on the TotalView support team.

*Permitted Values:* A positive integer number

*Default:* 4194304

**TV::MEMDEBUG::shared_data_filters**

Names a filter definition file that is not located in the default directory. (The default directory is the *lib* subdirectory within the TotalView installation directory.) The contents of this variable are read when TotalView begins executing. Consequently, TotalView ignores any changes you make during the debugging session. The following example names the directory in which the filter file resides. This example assumes that filter has the default name, which is *tv_filters.tvd*.

```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
TV::GUI:: Namespace

NOTE >> The variables in this section have meaning (and in some cases, a value) only when you are using the TotalView GUI.

TV::GUI::chase_mouse
When this variable is set to true, TotalView displays dialog boxes at the location of the mouse cursor. If this is set to false, TotalView displays them centered in the upper third of the screen.

Permitted Values: true or false
Default: true

TV::GUI::display_bytes_kb_mb
When true, the Memory Debugger displays memory block sizes in megabytes. If set to false, it displays memory blocks sizes in kilobytes.

Permitted Values: true or false
Default: true

TV::GUI::display_font_dpi
Indicates the video monitor DPI (dots per inch) at which fonts are displayed.

Permitted Values: An integer
Default: 75

TV::GUI::enabled
When true, you invoked the CLI from the GUI or a startup script. Otherwise, this read-only value is false.

Permitted Values: true or false
Default: true if you are running the GUI even though you are seeing this in a CLI window; false if you are only running the CLI

TV::GUI::fixed_font
Indicates the specific font TotalView uses when displaying program information such as source code in the Process Window or data in the Variable Window. This variable contains the value set when you select a Code and Data Font entry in the Fonts Page of the File > Preferences dialog box.

This is a read-only variable.

Permitted Values: A string naming a fixed font residing on your system
Default: While this is platform specific, here is a representative value:-adobe-courier-medium-r-normal--12-120-75-75-m-70-normal-1
**TV::GUI::fixed_font_family**
Indicates the specific font TotalView uses when displaying program information such as source code in the Process Window or data in the Variable Window. This variable contains the value set when you select a **Code and Data Font** entry of the **Fonts** Page of the **File > Preferences** dialog box.

*Permitted Values:* A string representing an installed font family

*Default:* fixed

**TV::GUI::fixed_font_size**
Indicates the point size at which TotalView displays fixed font text. This is only useful if you have set a fixed font family because if you set a fixed font, the value entered contains the point size.

Font sizes are indicated using printer points.

*Permitted Values:* An integer

*Default:* 12

**TV::GUI::font**
Indicates the specific font used when TotalView writes information as the text in dialog boxes and in menu bars. This variable contains the information set when you select a **Select by full name** entry in the **Fonts** Page of the **File > Preferences** dialog box.

*Permitted Values:* A string naming a fixed font residing on your system. While this is platform specific, here is a representative value: `-adobe-helvetica-medium-r-normal--12-120-75-75-p-67-iso8859-1`

*Default:* helvetica

**TV::GUI::force_window_positions**
Setting this variable to true tells TotalView that it should use the version 4 window layout algorithm. This algorithm tells the window manager where to set the window. It also cascades windows from a base location for each window type. If this is not set, which is the default, newer window managers such as kwm or Enlightenment can use their smart placement modes.

Dialog boxes still chase the pointer as needed and are unaffected by this setting.

*Permitted Values:* true or false

*Default:* false

**TV::GUI::frame_offset_x**
Not implemented.

**TV::GUI::frame_offset_y**
Not implemented.
**TV::GUI::geometry_call_tree**

Specifies the position at which TotalView displays the *Tools > Call Tree* Window. This position is set using a list containing four values: the window's \(x\) and \(y\) coordinates. These are followed by two more values specifying the window's width and height.

If you set any of these values to 0 (zero), TotalView uses its default value. This means, however, you cannot place a window at \(x, y\) coordinates of 0, 0. Instead, you'll need to place the window at 1, 1.

If you specify negative \(x\) and \(y\) coordinates, TotalView aligns the window to the opposite edge of the screen.

**Permitted Values:** A list containing four integers indicating the window's \(x\) and \(y\) coordinates and the window's width and height

**Default:** \(\{0 \ 0 \ 0 \ 0\}\)

**TV::GUI::geometry_cli**

Specifies the position at which TotalView displays the *Tools > CLI* Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

**Permitted Values:** A list containing four integers indicating the window's \(x\) and \(y\) coordinates and the window's width and height

**Default:** \(\{0 \ 0 \ 0 \ 0\}\)

**TV::GUI::geometry_expressions**

Specifies the position at which TotalView displays the *Tools > Expression List* Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

**Permitted Values:** A list containing four integers indicating the window's \(x\) and \(y\) coordinates and the window's width and height

**Default:** \(\{0 \ 0 \ 0 \ 0\}\)

**TV::GUI::geometry_globals**

Specifies the position at which TotalView displays the *Tools > Program Browser* Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

**Permitted Values:** A list containing four integers indicating the window's \(x\) and \(y\) coordinates and the window's width and height

**Default:** \(\{0 \ 0 \ 0 \ 0\}\)

**TV::GUI::geometry_help**

Specifies the position at which TotalView displays the *Help* Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

**Permitted Values:** A list containing four integers indicating the window's \(x\) and \(y\) coordinates and the window's width and height
**TV::GUI::geometry_memory_stats**

Specifies the position at which TotalView displays the **Tools > Memory Statistics** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

*Permitted Values:* A list containing four integers indicating the window's x and y coordinate's and the window's width and height

*Default:* \(\{0 0 0 0\}\)

**TV::GUI::geometry_message_queue**

Specifies the position at which TotalView displays the **Tools > Message Queue** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

*Permitted Values:* A list containing four integers indicating the window's x and y coordinates and the window's width and height

*Default:* \(\{0 0 0 0\}\)

**TV::GUI::geometry_message_queue_graph**

Specifies the position at which TotalView displays the **Tools > Message Queue Graph** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

*Permitted Values:* A list containing four integers indicating the window's x and y coordinates and the window's width and height

*Default:* \(\{0 0 0 0\}\)

**TV::GUI::geometry_modules**

Specifies the position at which TotalView displays the **Tools > Fortran Modules** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

*Permitted Values:* A list containing four integers indicating the window's x and y coordinates and the window's width and height

*Default:* \(\{0 0 0 0\}\)

**TV::GUI::geometry_process**

Specifies the position at which TotalView displays the Process Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

*Permitted Values:* A list containing four integers indicating the window's x and y coordinates and the window's width and height

*Default:* \(\{0 0 0 0\}\)
**TV::GUI::geometry_ptset**
No longer used.

**TV::GUI::geometry_root**
Specifies the position at which TotalView displays the Root Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

- **Permitted Values**: A list containing four integers indicating the window's $x$ and $y$ coordinates and the window's width and height
- **Default**: {0 0 0 0}

**TV::GUI::geometry_thread_objects**
Specifies the position at which TotalView displays the Tools > Thread Objects Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

- **Permitted Values**: A list containing four integers indicating the window's $x$ and $y$ coordinates and the window's width and height
- **Default**: {0 0 0 0}

**TV::GUI::geometry_variable**
Specifies the position at which TotalView displays the Variable Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

- **Permitted Values**: A list containing four integers indicating the window's $x$ and $y$ coordinates and the window's width and height
- **Default**: {0 0 0 0}

**TV::GUI::geometry_variable_stats**
Specifies the position at which TotalView displays the Tools > Statistics Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

- **Permitted Values**: A list containing four integers indicating the window's $x$ and $y$ coordinates and the window's width and height
- **Default**: {0 0 0 0}

**TV::GUI::hand_cursor_enabled**
Specifies whether the cursor should change to a hand cursor when hovering over an element you can dive into in the source pane of the process window.

- **Permitted Values**: true or false
- **Default**: true
**TV::GUI::heap_summary_refresh**
Not user settable.

**TV::GUI::inverse_video**
Not implemented.

**TV::GUI::keep_expressions**
Deprecated.

**TV::GUI::keep_search_dialog**
When true, TotalView doesn't remove the Edit > Find dialog box after you select that dialog box's Find button. If you select this option, you will need to select the Close button to dismiss the Edit > Find box.

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

**TV::GUI::old_root_window**
When true, TotalView replaces the Root Window with the Root Window used in versions prior to TotalView 8.15. You can override this value using the following command-line options:

  - -oldroot sets this variable to true
  - -newroot sets this variable to false

  NOTE >> Using the previous-version Root Window may affect performance of applications containing thousands of threads/processes.

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

**TV::GUI::pop_at_breakpoint**
When true, TotalView sets the Open (or raise) process window at breakpoint check box to be selected by default. If this variable is set to false, 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
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
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 304
- “Creating Structure and Class Transformations” on page 306
- “C++View” on page 314
Why Type Transformations

Modern programming languages allow you to use abstractions such as structures, class, and STL data types such as lists, maps, multimeaps, sets, multiset, 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 306
- “build_struct_transform Function” on page 308
- “Type Transformation Expressions” on page 308
- “Using Type Transformations” on page 312

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**
build_struct_transform Function

The build_struct_transform routine used in the example in the previous section is a Tcl helper function that builds the callbacks and addressing expressions that TotalView needs when it transforms data. It has two required arguments: name and members.

name Argument

The name argument contains a regular expression that identifies the structure or class. In this example, struct is part of the identifier's name. It does not mean that you are creating a structure. In contrast, if stuff is class, you would type:

name {^class stuff$}

If you use a wildcard such as asterisk ()* or question mark (?), TotalView can match more than one thing. In some cases, this is what you want. If it isn't, you need to be more precise in your wildcard.

members Argument

The members argument names the elements that TotalView will include in the information it will display. This argument contains one or more lists. The example in the previous section contained three lists: year, pName, and pStreet. Here again is the pName list:

{ pName   { * pName  } }

The first element in the list is the display name. In most cases, this is the name that exists in the structure or class. However, you can use another name. For example, since the transformation dereferences the pointer, you might want to change its name to Name:

{ Name    { * pName   } }

The sublist within the list defines a type transformation expression. These expressions are discussed in the next section.

Type Transformation Expressions

The list that defines a member has a name component and sublist within the list. This sublist defines a type transformation expression. This expression tells TotalView what it needs to know to locate the member. The example in the previous section used two of the six possible expressions. The following list describes these expressions:

{member}

No transformation occurs. The structure or class member that TotalView displays is the same as it displays if you hadn't used a transformation. This is most often used for simple data types such as ints and floats.

{* expr}

Dereferences a pointer. If the data element is a pointer to an element, this expression tells TotalView to dereference the pointer and display the dereferenced information.
{\textit{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.

{\textit{expr + offset}}

Use the data whose location is an offset away from \textit{expr}. This behaves just like pointer arithmetic in C and C++. The result is calculated based on the size of the type that \textit{expr} points to:

\[ \text{result} = \textit{expr} + \text{sizeof}(\ast\textit{expr}) \ast \text{offset} \]

{\textit{expr -> expr}}

Names a subelement in a structure accessed using a pointer. This is used in the same way as the \textit{\textgreater\textless} operator in C and C++. You must type a space before and after the \textit{\textgreater\textless} operator.

{\textit{datatype cast expr}}

Casts a data type. For example:

{\textit{double cast national_debt}}

{\textit{N upcast expr}}

Converts the current class type into one of its base classes. For example:

{\textit{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 \texttt{int **pfoo}:

{\textit{foo { * { * pfoo}}}]

\textbf{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:

```cpp
tv::tff::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.
Creating Type Transformations

Creating Structure and Class Transformations

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:

---

Figure 9 – Transformed Class

![Figure 9 – Transformed Class](image-url)
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:

\texttt{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 315
- “Templates” on page 317
- “Precedence - Searching for TV_ttf_display_type” on page 318
- “TV_ttf_add_row” on page 318
- “Return values from TV_ttf_display_type” on page 319
- “Elision” on page 320
- “Other Constraints” on page 321
- “Safety” on page 321
- “Memory Management” on page 322
- “Multithreading” on page 322
- “Tips and Tricks” on page 323
- “Core Files” on page 323
- “Using C++View with ReplayEngine” on page 323
- “C” on page 325
- “Fortran” on page 326
- “Compiling and linking tv_data_display.c” on page 330
- “C++View Example Files” on page 331
- “Limitations” on page 332
- “Licensing” on page 332
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:

```cpp
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:

```cpp
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:

```cpp
class A {
   int i;
   char *s;
};
class B {
   A a;
   double d;
};
```

We can define the display callback functions as follows:

```cpp
int TV_ttf_display_type ( const A *a )
{
   /* NOTE: error checking of value returned from TV ttf add_row \ omitted */
   (void) TV_ttf_add_row ( "i", TV_ttf_type_int, &(a->i) );
   (void) TV_ttf_add_row ( "s", TV_ttf_type_ascii_string, a->s );

   /* indicate success to TotalView */
   TV_ttf_format_ok;
}

int TV_ttf_display_type ( const B *b )
{
   /* NOTE: error checking of value returned from TV ttf add_row \ omitted */
   (void) TV_ttf_add_row ( "a", "A", &(b->a) );
   (void) TV_ttf_add_row ( "d", "double", &(b->d) );

   /* indicate success to TotalView */
   return TV_ttf_format_ok;
}
```

For brevity and clarity, we have omitted all error checking of the value returned from `TV_ttf_add_row`. We will discuss the possible values that a `TV_ttf_display_type` may return later.

For now, we just return a simple success.

We could have made one or both of the display callbacks a class method:

```cpp
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 323.)

This one definition can be used for any instance of the template class. In some cases, however, you may want a specialized implementation of the display function. As an illustration, consider this:
int TV_ttf_display_type ( const BoundsCheckedArray<char> *s )
{
    TV_ttf_add_row ( "string", TV_ttf_type_ascii_string, \
                    s->get_array () );
    return TV_ttf_format_ok;
}

Here we want to tell TotalView to display the array horizontally as a string instead of vertically as an array. For this reason, we want to pass `TV_ttf_type_ascii_string` to `TV_ttf_add_row` as the name of the type instead of the name constructed by the implementation of the general template display function. We therefore define a special version of the display function to handle `BoundsCheckedArray<char>`.

One remaining issue relating to templates is arranging for the various template display function instances to be instantiated. It is unlikely that display functions will be called directly by your program. (Indeed, we mentioned earlier that `TV_ttf_add_row` should not be called other than as a result of a call initiated by TotalView.) Consequently, the template functions may well not be generated automatically. You can either arrange for functions to be referenced, such as by calling them in a controlled manner, or by explicit template instantiation:

```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 \texttt{TV_ttf_add_row} are:

**\texttt{TV_ttf_ec_ok}**

Indicates that the operation succeeded.

**\texttt{TV_ttf_ec_not_active}**

Indicates that \texttt{TV_ttf_add_row} was called when the type formatting facility is not active. This is most likely to occur if \texttt{TV_ttf_add_row} is called other than as a result of a call to a \texttt{TV_ttf_display_type} initiated by TotalView.

**\texttt{TV_ttf_ec_invalid_characters}**

Indicates that either the field name or the type name contained illegal characters, such as \texttt{newline} or \texttt{tab}.

**\texttt{TV_ttf_ec_buffer_exhausted}**

Indicates that the internal buffer used by \texttt{TV_ttf_add_row} to marshal your formatted data for onward transmission to TotalView is full. See “Tips and Tricks” on page 323 for suggestions for reducing the number of calls to \texttt{TV_ttf_add_row}.

**Return values from \texttt{TV_ttf_display_type}**

The set of values your \texttt{TV_ttf_display_function} may return to TotalView is defined in the enum \texttt{TV_ttf_format_result} given in the file \texttt{tv_data_display.h} included with your distribution of TotalView. These values are:

**\texttt{TV_ttf_format_ok}**

Your function should return this value if it has successfully formatted the data and successfully registered its output using \texttt{TV_ttf_add_row}.

**\texttt{TV_ttf_format_ok_elide}**

As \texttt{TV_ttf_format_ok} but indicates that the output may be subject to type elision (see below).

**\texttt{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:

```plaintext
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:

```plaintext
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:

```cpp
class A { ... };
class B { A a; ... };

int TV_ttf_display_type ( const A *a )
{
    (void) TV_ttf_add_row ( ... );
    return TV_ttf_format_ok;
}
int TV_ttf_display_type ( const B *b )
{
    (void) TV_ttf_add_row ( ... );
    (void) TV_ttf_add_row ( "a", "A", &(b->a) );
    return TV_ttf_format_ok;
}
```

Note the following:

- `TV_ttf_display_type ( const A *a )` may not add an object of type `A` (direct inclusion) nor one of type `B` (indirect inclusion).
- When viewing an object of type `B`, TotalView will invoke `TV_ttf_add_row ( const B * )`, and then `TV_ttf_add_row ( const *A )`.

Safety

When you stop your program to inspect data, objects might not be in a fully consistent state. This may happen in a number of circumstances, such as:

- Stopping in a the middle of a constructor or destructor.
- Displaying an object in scope, but before its constructor has been called.
- Viewing a dangling pointer to an object, that is, a pointer to an object in memory that has been released by the program. This may be stack memory, but also heap memory. (If the target is running with memory debugging enabled, then TotalView does check that the object to be displayed does not lie in a deallocated region. If it does, then it does not call your `TV_ttf_display_type`, and will display the data in their raw form. You should not, however, rely on this check.)
In the absence of C++View, this is not a problem, as displaying the data is just a matter of reading memory. However, with C++View, displaying data now involves executing functions in the target code. Your functions should be careful to check that the object to be displayed is in a consistent state. If you can’t establish that with certainty, then it should not attempt to format the data, and instead it should return `TV_ttf_format_failed`.

Otherwise, your target program may crash when you attempt to display an object at an inappropriate time. As with any function call made from TotalView (expression list, evaluation window, etc.), TotalView recovers from this in a limited manner by posting an error message and restoring the stack to its original state. However, the target code may be left in an inconsistent or corrupted state, and further progress may not be possible or useful.

You may not place a breakpoint in a `TV_ttf_display_type` function. If you do, the callback will be aborted similarly, and TotalView will display an error.

**Memory Management**

You must make sure that the formatted data you want displayed by TotalView (the data whose address you supply as the third parameter to `TV_ttf_add_row`) remains allocated after the call to your `TV_ttf_display_type` function returns. In practice this means that you shouldn't allocate these data on the stack. Your `TV_ttf_display_type` function may be called at anytime, including when your target program may be in the memory manager. For this reason it is inadvisable to allocate or deallocate dynamic memory in your `TV_ttf_display_type` functions. If the formatted data are manufactured, that is, generated by `TV_ttf_display_type` rather than already existing, then the memory for those data should be allocated during the target's normal course of execution.

You may find it convenient to have your program format data as part of its normal operations. That way there are no side-effects to worry about when TotalView calls your `TV_ttf_display_type` callback function.

The `field_name` and `type_name` string parameters to `TV_ttf_add_row` do not need to remain allocated after the call to `TV_ttf_add_row`.

**Multithreading**

Accessing shared data in multithreaded environments will usually need some sort of access control mechanism to protect its consistency and correctness. Your `TV_ttf_display_type` functions must be coded carefully if they need to access data that are usually protected by a lock or mutex. Attempting to take the lock or mutex may result in deadlock if the mutex is already locked.

Usually the threads in the program will have been stopped when TotalView calls the `TV_ttf_display_type` function. If the mutex is locked before TotalView calls `TV_ttf_display_type`, then an attempt by `TV_ttf_display_type` to lock the mutex will result in deadlock.
If you are designing a `TV_ttf_display_type` that needs to access data usually protected by a lock or mutex, consider whether you are able to determine whether the data are in a consistent state without having to take the lock. It might be enough to be able to determine whether the mutex is locked. If the data cannot be accessed safely, have the `TV_ttf_display_type` return `TV_ttf_format_failed` or `TV_ttf_format_raw` according to what fits best with your requirements.

**Tips and Tricks**

Consider constructing the type name on-the-fly. This can save time and memory. As an example, consider the `TV_ttf_display_type` for `BoundsCheckedArray<T>` we discussed earlier:

```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 *Getting Started with ReplayEngine*.

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`.

```c
/* Example program demonstrating TotalView's C++View with ReplayEngine. */
/* Run with (in both record and replay modes) and without ReplayEngine. */
/* Note how c in main is displayed in the various cases. */

#include <stdio.h>
#include "tv_data_display.h"

static int counter;

class C {
  public:
    int   value;

    C() : value(0) {};
    /* C */
};

int
TV_ttf_display_type(const C *c)
{
  int   ret_val = TV_ttf_format_ok;
  int   err;

  // if Replay is engaged, this write to the global is lost because
  // the ttf function is evaluated in volatile mode
  counter++;

  // error checking omitted for brevity
  (void) TV_ttf_add_row ("value", "int", &c->value);

  // show how many times we've been called. Will always be zero
```
// with Replay engaged because the update is lost when the
// call to TV_ttf_display_type returns.
(void) TV_ttf_add_row ( "number_of_times_called", "int", &counter );

    return ret_val ;
} /* TV_ttf_display_type */

int main(int argc, char *argv[])
{
    C c;

    c.value = 1;

    c.value++;      // should be 1 **before** this line is executed
    c.value++;      // should be 2 **before** this line is executed

    /* c.value should be 3 */

    return 0;
} /* main */

Compile and link the program with tv_data_display.c (see Compiling and linking tv_data_display.c). Follow this procedure:

1. Start the program under TotalView and enter the function main.
2. Dive on the local variable c, and note how the synthetic member number_of_times-called changes as you step through the program.
3. Restart, but this time with ReplayEngine engaged.
4. Notice the changes to the value member as you move forwards and backwards, and that the synthetic member number_of_times-called remains 0 because the increment in TV_ttf_display_type is lost when the function returns.

C

Although primarily intended for C++, C++View may be usable with C. C does not allow overloading so there may be at most one TV_ttf_display_type function with external linkage present. If you are interested in formatting only one type, then this restriction will not be constraining.

You may be able to work around this problem by defining separate TV_ttf_display_type functions as before, but placing each in a different file, and defining them to be static. Since the visibility of each definition is limited to the translation unit in which it appears, multiple functions can coexist.
This work-around, however, depends on the nature of the debug information emitted by the compiler. Some compilers do not place static functions in an indexable section in the debug information, or may try to optimize them out. If TotalView cannot find the function, it will not be called. TotalView cannot traverse the entire resolved symbol table to find these functions, as it would incur significant performance problems.

**Fortran**

Fortran variables don't readily lend themselves to transformation by C++View, but in some cases, such as when using a common block with Cray pointer variables, it is possible to set up a corresponding C structure and then use that type to push the transformation.

**Example**

Consider this test case using Cray pointers in a common block, including three parts:

- The Fortran code
- A common block defined in an include file
- The C code containing the C++View code

**The Fortran Code**

Here, the Fortran code sets up a common block with a few variables and then assigns them some values.

```fortran
program pointerp

call stuff

end

subroutine stuff

include 'foop.cmn'

foo = 42
ix = 11
iy = 12
iz = 13

call doit(ix)

call readit

return
end
```
subroutine doit (ix_x)
include 'foop.cmn'
ipxp = malloc (8*ix_x*foo)
ipyp = malloc (8*iy*iz)
xp = 3
yp = 5
return
end

subroutine readit
include 'foop.cmn'
xp = 4
return
end

The include File
The Fortran include file foop.cmn sets up a common block foo1 that corresponds to the C structure extern foo1_, both in bold below.
The include file, foop.cmn:
   integer :: foo, ix, iy, iz
   real(kind=8) :: xp, yp
   pointer (ipxp, xp(foo,ix))
   pointer (ipyp, yp(iy,iz))
   common /foo1/ ix, iy, iz, foo, ipxp, ipyp

The C Code
The C code fortranTV.c defines structure extern foo1_ aligned to the Fortran common block foo1. Then, in the
tv_ttf_display_type routine for the struct foo, the calls to TV_ttf_add_row follow the layout of the data in the
common block, allowing us to view the data as we want to see it
The C code, fortranTV.c:
#include <stdio.h>

#include "tv_data_display.h"

#ifdef __cplusplus
extern "C" {
#endif

#ifdef __cplusplus
#endif
extern struct foo { int x; } foo1;

#ifdef __cplusplus
}
#endif

// Routine data display declaration
int TV_ttf_display_type(const struct foo *parameter)
{
    // Assign 'data' to the start of the common block
    int *data = (int *)parameter;

    // Pick up the Cray pointer
    double **ptr = (double **) &data[4];
    char typeName[64];

    TV_ttf_add_row("ix", "int", &data[0]);
    TV_ttf_add_row("iy", "int", &data[1]);
    TV_ttf_add_row("iz", "int", &data[2]);
    TV_ttf_add_row("foo", "int", &data[3]);

    sprintf(typeName, "double[%d]", data[0]*data[3]);
    TV_ttf_add_row("ipxp", typeName, ptr[0]);

    sprintf(typeName, "double[%d]", data[1]*data[2]);
    TV_ttf_add_row("ipyp", typeName, ptr[1]);

    return TV_ttf_format_ok;
}

Compiling and Linking

First compile the TotalView tv_data_display.c routine, as described in “Compiling and linking tv_data_display.c” on page 330.

Build the program and the C program to add in the C++View transform:

    ifort -g -c pointerp.f

    ifort -g -c fortranTV.c -I$TVINCLUDE

Finally, link the program:

    ifort -g -o crayptr pointerp.o tv_data_display.o fortranTV.o

Debugging
When you debug, set a breakpoint on the return statement on line 20, in subroutine `doit`. Run to the breakpoint and then dive on the common block `foo1`.

**Figure 10 – Using C++ View with Fortran, diving on the Fortran pointer data**

![Image of C++ View with Fortran, diving on the Fortran pointer data](image1.png)

To see the data transformed more clearly, expand the type information (downward arrow with the + sign) and change the language to C or C++.

**Figure 11 – Using C++ View with Fortran, changing language to C++**

![Image of C++ View with Fortran, changing language to C++](image2.png)

Then change the type from `void` to `foo`, **Figure 12**.
Note that, while the original display of the common block shows the Cray pointers as integers (because a Cray pointer is actually an integer that holds only a memory address), the final, transformed display shows the data referenced by the pointers, or the arrays of doubles.

**Figure 12 – Using C++ View with Fortran, transform the type**

![Figure 12](image)

### 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/totalview.8.9.0-2/src
setenv TVINCLUDE /usr/local/toolworks/totalview.8.9.0-2/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:

- **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_added_row`.

- **cppview_example_3**
  A more complex example using templates, showing how a `TV_ttf_display_type` function can be either generic or specialized for a particular instantiation of a template class. It also demonstrates elision.

- **cppview_example_4**
  A more complex example showing the use of STL container classes, elision, and the different values that `TV_ttf_display_type` can return.

- **cppview_example_5**
  This example adds a synthetic member to a class, and can be used to explore how C++View behaves under ReplayEngine.

**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.
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.
Running TotalView

This section of the TotalView Reference Guide contains information about command-line options you use when starting TotalView and the TotalView Debugger Server.

Chapter 7, “TotalView Command Syntax,” on page 334
TotalView contains a great number of command-line options. Many of these options allow you to override default behavior or a behavior that you've set in a preference or a startup file.

Chapter 8, “TotalView Debugger Server Command Syntax,” on page 348
This chapter describes how you modify the behavior of the tvdsvr. These options are most often used if a problem occurs in launching the server or if you have some very specialized need. In most cases, you can ignore the information in this chapter.
This chapter describes the syntax of the `totalview` command. Topics in this chapter are:

- Command-Line Syntax
- Command-Line Options
Command-Line Syntax

**Format**

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

or

```
totalview [ options ] -args executable [ args ]
```

**Arguments**

- **options**
  - TotalView options.

- **executable**
  - Specifies the path name of the executable being debugged. This can be an absolute or relative path name. The executable must be compiled with debugging symbols turned on, normally the `-g` compiler option. Any 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**

TotalView is a source-level debugger with a motif-based graphic user interface and features for debugging distributed programs, multiprocess programs, and multithreaded programs. TotalView is available on a number of different platforms.

If you specify mutually exclusive options on the same command line (for example, `-dynamic` and `-no_dynamic`), the last option listed is used.
Command-Line Options

-a *args*

Pass all subsequent arguments (specified by *args*) to the program specified by *filename*. This option must be the last one on the command line.

-aix_use_fast_ccw

Defined only on AIX, a synonym for the platform-independent -use_fast_wp, for TotalView script backward compatibility. See -use_fast_wp for more information. You must set this option on the command line; you cannot set it interactively using the CLI.

-aix_use_fast_trap

Defined only on AIX, a synonym for the platform-independent -use_fast_trap, for TotalView script backward compatibility. See -use_fast_trap for more information. You must set this option on the command line; you cannot set it interactively using the CLI.

-args *filename [args]*

Specifies *filename* as the executable to debug, with *args* as optional arguments to pass to your program. This option must be listed last on the command line. You can also use --args instead of -args, for compatibility with other debuggers.

-background *color*

Sets the general background color to *color*.

-bg *color*

Same as -background.

Default: light blue

-bluegene_q_user_threads

Enables handling of user-level (M:N) thread packages on BlueGene/Q systems.

-no_bluegene_q_user_threads

(Default) Disables handling of user-level (M:N) thread packages, improving startup performance at high scale. There is usually a 1:1 correspondence between user-level threads and kernel-level threads on BlueGene/Q systems.

-compiler_vars

(Alpha, HP, and SGI only.) Shows variables created by the Fortran compiler, as well as those in the user’s program.

Some Fortran compilers (HP f90/f77, HP f90, SGI 7.2 compilers) output debugging information that describes variables the compiler itself has invented for purposes such as passing the length of character*(*) variables. By default, TotalView suppresses the display of these compiler-generated variables.

However, you can specify the -compiler_vars option to display these variables. This is useful when you are looking for a corruption of a run-time descriptor or are writing a -compiler.
-no_compiler_vars
(Default) Tells TotalView that it should not show variables created by the Fortran compiler.

-control_c_quick_shutdown-ccq
(Default) Tells TotalView to kills attached processes and exits.

-no_control_c_quick_shutdown -nccq
Invokes code that sometimes allows TotalView to better manage the way it kills parallel jobs when it works with management systems. This has only been tested with SLURM. It may not work with other systems.

cuda
(Default) Enables CUDA debugging with TotalView.

-no_cuda
Disables CUDA debugging. Any CUDA kernels launched on a GPU device are not seen by the debugger, so the debugger can only debug the host code. -nocuda is the identical command.

dbfork
(Default) Catches the fork(), vfork(), and execve() system calls if your executable is linked with the dbfork library.

-no_dbfork
Tells TotalView that it should not catch fork(), vfork(), and execve() system calls even if your executable is linked with the dbfork library.

default_parallel_attach_subset subset_specification
Specifies a set of MPI ranks to be attached to when an MPI job is created or attached to. The subset_specification is a space-separated list, the elements of which can be in one of these forms:

rank: that rank only
rank1-ranks2: all ranks between rank1 and rank2 inclusive
rank1-rank2:stride: every strideth rank between rank1 and rank2

A rank 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.

The default_parallel_attach_subset is used to initialize the -parallel_attach_subset property of an MPI starter process, which can be get or set in the CLI using:

TV::process get dpid parallel_attach_subset
TV::process set dpid parallel_attach_subset -subset_specification
The CLI and `dload -parallel_attach_subset -subset_specification` overrides the `default_parallel_attach_subset` and sets the `parallel_attach_subset` property of the process being attached or loaded.

```-demangler=compiler```

Overrides the demangler and mangler TotalView uses by default. The following indicate override options.

```-demangler=compaq``` HP cxx on Linux (alpha)

```-demangler=gnu``` GNU C++ on Linux Alpha

```-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```  
Set the name of the X Windows display to `displayname`. For example, `-display vinnie:0.0` will display TotalView on the machine named “vinnie.”

Default: The value of your DISPLAY environment variable.

```-dll_ignore_prefix list```  
The colon-separated argument to this option tells TotalView that it should ignore files having this prefix when making a decision to ask about stopping the process when it `dlopes` a dynamic library. If the DLL being opened has any of the entries on this list as a prefix, the question is not asked.

```-dll_stop_suffix list```  
The colon-separated argument to this option tells TotalView that if the library being opened has any of the entries on this list as a suffix, it should ask if it should open the library.

```-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`).

This setting impacts scalability in HPC computing environments. For details, see “Filtering dlopen Events” on page 379.
-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 dlopened events in parallel.

  This setting impacts scalability in HPC computing environments. For details, see “Handling dlopen Events in Parallel” on page 381.

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

-e commands
  Tells TotalView to immediately execute the CLI commands named within this argument. All information you enter here is sent directly to the CLI's Tcl interpreter. For example, the following writes a string to stdout:

  cli -e 'puts hello'

  You can have more than one -e option on a command line.

-ent
  Tells TotalView that it should only use an Enterprise license.

  -no_ent
  Tells TotalView that it should not use an Enterprise license. You may combine this with -no_team or -noteamplus.

-env variable=value
  Tells TotalView to add an environment variable to the environment variables passed to your program by the shell. If the variable already exists, it effectively replaces the previous value. You need to use this command for each variable being added; that is, you cannot add more than one variable with an env command.

-foreground color
  Sets the general foreground color (that is, the text color) to color.
**TotalView Command Syntax**

**Command-Line Options**

**-fg color**
Same as **foreground**.

**Default:** black

**-f9x_demangler=compiler**
Overrides the Fortran demangler and mangler TotalView uses by default. The following indicate override options.

**-demangler=spro_f9x_4** SunPro Fortran, 4.0 or later

**-demangler=xlf** IBM Fortran

**-global_types**
(Default) Let TotalView assume that type names are globally unique within a program and that all type definitions with the same name are identical. The C++ standard asserts that this must be true for standard-conforming code.

If this option is set, TotalView will attempt to replace an opaque type *(struct foo *p;)* declared in one module, with an identically named defined type in a different module.

If TotalView has read the symbols for the module containing the non-opaque type definition, then when displaying variables declared with the opaque type, TotalView will automatically display the variable by using the non-opaque type definition.

**-no_global_types**
Specifies that TotalView *cannot* assume that type names are globally unique in a program. You should specify this option if your code has multiple different definitions of the same named type, since otherwise TotalView can use the wrong definition for an opaque type.

**-gnu_debuglink**
Tells TotalView that if a program or library has a `.gnu_debug_link` section, it should look for a `gnu_debug_link` file. If found, TotalView reads the debugging information from this file.

**-no_gnu_debuglink**
Do not load information from a `gnu_debug_link` file even if the file has a `.gnu_debug_link` section.

**-gnu_debuglink_checksum**
Tells TotalView that it should validate the `gnu_debug_link` file’s checksum against the checksum contained in the process’s `gnu_debuglink` section.

**-no_gnu_debuglink_checksum**
Do not compare checksums. Only do this if you are absolutely certain that the debug file matches.

**-ipv6_support**
Directs TotalView to support IPv6 addresses.

**-no_ipv6_support**
(Default) Do not support IPv6 addresses.
-kcc_classes
(Defualt) Converts structure definitions output by the KCC compiler into classes that show base classes and virtual base classes in the same way as other C++ compilers. See the description of the TV::kcc_classes variable for a description of the conversions that TotalView performs.

-no_kcc_classes
Specifies that TotalView will not convert structure definitions output by the KCC compiler into classes. Virtual bases will show up as pointers, rather than as data.

-li
(Defualt) Loads action points automatically from the filename.TVD.v3breakpoints file, providing the file exists.

-nlb
Tells TotalView that it should not automatically load action points from an action points file.

-load_session session_name
Loads into TotalView the session named in session_name. If the preference “Show Startup Parameters when TotalView starts” is set, this option launches the Session Manager’s Program Session screen where you can edit the session’s properties and then launch the session; otherwise, the option immediately loads the session into TotalView, launching the Root and Process windows. 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.)

-memory_debugging
Enables memory debugging. By adding the following suboptions, you enable that particular feature using its the feature’s default configuration. In most cases, you will want to use one or more of the following sub-options.

-mem_detect_use_after_free
Tests for memory use after memory is freed.

-mem_detect_use_after_free
Tests for memory use after memory is freed.

-mem_guard_blocks
Surrounds allocated memory blocks with guard blocks.

-mem_hoard_freed_memory
Tells the Memory Debugger to hoard memory blocks instead of releasing them when a free() routine is called.

-mem_hoard_low_memory_threshold nnnn
Sets the low memory threshold amount. When memory falls below this amount an event will be fired.

-mem_notify_events
Turns on memory event notification.
-no_mem_notify_events turns event notification off.

-mem_paint_all
    Paint both allocated and deallocated blocks with a bit pattern.

-mem_paint_on_alloc
    Paint memory blocks with a bit pattern when they are allocated.

-mem_paint_on_dealloc
    Paint memory blocks with a bit pattern when they are freed.

-mem_red_zones_overruns
    Turn on testing for Red Zones overruns.

-mem_red_zones_size_ranges min:max,min:max,...
    Defines the memory allocations ranges for which Red Zones are in effect. Ranges can be specified as follows: x:y allocations from x to y.
    :y allocations from 1 to y
    x: allocations of x and higher
    x allocation of x

-mem_red_zones_underruns
    Turn on testing for Red Zones underruns.

-message_queue
    (Default) Enables the display of MPI message queues when debugging an MPI program.

-mqd
    Same as -message_queue.

-mqd
    Same as -message_queue.

-no_message_queue
    Disables the display of MPI message queues when you are debugging an MPI program. This might be useful if something is overwriting the message queues and causing TotalView to become confused.

-no_mqd
    Same as -no_message_queue.

-mpi starter
    Names the MPI that your program requires. The list of starter names that you enter are those that appear in the Parallel system pull down list contained within the New Program's Parallel tab. If the starter name has more than one word (for example, Open MPI), enclose the name in quotes. For example:

    -mpi "Open MPI"

-nodes
    Specifies the number of nodes upon which the MPI job will run.
-no_startup_scripts
Tells TotalView not to reference any initialization files during startup. Note that this negates all settings in all initialization files. Aliases are -nostartups 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
Tells TotalView that your application is using NPTL threads. You only need use this option if default cannot determine that you are using this threads package.

-no_nptl_threads
Tells TotalView that you are not using the NPTL threads package. Use this option if TotalView thinks your application is using it and it isn't.

-oldroot
Displays the Root Window used in versions prior to TotalView 8.15.0. Using --oldroot or --newroot overrides the TV::GUI::old_root_window value.

-newroot
(Default) Displays the new Root Window. This is useful when TV::GUI::old_root_window is set to true in the .tvdrc file and you wish to use the new Root Window.

-parallel
(Default) Enables handling of parallel program run-time libraries such as MPI, PE, and UPC.

-no_parallel
Disables handling of parallel program run-time libraries such as MPI, PE, and UPC. This is useful for debugging parallel programs as if they were single-process programs.

-parallel_attach option
Sets the action that TotalView takes when starting a parallel program. Possible options are:

-yes (default)
Attaches to all processes in a parallel program, unless the process being launched or attached to has a non-empty parallel_attach_subset property. In this case, only the subset of processes specified in the parallel_attach_subset are attached.

-no
Attaches to no processes in a parallel program.

-ask
Asks which processes to attach to by posting the subset attach dialog box if the debugger GUI is open.

This option works in concert with the parallel_attach_subset property (see -default_parallel_attach_subset) of an MPI starter process, which specifies a set of MPI tasks to attach to when the debugger launches or attaches to an MPI job.
Modifying this setting does not affect the `parallel_attach_subset` property itself.

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

- **patch_area_length** `length`
  Sets the length of the dynamically allocated patch space to this `length`. See “Allocating Patch Space for Compiled Expressions” in the TotalView Users 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.

- **processgrid**
  Displays the Processes/Ranks Tab in the Process Window. Note that enabling this tab can significantly affect performance, particularly for large, massively parallel applications.

  - **-noprocessgrid** (default)
    Does not display the Processes/Ranks Tab in the Process Window. This command qualifier is helpful when you wish to disable the Processes/Ranks Tab for a debug session and you have `TV::GUI::process_grid_wanted` set to `true` in your `.tvsrc` file.

- **remote** `hostname`[:`portnumber`]
  Debugs an executable that is not running on the same machine as TotalView. For `hostname`, you can specify a TCP/IP host name (such as `vinnie`) or a TCP/IP address (such as `128.89.0.16`). Optionally, you can specify a TCP/IP port number for `portnumber`, such as `:4174`. When you specify a port number, you disable the autolaunch feature. For more information on the autolaunch feature, see “Setting Single Process Server Launch” in the TotalView Users Guide.

  - **-r** `hostname`[:`portnumber`]
    Same as `-remote`.

- **replay**
  Enables the ReplayEngine when TotalView begins. This command-line option is ignored if you do not have a license for ReplayEngine.

- **s** `pathname`
  Specifies the path name of a startup file that will be loaded and executed. This path name can be either an absolute or relative name.
  
  You can add more than one `-s` option on a command line.
-serial device[options]
  Debugs an executable that is not running on the same machine as TotalView. For device, specify the device name of a serial line, such as /dev/com1. Currently, the only option you are allowed to specify is the baud rate, which defaults to 38400. For more information on debugging over a serial line, see “Debugging Over a Serial Line” in Chapter 4 of the TotalView Users Guide.

-search_path pathlist
  Specify a colon-separated list of directories that TotalView will search when it looks for source files. For example:
  totalview -search_path proj/bin:proj/util

-signal_handling_mode "action_list"
  Modifies the way in which TotalView handles signals. You must enclose the action_list string in quotation marks to protect it from the shell.
  An action_list consists of a list of signal_action descriptions separated by spaces:
    signal_action[ signal_action] ...
  A signal action description consists of an action, an equal sign (=), and a list of signals:
    action=signal_list
  An action can be one of the following: Error, Stop, Resend, or Discard, For more information on the meaning of each action, see Chapter 3 of the TotalView User Guide.
  A signal_specifier can be a signal name (such as SIGSEGV), a signal number (such as 11), or a star (*), which specifies all signals. We recommend that you use the signal name rather than the number because number assignments vary across UNIX sessions.
  The following rules apply when you are specifying an action_list:
  (1) If you specify an action for a signal in an action_list, TotalView changes the default action for that signal.
  (2) If you do not specify a signal in the action_list, TotalView does not change its default action for the signal.
  (3) If you specify a signal that does not exist for the platform, TotalView ignores it.
  (4) If you specify an action for a signal more than once, TotalView uses the last action specified.
  If you need to revert the settings for signal handling to built-in defaults, use the Defaults button in the File > Signals dialog box.
  For example, here’s how to set the default action for the SIGTERM signal to resend:
    "Resend=SIGTERM"
  Here’s how to set the action for SIGSEGV and SIGBUS to error, the action for SIGHUP to resend, and all remaining signals to stop:
    "Stop=* Error=SIGSEGV,SIGBUS Resend=SIGHUP"

-shm "action_list"
  Same as -signal_handling_mode.
-starter_args "arguments"
   Tells TotalView to pass arguments to the starter program. You can omit the quotation marks if arguments is just one string without any embedded spaces.

-stderr pathname
   Names the file to which 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
   Tells TotalView to append 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
   Tells TotalView to redirect the target program's stderr to stdout.

-direct 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
   Tells TotalView to append 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.

-tasks
   Specifies how many tasks that TotalView should launch for the job. This argument usually follows a -mpi command-line option.

-team
   Tells TotalView that it should only use a Team license.
   -no_team
      Tells TotalView that it should not use an Enterprise license. You may combine this with -no_ent or -noteamplus.

-teamplus
   Tells TotalView that it should only use a Team Plus license.
   -no_teamplus
      Tells TotalView that it should not use a Team PLus license. You may combine this with -no_ent or -noteam.

-tvhome pathname
   The directory from which TotalView reads preferences and other related information and the directory to which it writes this information.
**-use_fast_trap**

Controls TotalView's use of the target operating system's support of the fast trap mechanism for compiled conditional breakpoints, also known as EVAL points. As of TotalView 8.7, when this was introduced, only AIX supported the fast trap mechanism for breakpoints, but we anticipate other operating systems adding support. You must set this option on the command line; you cannot set it interactively using the CLI.

Your operating system may not be configured correctly to support this option. See the TotalView Release Notes on our web site for more information.

**-use_fast_wp**

Controls TotalView's use of the target operating system's support of the fast trap mechanism for compiled conditional watchpoints, also known as CDWP points. As of TotalView 8.7, when this was introduced, only AIX supported the fast trap mechanism for watchpoints, but we anticipate other operating systems adding support. You must set this option on the command line; you cannot set it interactively using the CLI.

Your operating system may not be configured correctly to support this option. See the TotalView Release Notes on our web site for more information.

**-user_threads**

(Default) Enables handling of user-level (M:N) thread packages on systems where two-level (kernel and user) thread scheduling is supported. (Note: This option does not apply to -BlueGene/Q systems; instead, see -bluegene_q_user_threads.)

**-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 will attempt to create an xterm window.
Overview

This chapter summarizes the syntax of the TotalView Debugger Server command, `tvdsvr`, which is used for remote debugging. Remote debugging occurs when you explicitly call for it or when you are using disciplines like MPI that startup processes on remote servers. For more information on remote debugging, refer to "Setting Up Remote Debugging Sessions" in the TotalView Users Guide.

Topics in this chapter are:

- The `tvdsvr` Command and Its Options
- Replacement Characters
The tvdsvr Command and Its Options

**tvdsvr** (-server | -callback hostname:port | -serial device) [other options]

### Description

**tvdsvr** allows TotalView to control and debug a program on a remote machine. To accomplish this, the **tvdsvr** program must run on the remote machine, and it must have access to the executables being debugged. These executables must have the same absolute path name as the executable that TotalView is debugging, or the **PATH** environment variable for **tvdsvr** must include the directories containing the executables.

You must specify a -server, -callback, or -serial option with the **tvdsvr** command. By default, TotalView automatically launches **tvdsvr** using the -callback option, and the server establishes a connection with TotalView. (Automatically launching the server is called autolaunching.)

If you prefer not to automatically launch the server, you can start **tvdsvr** manually and specify the -server option. Be sure to note the password that **tvdsvr** prints out with the message:

\[pw = \text{hexnumhigh:hexnumlow}\]

TotalView will prompt you for **hexnumhigh:hexnumlow** later. By default, **tvdsvr** automatically generates a password that it uses when establishing connections. If desired, you can set your own password by using the -set_pw option.

To connect to the **tvdsvr** from TotalView, you use the **File > New Program** Dialog Box and must specify the host name and TCP/IP port number, **hostname:portnumber** on which **tvdsvr** is running. Then, TotalView prompts you for the password for **tvdsvr**.

### Options

The following options name the port numbers and passwords that TotalView uses to connect with **tvdsvr**.

- **-callback hostname:port**
  (Autolaunch feature only) Immediately establishes a connection with a TotalView process running on **hostname** and listening on **port**, where **hostname** is either a host name or TCP/IP address. If **tvdsvr** cannot connect with TotalView, it exits.

  If you use the -port, -search_port, or -server options with this option, **tvdsvr** ignores them.

- **-callback_host hostname**
  Names the host upon which the callback is made. The **hostname** argument indicates the machine upon which TotalView is running. This option is most often used with a bulk launch.
-callback_ports port-list
Names the ports on the host machines that are used for callbacks. The port-list argument contains a comma-separated list of the host names and TCP/IP port numbers (hostname:port, hostname:port...) on which TotalView is listening for connections from tvdsvr. This option is most often used with a bulk launch.

For more information, see Chapter 4, “Setting Up Remote Debugging Sessions” in the TotalView Users Guide.

-debug_file console_outputfile
Redirects TotalView Debugger Server console output to a file named console_outputfile.

If console_outputfile is the string UNIQUE, the filename tv_dump.hostname.pid is used. If console_outputfile contains the string ‘$$’ (note the escaping single quotes), hostname.pid is substituted. UNIQUE and ‘$$’ are useful for separating the console output when running multiple tvdsvr processes.

Default: All console output is written to stderr.

-nodes_allowed num
Explicitly tells tvdsvr how many nodes the server supports and how many licenses it needs. This is only used for the Cray XT3.

-port number
Sets the TCP/IP port number on which tvdsvr should communicate with TotalView. If this port is busy, tvdsvr does not select an alternate port number (that is, it won’t communicate with anything) unless you also specify -search_port.

Default: 4142

-search_port
Searches for an available TCP/IP port number, beginning with the default port (4142) or the port set with the -port option and continuing until one is found. When the port number is set, tvdsvr displays the chosen port number with the following message:

   port = number

Be sure that you remember this port number, since you will need it when you are connecting to this server from TotalView.

-serial device[options]
Waits for a serial line connection from TotalView. For device, specifies the device name of a serial line, such as /dev/com1. The only option you can specify is the baud rate, which defaults to 38400. For more information on debugging over a serial line, see “Debugging Over a Serial Line” in the TotalView User Guide.

-server
Listens for and accepts network connections on port 4142 (default).

Using -server can be a security problem. Consequently, you must explicitly enable this feature by placing an empty file named tvdsvr.conf in your /etc directory. This file must be owned by user ID 0 (root). When tvdsvr encounters this option, it checks if this file exists. This file’s contents are ignored.
You can use a different port by using one of the following options: `-search_port` or `-port`. To stop `tvdsvr` from listening and accepting network connections, you must terminate it by pressing Ctrl+C in the terminal window from which it was started or by using the `kill` command.

**-set_pw hexnumhigh:hexnumlow**

Sets the password to the 64-bit number specified by the `hexnumhigh` and `hexnumlow` 32-bit numbers. When a connection is established between `tvdsvr` and TotalView, the 64-bit password passed by TotalView must match this password set with this option. `tvdsvr` displays the selected number in the following message:

```
pw = hexnumhigh:hexnumlow
```

We recommend using this option to avoid connections by other users.

If necessary, you can disable password checking by specifying the “-set_pw 0:0” option with the `tvdsvr` command. Disabling password checking is dangerous; it allows anyone to connect to your server and start programs, including shell commands, using your UID. Therefore, we do not recommend disabling password checking.

**-set_pws password-list**

Sets 64-bit passwords. TotalView must supply these passwords when `tvdsvr` establishes the connection with it. The argument to this command is a comma-separated list of passwords that TotalView automatically generates. This option is most often used with a bulk launch.

For more information, see Chapter 4, “Setting Up Remote Debugging Sessions” in the *TotalView Users Guide*.

**-verbosity level**

Sets the verbosity level of TotalView Debugger Server-generated messages to `level`, which may be one of `silent`, `error`, `warning`, or `info`.

Default: `info`

**-working_directory directory**

Makes `directory` the directory to which TotalView connects.

Note that the command assumes that the host machine and the target machine mount identical file systems. That is, the path name of the directory to which TotalView is connected must be identical on both the host and target machines.

After performing this operation, the TotalView Debugger Server is started.
Replacement Characters

When placing a tvdsvr command in a Server Launch or Bulk Launch string (see the File > Preferences command within the online Help for more information), you will need to use special replacement characters. When your program needs to launch a remote process, TotalView replaces these command characters with what they represent. Here are the replacement characters:

%A
Expands to the ALPS Application ID (**apid**), which is a unique identifier for an application started using ALPS aprun on Cray XT, XE, and XK. The token is used to construct server path references copied onto the compute nodes' ramdisk under the */var/spool/alps/apid* directory by the ALPS Tool Helper library.

%B
Expands to the bin directory where tvdsvr is installed.

%C
Is replaced by the name of the server launch command being used. On most platforms, this is **ssh -x**. On Sun SPARC, this command is **rsh**. If the **TVDSVRLAUNCHCMD** environment variable exists, TotalView will use its value instead of its platform-specific value.

%D
Is replaced by the absolute path name of the directory to which TotalView will be connected.

%F
Contains the “tracer configuration flags” that need to be sent to tvdsvr processes. These are system-specific startup options that the tvdsvr process needs.

%H
Expands to the host name of the machine upon which TotalView is running. (This replacement character is most often used in bulk server launch commands. However, it can be used in a regular server launch and within a tvdsvr command contained within a temporary file.)

%I
Expands to the **pid** of the MPI starter process. For example, it can contain **mpirun, aprun**, etc. It can also be the process to which you manually attach. If no **pid** is available, %I expands to 0.

%J
Expands to the job ID. For MPICH or poe jobs, is the contents of the **totalview_jobid** variable contained either in the starter or first process. If that variable does not exist, it is set to zero (“0”). If it is not appropriate for the kind of job being launched, its value is -1.

%K
Expands to the tvdsvr platform suffix string in situations where a different server must be used. On Blue Gene/L and Blue Gene/P, the %K expansion includes _bg1, on Blue Gene/Q it includes _bgq, and on Cray XT3 Catamount (RedStorm) it includes _rs.
When MRNet is being used as the debugger infrastructure, \texttt{\_mrnet} is appended to the normal \texttt{\%K} expansion. On Cray XT with MRNet enabled the \texttt{\%K} token is expanded to \texttt{\_mrnet}, while on Blue Gene/L or Blue Gene/P with MRNet enabled the \texttt{\%K} token is expanded to \texttt{\_bgl\_mrnet}. This convention allows MRNet-specific debugger servers to be launched only when MRNet is being used as the debugger infrastructure.

\texttt{\%L}

If TotalView is launching one process, this is replaced by the host name and TCP/IP port number (\texttt{hostname:port}) on which TotalView is listening for connections from \texttt{tvdsvr}.

If a bulk launch is being performed, TotalView replaces this with a comma-separated list of the host names and TCP/IP port numbers (\texttt{hostname:port,hostname:port,...}) on which TotalView is listening for connections from \texttt{tvdsvr}.

For more information, see Chapter 4, “Setting Up Remote Debugging Sessions” in the \textit{TotalView Users Guide}.

\texttt{\%M}

(Sun) Expands to the command name used for a local server launch.

\texttt{\%N}

Is replaced by the number of servers that TotalView will launch. This is only used in a bulk server launch command.

\texttt{\%P}

If TotalView is launching one process, this is replaced by the password that it automatically generated.

If a bulk launch is being performed, TotalView replaces this with a comma-separated list of 64-bit passwords.

\texttt{\%R}

Is replaced by the host name of the remote machine specified in the \texttt{\textbf{File > New Program}} command. When performing a bulk launch, this is replaced by a comma-separated list of the names of the hosts upon which TotalView will launch \texttt{tvdsvr} processes.

\texttt{\%S}

If TotalView is launching one process, it replaces this symbol with the port number on the machine upon which TotalView is running.

If a bulk server launch is being performed, TotalView replaces this with a comma-separated list of port numbers.

\texttt{\%t1} and \texttt{\%t2}

Is replaced by files that TotalView creates containing information it generates. This is only available in a bulk launch.

These temporary files have the following structure:

1. An optional header line containing initialization commands required by your system.
2. One line for each host being connected to, containing host-specific information.
3. An optional trailer line containing information needed by your system to terminate the temporary file.

The \texttt{\textbf{File > Preferences Bulk Server}} Page allows you to define templates for the contents of temporary files. These files may use these replacement characters. The \texttt{\%N}, \texttt{\%t1}, and \texttt{\%t2} replacement characters can only be used within header and trailer lines of temporary files. All other characters can be used in header or trailer lines.
or within a host line defining the command that initiates a single-process server launch. In header or trailer lines, they behave as defined for a bulk launch within the host line. Otherwise, they behave as defined for a single-server launch.

%U
(Sun) Expands to the local socket ID.

%V
Is replaced by the current TotalView verbosity setting.

%Z
Expands to the job ID. For MPICH or poe jobs, is the contents of the totalview_jobid variable contained either in the starter or first process. If that variable does not exist, it is set to zero ("0"). If it is not appropriate for the kind of job being launched, its value is -1.
PART III

Platforms and Operating Systems

The three chapters in this part of the Reference Guide describe information that is unique to the computers, operating systems, and environments in which TotalView runs.

Chapter 9, “Platforms and Compilers,” on page 356
Here you will find general information on the compilers and runtime environments that TotalView supports. This chapter also contains commands for starting TotalView and information on linking with the dbfork library.

Chapter 10, “Operating Systems,” on page 367
While how you use TotalView is the same on all operating systems, there are some things you will need to know that are differ from platform to platform.

Chapter 11, “Architectures,” on page 385
When debugging assembly-level functions, you will need to know how TotalView refers to your machines registers.
Chapter 9

Platforms and Compilers

Overview

This chapter describes the compilers and parallel runtime environments used on platforms supported by TotalView. You must refer to the TotalView Platforms and Systems Requirement Guide for information on the specific compiler and runtime environments that TotalView supports.

For information on supported operating systems, please refer to Chapter 10, “Operating Systems,” on page 367.

Topics in this chapter are:

• Compiling with Debugging Symbols
• Using gnu_debuglink Files
• Linking with the dbfork Library
Compiling with Debugging Symbols

You need to compile programs with the `-g` option and possibly other compiler options so that debugging symbols are included. This section shows the specific compiler commands to use for each compiler that TotalView supports.

**NOTE >>** Please refer to the release notes in your TotalView distribution for the latest information about supported versions of the compilers and parallel runtime environments listed here.

### Apple Running Mac OS X

On Mac OS, in all cases use the standard compiler invocation, just being sure to include the `-g` option.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absoft Fortran 77</td>
<td>f77 -g program</td>
</tr>
<tr>
<td></td>
<td>.f f77 -g program.for</td>
</tr>
<tr>
<td>Absoft Fortran 90</td>
<td>f90 -g program.f90</td>
</tr>
<tr>
<td>GCC C</td>
<td>gcc -g program.c</td>
</tr>
<tr>
<td>GCC C++</td>
<td>g++ -g program.cxx</td>
</tr>
<tr>
<td>GCC Fortran</td>
<td>g77 -g program.f</td>
</tr>
<tr>
<td>IBM xlC C</td>
<td>xlc -g program.c</td>
</tr>
<tr>
<td>IBM xlC C++</td>
<td>xlc -g program.cxx</td>
</tr>
<tr>
<td>IBM xlf Fortran 77</td>
<td>xlf -g program.f</td>
</tr>
<tr>
<td>IBM xlf90 Fortran 90</td>
<td>xlf90 -g program.f90</td>
</tr>
</tbody>
</table>

On Mac OS X, you can create 64-bit applications using GCC 4 by adding the `-m64` command-line option.

### HP-UX for IA-64

The following table lists the procedures to compile programs on HP HP-UX for IA-64.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC C</td>
<td>gcc -g program.c</td>
</tr>
<tr>
<td>GCC C++</td>
<td>g++ -g program.cxx</td>
</tr>
<tr>
<td>GCC Fortran</td>
<td>g77 -g program.f</td>
</tr>
</tbody>
</table>
IBM AIX on RS/6000 Systems

The following table lists the procedures to compile programs on IBM RS/6000 systems running AIX.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP C</td>
<td>cc -g program.c</td>
</tr>
<tr>
<td>HP aCC</td>
<td>aCC -g program.cxx</td>
</tr>
<tr>
<td>HP Fortran 77</td>
<td>f77 -g program.cc</td>
</tr>
<tr>
<td>HP Fortran 90</td>
<td>f90 -g program.cc</td>
</tr>
</tbody>
</table>

You can set up to seven variables when debugging threaded applications. Here's how you might set six of these variables within a C shell:

```
setenv AIXTHREAD_MNRATIO "1:1"
setenv AIXTHREAD_SLPRATIO "1:1"
setenv AIXTHREAD_SCOPE "S"
setenv AIXTHREAD_COND_DEBUG "ON"
setenv AIXTHREAD_MUTEX_DEBUG "ON"
setenv AIXTHREAD_RWLOCK_DEBUG "ON"
```

The first three variables must be set. Depending upon what you need to examine, you will also need to set one or more of the "DEBUG" variables.

The seventh variable, AIXTHREAD_DEBUG, should not be set. If it is, you should unset it before running TotalView.

NOTE >> Setting these variables can slow down your application's performance. None of them should be set when you are running non-debugging versions of your program.

When compiling with KCC, you must specify the -qnofullpath option; KCC is a preprocessor that passes its output to the IBM xlc C compiler. It will discard #line directives necessary for source-level debugging if you do not use the -qfullpath option. We also recommend that you use the +K0 option and not the -g option.
When compiling with `guidef77`, the `-WG,-cmpo=i` option may not be required on all versions because `-g` can imply these options.

When compiling Fortran programs with the C preprocessor, pass the `-d` option to the compiler driver. For example: `xlf -d - program.F`

If you will be moving any program compiled with any of the IBM `xl` compilers from its creation directory, or you do not want to set the search directory path during debugging, use the `-qfullpath` compiler option. For example:

`xlf -qfullpath -g -c program.f`

**IBM Blue Gene**

The following table lists the procedures to compile programs on IBM Blue Gene computers.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Visual Age C</td>
<td><code>xlc -g program.c</code></td>
</tr>
<tr>
<td>IBM Visual Age C++</td>
<td><code>xlc -g program.cxx</code></td>
</tr>
<tr>
<td>IBM Visual Age FORTRAN 77</td>
<td><code>xlf -g program.f</code></td>
</tr>
<tr>
<td>IBM Visual Age Fortran 90</td>
<td><code>xlf90 -g program.f90</code></td>
</tr>
</tbody>
</table>

**IBM Power Linux**

The following table lists the procedures to compile programs on the IBM Power Linux computer.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absoft Fortran 77</td>
<td><code>f77 -g program</code></td>
</tr>
<tr>
<td>Absoft Fortran 90</td>
<td><code>f90 -g program.f90</code></td>
</tr>
<tr>
<td>GCC C</td>
<td><code>gcc -g program.c</code></td>
</tr>
<tr>
<td>GCC C++</td>
<td><code>g++ -g program.cxx</code></td>
</tr>
<tr>
<td>IBM Visual Age C</td>
<td><code>xlc -g program.c</code></td>
</tr>
<tr>
<td>IBM Visual Age C++</td>
<td><code>xlc -g program.cxx</code></td>
</tr>
<tr>
<td>IBM Visual Age FORTRAN 77</td>
<td><code>xlf -g program.f</code></td>
</tr>
<tr>
<td>IBM Visual Age Fortran 90</td>
<td><code>xlf90 -g program.cc</code></td>
</tr>
</tbody>
</table>
## Linux Running on an x86 Platform

The following table lists the procedures to compile programs on Linux x86 platforms.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absoft Fortran 77</td>
<td><code>f77 -g program.f</code></td>
</tr>
<tr>
<td></td>
<td><code>.f f77 -g program.for</code></td>
</tr>
<tr>
<td>Absoft Fortran 90</td>
<td><code>f90 -g program.f90</code></td>
</tr>
<tr>
<td>Absoft Fortran 95</td>
<td><code>f95 -g program.f95</code></td>
</tr>
<tr>
<td>GCC C</td>
<td><code>gcc -g program.c</code></td>
</tr>
<tr>
<td>GCC C++</td>
<td><code>g++ -g program.cxx</code></td>
</tr>
<tr>
<td>G77</td>
<td><code>g77 -g program.f</code></td>
</tr>
<tr>
<td>Intel C++ Compiler</td>
<td><code>icc -g program.cxx</code></td>
</tr>
<tr>
<td>Intel Fortran Compiler</td>
<td><code>ifc -g program.f</code></td>
</tr>
<tr>
<td>Lahey/Fujitsu Fortran</td>
<td><code>lf95 -g program.f</code></td>
</tr>
<tr>
<td>PGI Fortran 77</td>
<td><code>pgf77 -g program.f</code></td>
</tr>
<tr>
<td>PGI Fortran 90</td>
<td><code>pgf90 -g program.f</code></td>
</tr>
</tbody>
</table>

## Linux Running on an x86-64 Platform

The following table lists the procedures to compile programs on Linux x86-64 platforms.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absoft Fortran 77</td>
<td><code>f77 -g program.f</code></td>
</tr>
<tr>
<td></td>
<td><code>.f f77 -g program.for</code></td>
</tr>
<tr>
<td>Absoft Fortran 90</td>
<td><code>f90 -g program.f90</code></td>
</tr>
<tr>
<td>Absoft Fortran 95</td>
<td><code>f95 -g program.f95</code></td>
</tr>
<tr>
<td>GCC C</td>
<td><code>gcc -g program.c</code></td>
</tr>
<tr>
<td>GCC C++</td>
<td><code>g++ -g program.cxx</code></td>
</tr>
<tr>
<td>G77</td>
<td><code>g77 -g program.f</code></td>
</tr>
<tr>
<td>Intel C++ Compiler</td>
<td><code>icc -g program.cxx</code></td>
</tr>
<tr>
<td>Intel Fortran Compiler</td>
<td><code>ifc -g program.f</code></td>
</tr>
<tr>
<td>Pathscale EKO C</td>
<td><code>pathcc -g program.f</code></td>
</tr>
<tr>
<td>Pathscale EKO C++</td>
<td><code>pathCC -g program.f</code></td>
</tr>
</tbody>
</table>
### Linux Running on an Itanium Platform

The following table lists the procedures to compile programs running on the Intel Itanium platform.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lahey/Fujitsu Fortran</td>
<td>lf95 -g program.f</td>
</tr>
<tr>
<td>PGI C++</td>
<td>pcCC -g program.f</td>
</tr>
<tr>
<td>PGI Fortran 77</td>
<td>pgf77 -g program.f</td>
</tr>
<tr>
<td>PGI Fortran 90</td>
<td>pgf90 -g program.f</td>
</tr>
</tbody>
</table>

### Sun Solaris

The following table lists the procedures to compile programs on SunOS 5 SPARC.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apogee C</td>
<td>apcc -g program.c</td>
</tr>
<tr>
<td>Apogee C++</td>
<td>apcc -g program.cxx</td>
</tr>
<tr>
<td>GCC C</td>
<td>gcc -g program.c</td>
</tr>
<tr>
<td>GCC C++</td>
<td>g++ -g program.cxx</td>
</tr>
<tr>
<td>Intel C++ Compiler</td>
<td>icc -g program.cxx</td>
</tr>
<tr>
<td>Intel Fortran Compiler</td>
<td>ifc -g program.f</td>
</tr>
<tr>
<td>Sun One Studio C</td>
<td>cc -g program.c</td>
</tr>
<tr>
<td>Sun One Studio C++</td>
<td>CC -g program.cxx</td>
</tr>
<tr>
<td>Sun One Studio Fortran 77</td>
<td>f77 -g program.f</td>
</tr>
<tr>
<td>Sun One Studio Fortran 90</td>
<td>f90 -g program.f90</td>
</tr>
</tbody>
</table>
Using gnu_debuglink Files

Some versions of Linux allow you to place debugging information in a separate file. These files, which can have any name, are called `gnu_debuglink` files. Because this information is stripped from the program's file, it almost always greatly reduces the size of your program. In most cases, you would create `gnu_debuglink` files for system libraries or other programs for which it is inappropriate to ship versions have debugging information.

After you create an unstripped executable or shared library, you can prepare the `gnu_debuglink` file as follows:

1. Create a `.debug` copy of the file. This second file will only contain debugging symbol table information. That is, it differs from the original in that it does not contain code or data.

Create this file on Linux systems that support the `-add-gnu-debuglink` and `-only-keep-debug` command-line options. If `objcopy --help` mentions `-add-gnu-debuglink`, you should be able to create this file. See `man objcopy` for more details.

   1. Create a stripped copy of the image file, and add a `.gnu_debuglink` section to the stripped file that contains the name of the `.debug` file and the checksum of the `.debug` file.

2. Distribute the stripped image and `.debug` files separately. The idea is that the stripped image file will normally take up less space on the disk, and if you want the debug information, you can also install the corresponding `.debug` file.

The following example creates the `gnu_debuglink` file for a program named `hello`. It also strips the debugging information from `hello`:

   ```
   objcopy --only-keep-debug hello hello.gnu_debuglink.debug
   objcopy --strip-all hello hello.gnu_debuglink
   objcopy --add-gnu-debuglink=hello.gnu_debuglink.debug \
       hello.gnu_debuglink
   ```

Total View Command-Line Options and CLI State Variables

The following command line options and CLI variables control how TotalView handles `.gnu_debuglink` files.

- `-gnu_debuglink` and `-no_gnu_debuglink`, `TV::gnu_debuglink`

  Controls Total View processing of the `.gnu_debuglink` section in executables and shared libraries; the default value is true. Setting the variable to false or using the `no_` command-line option prefix saves time when you do not want to process the debug-only files or when you need to avoid other problems associated with the debug-only files.

- `-[no_]gnu_debuglink_checksum` and `TV::gnu_debuglink_checksum_flag`
Tells TotalView if it should validate the checksum of the debug-only files against the checksum stored in the 
\texttt{.gnu_debuglink} section of the executable or shared library; the default is true. Setting the variable to false 
or using the \texttt{no} command-line option prefix can save time associated with computing the checksum of 
large files. Do this only if you are absolutely certain that the debug file matches.

- \texttt{-gnu_debuglink\_global\_directory} and \texttt{TV::gnu_debuglink\_global\_directory}
  Specifies the global debug directory; the default value is \texttt{/usr/lib/debug}.

\section*{Searching for the gnu\_debug\_link File}

If the \texttt{TV::gnu\_debuglink\_flag} variable is true and if the process contains a \texttt{.gnu\_debug\_link} section, TotalView 
searches for the \texttt{gnu\_debug\_link} file as follows:

1. In the directory containing the program.
2. In the \texttt{.debug} subdirectory of the directory containing the program.
3. In a directory named in the \texttt{TV::gnu\_debuglink\_global\_directory} variable.

For example, assume that the program's pathname is \texttt{/A/B/hello\_world} and the debug filename stored in the 
\texttt{.gnu\_debuglink} section of this program is \texttt{hello\_world.debug}, if the \texttt{TV::gnu\_debuglink\_global\_directory} vari-

- \texttt{/A/B/hello\_world.debug}
- \texttt{/A/B/.debug/hello\_world.debug}
- \texttt{/usr/lib/debug/A/B/hello\_world.debug}
Linking with the dbfork Library

If your program uses the `fork()` and `execve()` system calls, and you want to debug the child processes, you need to link programs with the dbfork library.

NOTE >> While you must link programs that use `fork()` and `execve()` with the TotalView dbfork library so that TotalView can automatically attach to them when your program creates them, programs that you attach to need not be linked with this library.

Linking with HP-UX

Add either the `-ldbfork` or `-ldbfork_64` argument to the command that you use to link your programs. If you are compiling 32-bit code, use one of the following arguments:

- `/opt/totalview/lib/hpux11-hppa/libdbfork.a`
- `-L/opt/totalview/hpux11-hppa/lib -ldbfork`

For example:
```
cc -n32 -o program program.c \n   -L/opt/totalview/hpux11-hppa/lib -ldbfork
```

If you are compiling 64-bit code, use the following arguments:

- `/opt/totalview/lib/hpux11-hppa/libdbfork_64.a`
- `-L/opt/totalview/hpux11-hppa/lib -ldbfork_64`

For example:
```
cc -64 -o program program.c \n   -L/opt/totalview/hpux11-hppa/lib \n   -ldbfork_64
```

As an alternative, you can set the `LD_LIBRARY_PATH` environment variable and omit the `-L` command-line option. For example:
```
setenv LD_LIBRARY_PATH \n   /opt/totalview/hpux11-hppa/lib
```

dfork on IBM AIX on RS/6000 Systems

Add either the `-ldbfork` or `-ldbfork_64` argument to the command that you use to link your programs. If you are compiling 32-bit code, use the following arguments:

- `/usr/totalview/lib/libdbfork.a` `-bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a`
• `-L/usr/totalview/lib -ldbfork -bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a`

For example:
```bash
c -o program program.c
  -L/usr/totalview/rs6000/lib -ldbfork
  -bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
```

If you are compiling 64-bit code, use the following arguments:

• `/usr/totalview/lib/libdbfork_64.a -bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a`

• `-L/usr/totalview/lib -ldbfork_64 -bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a`

For example:
```bash
c -o program program.c
  -L/usr/totalview/rs6000/lib -ldbfork
  -bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
```

When you use `gcc` or `g++`, use the `-Wl,-bkeepfile` option instead of using the `-bkeepfile` option, which will pass the same option to the binder. For example:
```bash
gcc -o program program.c
  -L/usr/totalview/rs6000/lib -ldbfork -Wl, -bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
```

### Linking C++ Programs with dbfork

You cannot use the `-bkeepfile` binder option with the IBM xIC C++ compiler. The compiler passes all binder options to an additional pass called `munch`, which will not handle the `-bkeepfile` option.

To work around this problem, we have provided the C++ header file `libdbfork.h`. You must include this file somewhere in your C++ program. This forces the components of the `dbfork` library to be kept in your executable. The file `libdbfork.h` is included only with the RS/6000 version of TotalView. This means that if you are creating a program that will run on more than one platform, you should place the `include` within an `#ifdef` statement's range. For example:
```cpp
#ifdef _AIX
  #include "/usr/totalview/include/libdbfork.h"
#endif
int main (int argc, char *argv[])
{
}
```

In this case, you would not use the `-bkeepfile` option and would instead link your program using one of the following options:

• `/usr/totalview/include/libdbfork.a`

• `-L/usr/totalview/include -ldbfork`
**Linux or Mac OS X**

Add one of the following arguments or command-line options to the command that you use to link your programs:

- `/usr/totalview/platform/lib/libdbfork.a`
- `-L/usr/totalview/platform/lib -ldbfork` or `-L/usr/totalview/platform/lib -ldbfork_64`

where `platform` is one of the following: `darwin-power`, `linux-x86`, `linux-x86-64`, or `linux-ia64`.

In general, 32-bit programs use `libdbfork.a` and 64-bit programs use `libdbfork_64.a`. Of course, if your architecture doesn't support 32-bit programs, the option won't work.

For example:

```bash
cc -o program program.c \
     -L/usr/totalview/linux-x86/lib -ldbfork
```

However, `linux-ia64` uses `libdbfork` for 64-bit programs.

**SunOS 5 SPARC**

Add one of the following command line arguments or options to the command that you use to link your programs:

- `/opt/totalview/sun5/lib/libdbfork.a`
- `-L/opt/totalview/sun5/lib -ldbfork`

For example:

```bash
cc -o program program.c \
     -L/opt/totalview/sun5/lib -ldbfork
```

As an alternative, you can set the `LD_LIBRARY_PATH` environment variable and omit the `-L` option on the command line:

```bash
setenv LD_LIBRARY_PATH /opt/totalview/sun5/lib
```
This chapter describes the operating system features that can be used with TotalView. This chapter includes the following topics:

- Supported Operating Systems
- Troubleshooting Mac OS X Installations
- Mounting the /proc File System (SunOS 5 only)
- Swap Space
- Shared Libraries
- Debugging Your Program's Dynamically Loaded Libraries
- Remapping Keys (Sun Keyboards only)
- Expression System
Supported Operating Systems

Here is an overview of operating systems and some of the environments supported by TotalView at the time when this book was printed. As this book isn't printed nearly as often as vendors update compilers and operating systems, the compiler and operating system versions mentioned here may be obsolete. For a definitive list, see the most recent platform guide on our website. You can download this document at http://www.roguewave.com/products-services/totalview, and selecting Supported Platforms to download the most recent platform guide.

- Apple Macintosh OS X 10.7, 10.8, and 10.9.
- Cray XT, XE, and XK running the Cray Linux Environment (CLE), version 2.2 or later.
- HP HP-UX for IA-64 systems running version 2.0 (11.23).
- IBM Blue Gene systems running Linux on the front end nodes.
- IBM RS/6000 and SP systems running AIX versions 5.3L, 6.1, and 7.1.
- Sun x86_64, Solaris 10
- Sun SPARC Solaris 10.
- Linux: see the Platforms Guide.
Troubleshooting Mac OS X Installations

Problem Description

At TotalView startup, the OS checks whether the Mach system call `-task_for_pid()` is working properly. If the call returns an error, no debugging is possible, and TotalView outputs an error message that begins “The Mach system call -task_for_pid() is not working properly.” TotalView cannot distinguish the circumstances that can lead to this error, which are varied and depend on the version of OS X. The following sections describe a series of steps to troubleshoot this problem.

For All OS X Versions

**Requirement:** The TotalView executables must be installed `setgid` procmod, and must be installed on a file system that is mounted `suid` (that is, the file system must not be mounted `nosuid`). The TotalView installation procedure warns if these requirements are not met at installation time.

**Remedies:** Log in to the Mac host as `root`. If the file system where TotalView is to be installed is mounted `nosuid`, remove that attribute and re-mount the file system. Then (while still logged in as `root`) re-install TotalView.

For OS X Versions 10.8 (Mountain Lion) or Later

**Requirements:**

- The TotalView executables must be codesigned.
- The Mac host’s system security policy must have `Developer mode` enabled.
- TotalView users must be members of the `_developer` group, and must run TotalView within the context of a login session that has been validated by a password challenge at the OS X console.

**Remedies:** It can be difficult to distinguish which of the Mountain Lion requirements are unmet, so several remedial steps are recommended. We suggest you follow them in the order given, which is from least to most intrusive.

**Step 1**

Go to the console of the Mac host, log in as a TotalView user, and launch TotalView. A password challenge should appear, with a message that “Developer Tools Access” is trying to take control of a process. Enter the TotalView user’s password at the challenge prompt. If TotalView still fails to launch, move on to the next remedial step.
If TotalView launches, the installation is OK, but note that if a user wishes to run TotalView at a different physical display, it is necessary to launch an `xterm` from the *same* console login session. The `xterm` can be displayed wherever is convenient, and it will inherit the validation of the console session, so TotalView will run successfully from that `xterm`.

**Step 2**

Check the TotalView user’s group memberships by issuing the command `id -Gn` from a user session. If `_developer` is among the group names displayed, this is not the problem, so move on to the next remedial step.

If `_developer` is not in the list, you must add the user to that group. Issue the following command from an Administrator or root session:

```
dsc .. append /Groups/_developer GroupMembership username
```

After adding the `_developer` group membership, verify that the user can now run TotalView from the Mac host's console (as in Step 1).

**Step 3**

At this point, the problem is likely to be with the codesigning of the TotalView executables, the certificate used to sign them, or the system security policy configuration.

- If TotalView was not originally installed by `root`, log in to the Mac host as `root`, re-install, and verify that the TotalView user can now run TotalView from the Mac host's console (as in Step 1).

- If TotalView was originally installed by `root`, please contact TotalView support.

**Remotely Debugging without Console Access**

If you want to debug a remote machine to which you do not have console access, you can try the following procedure:

1. Install XQuartz and TotalView on the remote machine, and then perform the following steps on that remote machine:

2. Allow X11 forwarding in the `sshd_config` file (disabled by default).

3. Make sure every user who might need to debug is in the `_developer` group.


5. Type: `sudo security authorizationdb write system.privilege.taskport allow`.

The last step above is necessary to allow the launching of TotalView when not on the console.
Mounting the /proc File System

To debug programs on SunOS 5 with TotalView, you need to mount the /proc file system.

If you receive one of the following errors from TotalView, the /proc file system might not be mounted:

- job_t::launch, creating process: process not found
- Error launching process while trying to read -dynamic symbols
- Creating Process... Process not found Clearing Thrown Flag Operation Attempted on an unbound process object

To determine whether the /proc file system is mounted, enter the appropriate command from the following table.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunOS 5</td>
<td>% /sbin/mount</td>
</tr>
</tbody>
</table>

If you receive one of these messages from the mount command, the /proc file system is mounted.

Mounting /proc with SunOS 5

To make sure that the /proc file system is mounted each time your system boots, add the appropriate line from the following table to the appropriate file.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Name of File</th>
<th>Line to add</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunOS 5</td>
<td>/etc/vfstab</td>
<td>/proc - /proc proc - no -</td>
</tr>
</tbody>
</table>

Then, to mount the /proc file system, enter the following command:

/sbin/mount /proc
Swap Space

Debugging large programs can exhaust the swap space on your machine. If you run out of swap space, TotalView exits with a fatal error, such as:

- **Fatal Error: Out of space trying to allocate**

This error indicates that TotalView failed to allocate dynamic memory. It can occur anytime during a debugging session. It can also indicate that the data size limit in the C shell is too small. You can use the C shell's `limit` command to increase the data size limit. For example:

```
limit datasize unlimited
```

- **job_t::launch, creating process: Operation failed**

This error indicates that the `fork()` or `execve()` system call failed while TotalView was creating a process to debug. It can happen when TotalView tries to create a process.

Swap Space on HP HP-UX

The `swapinfo` command on an HP-UX system lets you find out how much swap space is allocated and is being used.

To find out how much swap space is being used while TotalView is running, enter:

```
/usr/bin/ps -lf
```

Here is an example of what you might see. The `SZ` column shows the pages occupied by a program.

To add swap space, use the `/usr/sbin/swapon` or `SAM` (System Administration Manager) utility. If you use `SAM`, invoke the `Swap` command in the `Disks and File Systems` menu.

**Maximum Data Size**

To see the current data size limit in the C shell, enter:

```
limit datasize
```

The following command displays the current `hard` limit:

```
limit -h datasize
```

If the current limit is lower than the hard limit, you can easily raise the current limit. To change the current limit, enter:

```
limit datasize new_data_size
```

If the hard limit is too low, you must reconfigure and rebuild the kernel, and then reboot. This is most easily done using `SAM`.

---

RogueWave

 Operating Systems / Swap Space  372
To change `maxdsiz`, use the following path through the SAM menus:

```
Kernel Configuration > Configurable Parameters > maxdsiz > Actions > Modify
Configurable Parameter > Specify New Formula/Value > Formula/Value
```

You can now enter the new maximum data segment size.

You may also need to change the value for `maxdsiz_64`.

Here is the command that lets you rebuild the kernel with these changed values:

```
Configurable Parameter > Actions > Process New Kernel
```

Answer *yes* to process the kernel modifications, *yes* to install the new kernel, and *yes* again to reboot the machine with the new kernel.

When the machine reboots, the value you set for `maxdsiz` should be the new hard limit.

### Swap Space on IBM AIX

To find out how much swap space has been allocated and is currently being used, use the `/usr/sbin/pstat -s` command:

To find out how much swap space is in use while you are running TotalView:

1. Start TotalView with a large executable:
   ```
totalview executable
   ```
   Press Ctrl+Z to suspend TotalView.

1. Use the following command to see how much swap space TotalView is using:
   ```
   ps u
   ```
   For example, in this case the value in the SZ column is 5476 KB:
   ```
   USER    PID %CPU %MEM   SZ  RSS   TTY  ...
   smith 15080  0.0 6.0   5476 547 pts/1 ...
   ```
   To add swap space, use the AIX system management tool, `smit`. Use the following path through the `smit` menus:

```
System Storage Management > Logical Volume Manager > Paging Space
```

#### Swap Space on Linux

To find out how much swap space has been allocated and is currently being used, use either the `swapon` or `top` commands on Linux:

You can use the `mkswap(8)` command to create swap space. The `swapon(8)` command tells Linux that it should use this space.
Swap Space on SunOS 5

To find out how much swap space has been allocated and is currently being used, use the `swap -s` command:

```
To find out how much swap space is in use while you are running TotalView:

1. Start TotalView with a large executable:
   `totalview executable`

Press Ctrl+Z to suspend TotalView.

1. Use the following command to see how much swap space TotalView is using:
   `/bin/ps -l`

To add swap space, use the `mkfile(1M)` and `swap(1M)` commands. You must be `root` to use these commands. For more information, refer to the online manual pages for these commands.
Shared Libraries

TotalView supports dynamically linked executables, that is, executables that are linked with shared libraries.

When you start TotalView with a dynamically linked executable, TotalView loads an additional set of symbols for the shared libraries, as indicated in the shell from which you started TotalView. To accomplish this, TotalView:

1. Runs a sample process and discards it.
2. Reads information from the process.
3. Reads the symbol table for each library.

When you create a process without starting it, and the process does not include shared libraries, the PC points to the entry point of the process, usually the **start** routine. If the process does include shared libraries, TotalView takes the following actions:

- Runs the dynamic loader (SunOS 5: **ld.so**, Linux: **/lib/ld-linux.so.**?).
- Sets the PC to point to the location after the invocation of the dynamic loader but before the invocation of C++ static constructors or the **main()** routine.

**NOTE >>** On HP-UX, TotalView cannot stop the loading of shared libraries until after static constructors on shared library initialization routines have been run.

When you attach to a process that uses shared libraries, TotalView takes the following actions:

- If you attached to the process after the dynamic loader ran, then TotalView loads the dynamic symbols for the shared library.
- If you attached to the process before it runs the dynamic loader, TotalView allows the process to run the dynamic loader to completion. Then, TotalView loads the dynamic symbols for the shared library.

If desired, you can suppress the recording and use of dynamic symbols for shared libraries by starting TotalView with the **-no_dynamic** option. Refer to Chapter 1, “TotalView Command Syntax,” for details on this TotalView startup option.

If a shared library has changed since you started a TotalView session, you can use the **Group > Rescan Library** command to reload library symbol tables. Be aware that only some systems such as AIX permit you to reload library information.
Changing Linkage Table Entries and LD_BIND_NOW

If you are executing a dynamically linked program, calls from the executable into a shared library are made using the Procedure Linkage Table (PLT). Each function in the dynamic library that is called by the main program has an entry in this table. Normally, the dynamic linker fills the PLT entries with code that calls the dynamic linker. This means that the first time that your code calls a function in a dynamic library, the runtime environment calls the dynamic linker. The linker will then modify the entry so that next time this function is called, it will not be involved.

This is not the behavior you want or expect when debugging a program because TotalView will do one of the following:

- Place you within the dynamic linker (which you don’t want to see).
- Step over the function.

And, because the entry is altered, everything appears to work fine the next time you step into this function.

On most operating systems (except HP), you can correct this problem by setting the LD_BIND_NOW environment variable. For example:

```
setenv LD_BIND_NOW 1
```

This tells the dynamic linker that it should alter the PLT when the program starts executing rather than doing it when the program calls the function.

HP-UX does not have this (or an equivalent) variable. On HP systems, you can avoid this problem by using the -B immediate option the executable being debugged, or by invoking chatr with the -B immediate option. (See the chatr documentation for complete information on how to use this command.)

You will also have to enter pxdb -s on.

Using Shared Libraries on HP-UX

The dynamic library loader on HP-UX loads shared libraries into shared memory. Writing breakpoints into code sections loaded in shared memory can cause programs not under TotalView control to fail when they execute an unexpected breakpoint.

If you need to single-step or set breakpoints in shared libraries, you must set your application to load those libraries in private memory. This is done using HP's pxdb command.

```
pxdb -s on appname  (load shared libraries into private memory)
pxdb -s off appname (load shared libraries into shared memory)
```

For 64-bit platforms, use pxdb64 instead of pxdb. If the version of pxdb64 supplied with HP's compilers does not work correctly, you may need to install an HP-supplied patch. You will find additional information in the TotalView Release Notes at http://www.roguewave.com/help-support/documentation/totalview.
Debugging Your Program’s Dynamically Loaded Libraries

TotalView automatically reads the symbols of shared libraries that are dynamically loaded into your program at runtime. These libraries are those loaded using `dlopen` (or, on IBM AIX, `load` and `loadbind`).

TotalView automatically detects these calls, and then loads the symbol table from the newly loaded libraries and plants any enabled saved breakpoints for these libraries. TotalView then decides whether to ask you about stopping the process to plant breakpoints. You will set these characteristics by using the Dynamic Libraries page in the File > Preferences Dialog Box.

Figure 13 – File > Preferences Dialog Box: Dynamic Libraries Page

TotalView decides according to the following rules:

1. If either the Load symbols from dynamic libraries or Ask to stop when loading dynamic libraries preference is set to false, TotalView does not ask you about stopping.

2. If one or more of the strings in the When the file suffix matches preference list is a suffix of the full library name (including the path), TotalView asks you about stopping.

3. If one or more of the strings in the When the file path prefix does not match list is a prefix of the full library name (including the path), TotalView does not ask you about stopping.
4. If the newly loaded libraries have any saved breakpoints, TotalView does not ask you about stopping.

5. If none of the rules above apply, TotalView asks you about stopping.

If TotalView does not ask you about stopping the process, the process is continued.

If TotalView decides to ask you about stopping, it displays a dialog box, asking if it should stop the process so you can set breakpoints. To stop the process, answer **Yes**.

**Figure 14 – Stop Process Question Dialog Box**

![Stop Process Question Dialog Box](image)

To allow the process to continue executing, answer **No**. Stopping the process allows you to insert breakpoints in the newly loaded shared library.

Do either or both of the following to tell TotalView if it should ask:

- If you can set the `-ask_on_dlopen` command-line option to **true**, or you can set the `-no_ask_on_dlopen` option to false.

- Unset the **Load symbols from dynamic libraries** preference.

The following table lists paths where you are not asked if TotalView should stop the process:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP-UX</td>
<td><code>/usr/lib/</code> '/usr/lib/pa20_64`</td>
</tr>
<tr>
<td></td>
<td><code>/opt/langtools/lib/</code> '/opt/langtools/lib/pa20_64/'</td>
</tr>
<tr>
<td>IBM AIX</td>
<td><code>/lib/</code> '/usr/lib/' <code>/usr/lpp/</code> '/usr/ccs/lib/'</td>
</tr>
<tr>
<td></td>
<td><code>/usr/dt/lib/</code> '/tmp/'</td>
</tr>
<tr>
<td>SUN Solaris 2.x</td>
<td><code>/lib/</code> '/usr/lib/`</td>
</tr>
<tr>
<td></td>
<td><code>/usr/ccs/lib/</code></td>
</tr>
<tr>
<td>Linux x86</td>
<td><code>/lib/</code> '/usr/lib/`</td>
</tr>
<tr>
<td>Linux Alpha</td>
<td><code>/lib/</code> '/usr/lib/`</td>
</tr>
</tbody>
</table>

The values you enter in the TotalView preference should be space-separated lists of the prefixes and suffixes to be used.

After starting TotalView, you can change these lists by using the **When the file suffix matches** and **And the file path prefix does not match** preferences.
dlopen Options for Scalability

When a target process calls `dlopen()`, a `dlopen` event is generated in TotalView. Because handling `dlopen` events impacts startup time for dynamically linked executables, TotalView provides ways to configure `dlopen` for better performance and scalability in HPC computing environments:

- Filtering `dlopen` events to avoid stopping a process for each event.
- Handling `dlopen` events in parallel to reduce client/server communication overhead with MRNet enabled.

Filtering `dlopen` Events

Two state variables and their related command line options enable you to filter `dlopen` events to plant breakpoints in the `dlopened` libraries only when the process stops for some other reason.

`dlopen` event filtering is controlled by the settings on two state variables, `TV::dlopen_always_recalculate` and `TV::dlopen_recalculate_on_match`, and their related command line options `dlopen_always_recalculate` and `dlopen_recalculate_on_match`.

Three possible `dlopen` filtering modes are made possible by these variables: Slow, Medium and Fast.

In Fast mode, the process never stops for a `dlopen` event, not even "null" `dlopen` events. Using this option can result in significant performance gains, but may be impractical for some applications. In Medium mode, some libraries can be specified to always reevaluate their breakpoints, rather than all or none.

- **Slow Mode: Reloads libraries on every dlopen event**
  
  Option: `dlopen_always_recalculate==true`

  Reloads libraries on every `dlopen` event, retaining TotalView's traditional breakpoint reevaluation semantics. This mode is compatible with CUDA and is a good choice when your session has pending breakpoints. However, this mode does not perform or scale as well as the other modes, because it requires the TotalView client to handle every (non-null) `dlopen` event for every process.

  If performance is not the primary concern, or the application or runtime environment does not perform many `dlopen` events, then this may be a good choice.

  In this mode, when the target stops with a `dlopen` event, the server reports the event to the client, where the library list is reloaded and checked to see if any additional breakpoint locations need to be planted in the newly loaded libraries.

- **Medium Mode: Reports only libraries that match defined patterns on a dlopen event**

  Options:

  `dlopen_always_recalculate==false`

  `dlopen_recalculate_on_match=="glob-list"`
A glob-list 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 example:

dlopen_recalculate_on_match=="libcuda.so*:libmylib1*:libmylib2.so"

This mode strikes a balance between performance and enabling breakpoints to be planted in dlopened libraries.

In Medium mode, the target process stops on every dlopen event (just as in Slow mode), but the event is not reported to the client unless one of the newly loaded libraries matches the provided pattern.

This setting requires:

— Adding the names of any dlopened libraries to the TV::dlopen_recalculate_on_match list if you want breakpoints planted in the library when the library is loaded.

— Adding "libcuda.so*" to the match list if you are debugging CUDA; otherwise TotalView will miss CUDA kernel launch events.

• Fast Mode: Does not stop for dlopen events

Options:

dlopen_always_recalculate==false
dlopen_recalculate_on_match=="

This mode provides the best performance, disallowing planting breakpoints in dlopened libraries when the library is loaded. Breakpoints are planted in the dlopened libraries only when the process stops for some other reason; however, be aware with this option that an application may have executed past the point at which you want to start debugging inside the dlopened library.

Because the debugger does not plant the dlopen breakpoint in the process, the process never stops for a dlopen event, not even "null" dlopen events. While this mode may be impractical for some applications, the performance gains are significant.

Table 4 summarizes the pros and cons of each mode.

<table>
<thead>
<tr>
<th>Mode/Speed</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow</td>
<td>dlopen_always_recalculate==true</td>
</tr>
<tr>
<td>Pros:</td>
<td>Retains TotalView's traditional breakpoint reevaluation semantics.</td>
</tr>
<tr>
<td></td>
<td>Works best with pending breakpoints.</td>
</tr>
<tr>
<td></td>
<td>Compatible with CUDA.</td>
</tr>
<tr>
<td>Cons:</td>
<td>Does not perform or scale as well as the other modes because the TotalView client handles every (non-null) dlopen event for every process.</td>
</tr>
<tr>
<td>Medium</td>
<td>dlopen_always_recalculate==false dlopen_recalculate_on_match==&quot;glob-list&quot;</td>
</tr>
</tbody>
</table>
Handling dlopen Events in Parallel

As of 8.15, TotalView supports handling dlopened libraries in parallel, thus reducing client/server communication overhead by using MRNet to fetch library information.

To handle dlopened libraries in parallel, use the TV::dlopen_read_libraries_in_parallel and its related command line option -dlopen_read_libraries_in_parallel.

This sets the state variable to true. Placing this dset command in the tvdrc file ensures that all instances of TotalView launch with this option:

dset TV::dlopen_read_libraries_in_parallel true

To set this option on an individual instance of TotalView, use the command line option when you start TotalView:

totalview -dlopen_read_libraries_in_parallel

Table 4: dlopen Event Filtering Modes

<table>
<thead>
<tr>
<th>Mode/Speed</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pros:</td>
</tr>
<tr>
<td></td>
<td>• Performs better by filtering out dlopen events.</td>
</tr>
<tr>
<td></td>
<td>• Allows the TotalView client to process multiple dlopen events at a time.</td>
</tr>
<tr>
<td></td>
<td>• Compatible with CUDA.</td>
</tr>
<tr>
<td>Fast</td>
<td>dlopen_always_recalculate==false</td>
</tr>
<tr>
<td></td>
<td>dlopen_recalculate_on_match==&quot;&quot;</td>
</tr>
</tbody>
</table>

Pros:

• Performs best by never stopping the process at dlopen events.
• Allows the TotalView client to process multiple dlopen events at a time.

Cons:

• Breakpoints cannot be calculated when a particular library is loaded.
• Breaks CUDA support.
Known Limitations

Dynamic library support has the following known limitations:

- TotalView does not deal correctly with parallel programs that call `dlopen` on different libraries in different processes. TotalView requires that the processes have a uniform address space, including all shared libraries.

- TotalView does not yet fully support unloading libraries (using `dlclose`) and then reloading them at a different address using `dlopen`.
Remapping Keys

On the SunOS 5 keyboard, you may need to remap the page-up and page-down keys to the prior and next key-sym so that you can scroll TotalView windows with the page-up and page-down keys. To do so, add the following lines to your X Window System startup file:

```bash
# Remap F29/F35 to PgUp/PgDn
xmodmap -e 'keysym F29 = Prior'
xmodmap -e 'keysym F35 = Next'
```
Expression System

Depending on the target platform, TotalView supports:

- An interpreted expression system only
- Both an interpreted and a compiled expression system

Unless stated otherwise below, TotalView supports interpreted expressions only.

Expression System on IBM AIX-Power and Blue Gene/Q

On IBM AIX and Blue Gene/Q, TotalView supports compiled and interpreted expressions. TotalView also supports assembly language in expressions.

Some program functions called from the TotalView expression system on the Power architecture cannot have floating-point arguments that are passed by value. However, in functions with a variable number of arguments, floating-point arguments can be in the varying part of the argument list. For example, you can include floating-point arguments with calls to `printf`:

```c
  double d = 3.14159;
  printf("d = \%f\n", d);
```

On Blue Gene/Q, currently TotalView supports only statically allocated patch spaces linked into the base executable. The executable may be statically or dynamically linked. The static patch space must not reside in a shared library. See Allocating Static Patch Space in the TotalView User Guide.
This chapter describes the architectures TotalView supports, including:

- “AMD and Intel x86-64” on page 386
- “Power Architectures” on page 391
- “Intel IA-64” on page 398
- “Intel x86” on page 404 (Intel 80386, 80486 and Pentium processors)
- “Sun SPARC” on page 409
This section describes AMD's 64-bit processors and the Intel EM64T processors, including:

- “x86-64 General Registers” on page 386
- “x86-64 Floating-Point Registers” on page 387
- “x86-64 FPCR Register” on page 388
- “x86-64 FPSR Register” on page 389
- “x86-64 MXCSR Register” on page 390

The x86-64 can be programmed in either 32- or 64-bit mode. TotalView supports both. In 32-bit mode, the processor is identical to an x86, and the stack frame is identical to the x86. The information within this section describes 64-bit mode.

The AMD x86-64 processor supports the IEEE floating-point format.

x86-64 General Registers

TotalView displays the x86-64 general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>General registers</td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$rax</td>
</tr>
<tr>
<td>RDX</td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$rdx</td>
<td></td>
</tr>
<tr>
<td>RCX</td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$rcx</td>
<td></td>
</tr>
<tr>
<td>RBX</td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$rbx</td>
<td></td>
</tr>
<tr>
<td>RSI</td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$rsi</td>
<td></td>
</tr>
<tr>
<td>RDI</td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$rdi</td>
<td></td>
</tr>
<tr>
<td>RBP</td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$rbp</td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$rsp</td>
<td></td>
</tr>
<tr>
<td>R8-R15</td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$r8-$r15</td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>Selector registers</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$ra</td>
</tr>
<tr>
<td>SS</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$ss</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$ds</td>
<td></td>
</tr>
</tbody>
</table>
x86-64 Floating-Point Registers

TotalView displays the x86-64 floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0</td>
<td>ST(0)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st0</td>
</tr>
<tr>
<td>ST1</td>
<td>ST(1)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st1</td>
</tr>
<tr>
<td>ST2</td>
<td>ST(2)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st2</td>
</tr>
<tr>
<td>ST3</td>
<td>ST(3)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st3</td>
</tr>
<tr>
<td>ST4</td>
<td>ST(4)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st4</td>
</tr>
<tr>
<td>ST5</td>
<td>ST(5)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st5</td>
</tr>
<tr>
<td>ST6</td>
<td>ST(6)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st6</td>
</tr>
<tr>
<td>ST7</td>
<td>ST(7)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st7</td>
</tr>
<tr>
<td>FPCR</td>
<td>Floating-point control</td>
<td>$int</td>
<td>yes</td>
<td>no</td>
<td>$fpcr</td>
</tr>
<tr>
<td>FPSR</td>
<td>Floating-point status</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fpsr</td>
</tr>
<tr>
<td>FPTAG</td>
<td>Tag word</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fptag</td>
</tr>
<tr>
<td>FPOP</td>
<td>Floating-point operation</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fpop</td>
</tr>
<tr>
<td>FPI</td>
<td>Instruction address</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fpi</td>
</tr>
<tr>
<td>FPD</td>
<td>Data address</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fpd</td>
</tr>
</tbody>
</table>
NOTE >> The x86-64 has 16 128-bit registers that are used by SSE and SSE2 instructions. TotalView displays these as 32 64-bit registers. These registers can be used in the following ways: 16 bytes, 8 words, 2 longs, 4 floating point, 2 double, or a single 128-bit value. TotalView shows each of these hardware registers as two $long registers. To change the type, dive and then edit the type in the data window to be an array of the type you wish. For example, cast it to “$char[16]”, “$float[4]”, and so on.

### x86-64 FPCR Register

For your convenience, TotalView interprets the bit settings of the FPCR and FPSR registers.

You can edit the value of the FPCR and set it to any of the bit settings outlined in the next table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC=RN</td>
<td>0x0000</td>
<td>To nearest rounding mode</td>
</tr>
<tr>
<td>RC=R-</td>
<td>0x2000</td>
<td>Toward negative infinity rounding mode</td>
</tr>
<tr>
<td>RC=R+</td>
<td>0x4000</td>
<td>Toward positive infinity rounding mode</td>
</tr>
<tr>
<td>RC= RZ</td>
<td>0x6000</td>
<td>Toward zero rounding mode</td>
</tr>
</tbody>
</table>
Using the x86-64 FPCR Register

You can change the value of the FPCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPCR to include \( 0x0004 \) so that TotalView traps the “divide-by-zero” bit. The string displayed next to the FPCR register should now include \( EM=(ZM) \). Now, when your program divides by zero, it receives a \( SIGFPE \) signal, which you can catch with TotalView. See “Handling Signals” in Chapter 3 of the TotalView Users Guide for information on handling signals. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the \( EF=(ZE) \) bit in the FPSR.

x86-64 FPSR Register

The bit settings of the x86-64 FPSR register are outlined in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( PC=\text{SGL} )</td>
<td>0x0000</td>
<td>Single-precision rounding</td>
</tr>
<tr>
<td>( PC=\text{DBL} )</td>
<td>0x0080</td>
<td>Double-precision rounding</td>
</tr>
<tr>
<td>( PC=\text{EXT} )</td>
<td>0x00c0</td>
<td>Extended-precision rounding</td>
</tr>
<tr>
<td>( EM=\text{PM} )</td>
<td>0x0020</td>
<td>Precision exception enable</td>
</tr>
<tr>
<td>( EM=\text{UM} )</td>
<td>0x0010</td>
<td>Underflow exception enable</td>
</tr>
<tr>
<td>( EM=\text{OM} )</td>
<td>0x0008</td>
<td>Overflow exception enable</td>
</tr>
<tr>
<td>( EM=\text{ZM} )</td>
<td>0x0004</td>
<td>Zero-divide exception enable</td>
</tr>
<tr>
<td>( EM=\text{DM} )</td>
<td>0x0002</td>
<td>Denormalized operand exception enable</td>
</tr>
<tr>
<td>( EM=\text{IM} )</td>
<td>0x0001</td>
<td>Invalid operation exception enable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{TOP}=&lt;i&gt; )</td>
<td>0x3800</td>
<td>Register (&lt;i&gt;) is top of FPU stack</td>
</tr>
<tr>
<td>B</td>
<td>0x8000</td>
<td>FPU busy</td>
</tr>
<tr>
<td>C0</td>
<td>0x0100</td>
<td>Condition bit 0</td>
</tr>
<tr>
<td>C1</td>
<td>0x0200</td>
<td>Condition bit 1</td>
</tr>
<tr>
<td>C2</td>
<td>0x0400</td>
<td>Condition bit 2</td>
</tr>
<tr>
<td>C3</td>
<td>0x4000</td>
<td>Condition bit 3</td>
</tr>
<tr>
<td>ES</td>
<td>0x0080</td>
<td>Exception summary status</td>
</tr>
<tr>
<td>SF</td>
<td>0x0040</td>
<td>Stack fault</td>
</tr>
<tr>
<td>( EF=\text{PE} )</td>
<td>0x0020</td>
<td>Precision exception</td>
</tr>
</tbody>
</table>
x86-64 MXCSR Register

This register contains control and status information for the SSE registers. Some of the bits in this register are editable. You cannot dive in these values.

The bit settings of the x86-64 MXCSR register are outlined in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF=UE</td>
<td>0x0010</td>
<td>Underflow exception</td>
</tr>
<tr>
<td>EF=OE</td>
<td>0x0008</td>
<td>Overflow exception</td>
</tr>
<tr>
<td>EF=ZE</td>
<td>0x0004</td>
<td>Zero divide exception</td>
</tr>
<tr>
<td>EF=DE</td>
<td>0x0002</td>
<td>Denormalized operand exception</td>
</tr>
<tr>
<td>EF=IE</td>
<td>0x0001</td>
<td>Invalid operation exception</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FZ</td>
<td>0x8000</td>
<td>Flush to zero</td>
</tr>
<tr>
<td>RC=RN</td>
<td>0x0000</td>
<td>To nearest rounding mode</td>
</tr>
<tr>
<td>RC=R-</td>
<td>0x2000</td>
<td>Toward negative infinity rounding mode</td>
</tr>
<tr>
<td>RC=R+</td>
<td>0x4000</td>
<td>Toward positive infinity rounding mode</td>
</tr>
<tr>
<td>RC=RZ</td>
<td>0x6000</td>
<td>Toward zero rounding mode</td>
</tr>
<tr>
<td>EM=PM</td>
<td>0x1000</td>
<td>Precision mask</td>
</tr>
<tr>
<td>EM=UM</td>
<td>0x0800</td>
<td>Underflow mask</td>
</tr>
<tr>
<td>EM=OM</td>
<td>0x0400</td>
<td>Overflow mask</td>
</tr>
<tr>
<td>EM=ZM</td>
<td>0x0200</td>
<td>Divide-by-zero mask</td>
</tr>
<tr>
<td>EM=DM</td>
<td>0x0100</td>
<td>Denormal mask</td>
</tr>
<tr>
<td>EM=IM</td>
<td>0x0080</td>
<td>Invalid operation mask</td>
</tr>
<tr>
<td>DAZ</td>
<td>0x0040</td>
<td>Denormals are zeros</td>
</tr>
<tr>
<td>EF=PE</td>
<td>0x0020</td>
<td>Precision flag</td>
</tr>
<tr>
<td>EF=UE</td>
<td>0x0010</td>
<td>Underflow flag</td>
</tr>
<tr>
<td>EF=OE</td>
<td>0x0008</td>
<td>Overflow flag</td>
</tr>
<tr>
<td>EF=ZE</td>
<td>0x0004</td>
<td>Divide-by-zero flag</td>
</tr>
<tr>
<td>EF=DE</td>
<td>0x0002</td>
<td>Denormal flag</td>
</tr>
<tr>
<td>EF=IE</td>
<td>0x0001</td>
<td>Invalid operation flag</td>
</tr>
</tbody>
</table>
Power Architectures

This section contains the following information:

- Power General Registers
- Blue Gene Power Registers
- Power MSR Register
- Power Floating-Point Registers
- Blue Gene/Q QPX Floating-Point Registers
- Power FPSCR Register
- Using the Power FPSCR Register

**NOTE >>** The Power architecture supports the IEEE floating-point format.

### Power General Registers

TotalView displays Power general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>General register 0</td>
<td>$int/$long</td>
<td>yes</td>
<td>yes</td>
<td>$r0</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer</td>
<td>$int/$long</td>
<td>yes</td>
<td>yes</td>
<td>$sp</td>
</tr>
<tr>
<td>RTOC</td>
<td>TOC pointer</td>
<td>$int/$long</td>
<td>yes</td>
<td>yes</td>
<td>$rtoc</td>
</tr>
<tr>
<td>R3 - R31</td>
<td>General registers 3 - 31</td>
<td>$int/$long</td>
<td>yes</td>
<td>yes</td>
<td>$r3 - $r31</td>
</tr>
<tr>
<td>INUM</td>
<td></td>
<td>$int/$long</td>
<td>yes</td>
<td>no</td>
<td>$inum</td>
</tr>
<tr>
<td>PC</td>
<td>Program counter</td>
<td>$code[]</td>
<td>no</td>
<td>yes</td>
<td>$pc</td>
</tr>
<tr>
<td>SRR1</td>
<td>Machine status save/restore</td>
<td>$int/$long</td>
<td>yes</td>
<td>no</td>
<td>$srr1</td>
</tr>
<tr>
<td>LR</td>
<td>Link register</td>
<td>$code[]</td>
<td>yes</td>
<td>no</td>
<td>$lr</td>
</tr>
<tr>
<td>CTR</td>
<td>Counter register</td>
<td>$int/$long</td>
<td>yes</td>
<td>no</td>
<td>$ctr</td>
</tr>
<tr>
<td>CR</td>
<td>Condition register (see below)</td>
<td>$int/$long</td>
<td>yes</td>
<td>no</td>
<td>$cr</td>
</tr>
</tbody>
</table>
CR Register

TotalView writes information for each of the eight condition sets, appending a a >, <, or = symbol. For example, if the summary overflow (0x1) bit is set, TotalView might display the following:

0x22424444 (574768196) (0=,1=,2>,3=,4>,5>,6>,7>)

XER Register

Depending upon what was set, TotalView can display up to five kinds of information, as follows:

STD:0x%02x
The string terminator character (bits 25-31)

SL:%d
The string length field (bits 16-23)

S0
Displayed if the summary overflow bit is set (bit 0)

OV
Displayed if the overflow bit is set (bit 1)

CA
Displayed if the carry bit is set (bit 2)

For example:
Blue Gene Power Registers

TotalView displays Blue Gene Power registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each Blue Gene register, and the actions you can take with each register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>General register 0</td>
<td>$int/$long</td>
<td>yes</td>
<td>yes</td>
<td>$r0</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer</td>
<td>$int/$long</td>
<td>yes</td>
<td>yes</td>
<td>$sp</td>
</tr>
<tr>
<td>RTOC</td>
<td>TOC pointer</td>
<td>$int/$long</td>
<td>yes</td>
<td>yes</td>
<td>$rtoc</td>
</tr>
<tr>
<td>R3 - R31</td>
<td>General registers 3 - 31</td>
<td>$int/$long</td>
<td>yes</td>
<td>yes</td>
<td>$r3 - $r31</td>
</tr>
<tr>
<td>LR</td>
<td>Link register</td>
<td>$code[]</td>
<td>yes</td>
<td>no</td>
<td>$lr</td>
</tr>
<tr>
<td>CR</td>
<td>Condition register (see below)</td>
<td>$int/$long</td>
<td>yes</td>
<td>no</td>
<td>$cr</td>
</tr>
<tr>
<td>XER</td>
<td>Integer exception register (see below)</td>
<td>$int/$long</td>
<td>yes</td>
<td>no</td>
<td>$xer</td>
</tr>
<tr>
<td>CTR</td>
<td>Counter register</td>
<td>$int/$long</td>
<td>yes</td>
<td>no</td>
<td>$ctr</td>
</tr>
<tr>
<td>IAR</td>
<td>Program counter</td>
<td>$code[]</td>
<td>no</td>
<td>yes</td>
<td>$pc</td>
</tr>
<tr>
<td>MSR</td>
<td>Machine state register</td>
<td>$int/$long</td>
<td>yes</td>
<td>no</td>
<td>$msr</td>
</tr>
<tr>
<td>DEAR</td>
<td>Data address register</td>
<td>$int/$long</td>
<td>yes</td>
<td>no</td>
<td>$dar</td>
</tr>
<tr>
<td>ESR</td>
<td>Exception status register</td>
<td>$int/$long</td>
<td>no</td>
<td>no</td>
<td>$esr</td>
</tr>
</tbody>
</table>
Blue Gene/Q QPX Floating-Point Registers

TotalView displays the Blue Gene/Q Quad Processing eXtension to the Power ISA (QPX) registers in the Stack Frame Pane of the Process Window. The architecture provides for 32 256-bit registers that can be used as four doubles, eight floats, four 64-bit integers, or eight 32-bit integers. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0 - F31</td>
<td>QPX floating-point reg-</td>
<td>$qpx_reg</td>
<td>yes</td>
<td>yes</td>
<td>$q0 - $q31</td>
</tr>
<tr>
<td></td>
<td>isters 0 - 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPSCR</td>
<td>Floating-point status</td>
<td>$long</td>
<td>yes</td>
<td>no</td>
<td>$fpscr</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The data type $qpx_reg is a TotalView predefined type that is defined as follows:

```c
union $qpx_reg {
    $double q4_double[4];
    $float q8_float[8];
    $int64 q4_int64[4];
    $int32 q8_int32[8];
};
```

The traditional Book 1 Power PC floating point instructions that operate on the FPR register set operate on slot 0 of the corresponding QPX register. Therefore, the Stack Frame Pane of the Process Window shows the double contained in slot 0 (the q4_double[0] field of the $qpx_reg data type) for each QPX register. Dive on a QPX register to open a Data Pane displaying the full contents of the register.

Power MSR Register

For your convenience, TotalView interprets the bit settings of the Power MSR register. You can edit the value of the MSR and set it to any of the bit settings outlined in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000000000000000</td>
<td>SF</td>
<td>Sixty-four bit mode</td>
</tr>
<tr>
<td>0x0000000000040000</td>
<td>POW</td>
<td>Power management enable</td>
</tr>
<tr>
<td>0x0000000000020000</td>
<td>TGPR</td>
<td>Temporary GPR mapping</td>
</tr>
<tr>
<td>0x0000000000010000</td>
<td>ILE</td>
<td>Exception little-endian mode</td>
</tr>
<tr>
<td>0x0000000000008000</td>
<td>EE</td>
<td>External interrupt enable</td>
</tr>
<tr>
<td>0x0000000000004000</td>
<td>PR</td>
<td>Privilege level</td>
</tr>
<tr>
<td>0x0000000000002000</td>
<td>FP</td>
<td>Floating-point available</td>
</tr>
<tr>
<td>0x0000000000001000</td>
<td>ME</td>
<td>Machine check enable</td>
</tr>
</tbody>
</table>
TotalView displays the Power floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0 - F31</td>
<td>Floating-point registers 0 - 31</td>
<td>$double</td>
<td>yes</td>
<td>yes</td>
<td>$f0 - $f31</td>
</tr>
<tr>
<td>FPSCR</td>
<td>Floating-point status register</td>
<td>$int</td>
<td>yes</td>
<td>no</td>
<td>$fpscr</td>
</tr>
<tr>
<td>FPSCR2</td>
<td>Floating-point status register 2</td>
<td>$int</td>
<td>yes</td>
<td>no</td>
<td>$fpscr2</td>
</tr>
</tbody>
</table>

**Power FPSCR Register**

For your convenience, TotalView interprets the bit settings of the Power FPSCR register. You can edit the value of the FPSCR and set it to any of the bit settings outlined in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80000000</td>
<td>FX</td>
<td>Floating-point exception summary</td>
</tr>
<tr>
<td>0x40000000</td>
<td>FEX</td>
<td>Floating-point enabled exception summary</td>
</tr>
<tr>
<td>0x20000000</td>
<td>VX</td>
<td>Floating-point invalid operation exception summary</td>
</tr>
<tr>
<td>0x10000000</td>
<td>OX</td>
<td>Floating-point overflow exception</td>
</tr>
<tr>
<td>0x08000000</td>
<td>UX</td>
<td>Floating-point underflow exception</td>
</tr>
<tr>
<td>Value</td>
<td>Bit Setting</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>0x04000000</td>
<td>ZX</td>
<td>Floating-point zero divide exception</td>
</tr>
<tr>
<td>0x02000000</td>
<td>XX</td>
<td>Floating-point inexact exception</td>
</tr>
<tr>
<td>0x01000000</td>
<td>VXSNAN</td>
<td>Floating-point invalid operation exception for SNaN</td>
</tr>
<tr>
<td>0x00800000</td>
<td>VXISI</td>
<td>Floating-point invalid operation exception: ¥ - ¥, or infinity-infinity</td>
</tr>
<tr>
<td>0x00400000</td>
<td>VXIDi</td>
<td>Floating-point invalid operation exception: ¥ / ¥, or infinity divided by infinity</td>
</tr>
<tr>
<td>0x00200000</td>
<td>VXZDZ</td>
<td>Floating-point invalid operation exception: 0 / 0</td>
</tr>
<tr>
<td>0x00100000</td>
<td>VXIMZ</td>
<td>Floating-point invalid operation exception: ¥ * ¥, or infinity times infinity</td>
</tr>
<tr>
<td>0x00080000</td>
<td>VXVC</td>
<td>Floating-point invalid operation exception: invalid compare</td>
</tr>
<tr>
<td>0x00040000</td>
<td>FR</td>
<td>Floating-point fraction rounded</td>
</tr>
<tr>
<td>0x00020000</td>
<td>FI</td>
<td>Floating-point fraction inexact</td>
</tr>
<tr>
<td>0x00010000</td>
<td>FPRF=(C)</td>
<td>Floating-point result class descriptor</td>
</tr>
<tr>
<td>0x00008000</td>
<td>FPRF=(L)</td>
<td>Floating-point less than or negative</td>
</tr>
<tr>
<td>0x00004000</td>
<td>FPRF=(G)</td>
<td>Floating-point greater than or positive</td>
</tr>
<tr>
<td>0x00002000</td>
<td>FPRF=(E)</td>
<td>Floating-point equal or zero</td>
</tr>
<tr>
<td>0x00001000</td>
<td>FPRF=(U)</td>
<td>Floating-point unordered or NaN</td>
</tr>
<tr>
<td>0x00011000</td>
<td>FPRF=(QNAN)</td>
<td>Quiet NaN; alias for FPRF=(C+U)</td>
</tr>
<tr>
<td>0x00009000</td>
<td>FPRF=(-INF)</td>
<td>-Infinity; alias for FPRF=(L+U)</td>
</tr>
<tr>
<td>0x00008000</td>
<td>FPRF=(-NORM)</td>
<td>-Normalized number; alias for FPRF=(L)</td>
</tr>
<tr>
<td>0x00018000</td>
<td>FPRF=(-DENORM)</td>
<td>-Denormalized number; alias for FPRF=(C+L)</td>
</tr>
<tr>
<td>0x00012000</td>
<td>FPRF=(-ZERO)</td>
<td>-Zero; alias for FPRF=(C+E)</td>
</tr>
<tr>
<td>0x00002000</td>
<td>FPRF=(+ZERO)</td>
<td>+Zero; alias for FPRF=(E)</td>
</tr>
<tr>
<td>0x00014000</td>
<td>FPRF=(+DENORM)</td>
<td>+Denormalized number; alias for FPRF=(C+G)</td>
</tr>
<tr>
<td>0x00004000</td>
<td>FPRF=(+NORM)</td>
<td>+Normalized number; alias for FPRF=(G)</td>
</tr>
<tr>
<td>0x00005000</td>
<td>FPRF=(+INF)</td>
<td>+Infinity; alias for FPRF=(G+U)</td>
</tr>
<tr>
<td>0x00000400</td>
<td>VXSOFT</td>
<td>Floating-point invalid operation exception: software request</td>
</tr>
<tr>
<td>0x00000200</td>
<td>VXSQRT</td>
<td>Floating-point invalid operation exception: square root</td>
</tr>
<tr>
<td>0x00000100</td>
<td>VXCVI</td>
<td>Floating-point invalid operation exception: invalid integer convert</td>
</tr>
<tr>
<td>0x00000080</td>
<td>VE</td>
<td>Floating-point invalid operation exception enable</td>
</tr>
</tbody>
</table>
Using the Power FPSCR Register

On AIX, if you compile your program to catch floating-point exceptions (IBM compiler -qflttrap option), you can change the value of the FPSCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPSCR to include `0x10` so that TotalView traps the “divide by zero” exception. The string displayed next to the FPSR register should now include `ZE`. Now, when your program divides by zero, it receives a SIGTRAP signal, which will be caught by TotalView. See “Handling Signals” in Chapter 3 of the TotalView Users Guide for more information. If you did not set the bit for trapping divide by zero or you did not compile to catch floating-point exceptions, your program would not stop and the processor would set the ZX bit.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000040</td>
<td>OE</td>
<td>Floating-point overflow exception enable</td>
</tr>
<tr>
<td>0x00000020</td>
<td>UE</td>
<td>Floating-point underflow exception enable</td>
</tr>
<tr>
<td>0x00000010</td>
<td>ZE</td>
<td>Floating-point zero divide exception enable</td>
</tr>
<tr>
<td>0x00000008</td>
<td>XE</td>
<td>Floating-point inexact exception enable</td>
</tr>
<tr>
<td>0x00000004</td>
<td>NI</td>
<td>Floating-point non-IEEE mode enable</td>
</tr>
<tr>
<td>0x00000000</td>
<td>RN=NEAR</td>
<td>Round to nearest</td>
</tr>
<tr>
<td>0x00000001</td>
<td>RN=ZERO</td>
<td>Round toward zero</td>
</tr>
<tr>
<td>0x00000002</td>
<td>RN=PINF</td>
<td>Round toward +infinity</td>
</tr>
<tr>
<td>0x00000003</td>
<td>RN=NINF</td>
<td>Round toward -infinity</td>
</tr>
</tbody>
</table>

Using the Power FPSCR Register

On AIX, if you compile your program to catch floating-point exceptions (IBM compiler -qflttrap option), you can change the value of the FPSCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPSCR to include `0x10` so that TotalView traps the “divide by zero” exception. The string displayed next to the FPSR register should now include `ZE`. Now, when your program divides by zero, it receives a SIGTRAP signal, which will be caught by TotalView. See “Handling Signals” in Chapter 3 of the TotalView Users Guide for more information. If you did not set the bit for trapping divide by zero or you did not compile to catch floating-point exceptions, your program would not stop and the processor would set the ZX bit.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000040</td>
<td>OE</td>
<td>Floating-point overflow exception enable</td>
</tr>
<tr>
<td>0x00000020</td>
<td>UE</td>
<td>Floating-point underflow exception enable</td>
</tr>
<tr>
<td>0x00000010</td>
<td>ZE</td>
<td>Floating-point zero divide exception enable</td>
</tr>
<tr>
<td>0x00000008</td>
<td>XE</td>
<td>Floating-point inexact exception enable</td>
</tr>
<tr>
<td>0x00000004</td>
<td>NI</td>
<td>Floating-point non-IEEE mode enable</td>
</tr>
<tr>
<td>0x00000000</td>
<td>RN=NEAR</td>
<td>Round to nearest</td>
</tr>
<tr>
<td>0x00000001</td>
<td>RN=ZERO</td>
<td>Round toward zero</td>
</tr>
<tr>
<td>0x00000002</td>
<td>RN=PINF</td>
<td>Round toward +infinity</td>
</tr>
<tr>
<td>0x00000003</td>
<td>RN=NINF</td>
<td>Round toward -infinity</td>
</tr>
</tbody>
</table>
Intel IA-64

This section contains the following information:

- Intel IA-64 General Registers
- “IA-64 Processor Status Register Fields (PSR)” on page 400
- “Current Frame Marker Register Fields (CFM)” on page 401
- “Register Stack Configuration Register Fields (RSC)” on page 401
- “Previous Function State Register Fields (PFS)” on page 402
- “Floating Point Registers” on page 402
- “Floating Point Status Register Fields” on page 402

The Cray XT3 front end runs on this chip.

Intel IA-64 General Registers

TotalView displays the IA-64 general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>in expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>register 0</td>
<td>$long</td>
<td>N</td>
<td>Y</td>
<td>$r0</td>
</tr>
<tr>
<td>r1</td>
<td>global pointer</td>
<td>$long</td>
<td>N</td>
<td>Y</td>
<td>$r1</td>
</tr>
<tr>
<td>r2-r31</td>
<td>static general registers</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$r2-$r31</td>
</tr>
<tr>
<td>r32-r127</td>
<td>stacked general registers (all may not be valid)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$r32-$r127</td>
</tr>
<tr>
<td>b0-b7</td>
<td>branch registers</td>
<td>$code[]</td>
<td>Y</td>
<td>Y</td>
<td>$b0-$b7</td>
</tr>
<tr>
<td>ip</td>
<td>instruction pointer</td>
<td>$code[]</td>
<td>N</td>
<td>Y</td>
<td>$ip</td>
</tr>
<tr>
<td>cfm</td>
<td>current frame marker</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$cfm</td>
</tr>
<tr>
<td>psr</td>
<td>processor status register</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$psr</td>
</tr>
</tbody>
</table>

NOTE >> The descriptions in this section are taken (almost verbatim) from the “Intel Itanium Architecture Software Developer’s Manual. Volume 1: Application Architecture”. This was revision 2.0, printed in December 2001.
<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>in expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsc</td>
<td>register stack configuration register (AR 16)</td>
<td>$long</td>
<td>Y(N on HP-UX)</td>
<td>Y</td>
<td>$rsc</td>
</tr>
<tr>
<td>bsp</td>
<td>rse backing store pointer (AR 17)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$bsp</td>
</tr>
<tr>
<td>bspstore</td>
<td>rse backing store pointer for memory stores (AR 18)</td>
<td>$long</td>
<td>N</td>
<td>Y</td>
<td>$bspstore</td>
</tr>
<tr>
<td>rnat</td>
<td>rse NAT collection register (AR 19)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$rnat</td>
</tr>
<tr>
<td>ccv</td>
<td>compare and exchange value register (AR 32)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$ccv</td>
</tr>
<tr>
<td>unat</td>
<td>user NAT collection register (AR 36)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$unat</td>
</tr>
<tr>
<td>fpsr</td>
<td>floating point status register (AR 40)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$fpsr</td>
</tr>
<tr>
<td>pfs</td>
<td>previous function state (AR 64)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$pfs</td>
</tr>
<tr>
<td>lc</td>
<td>loop count register (AR 65)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$lc</td>
</tr>
<tr>
<td>ec</td>
<td>epilog count register (AR 66)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$ec</td>
</tr>
<tr>
<td>pr</td>
<td>predication registers (packed)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$pr</td>
</tr>
<tr>
<td>nat</td>
<td>nat registers (packed)</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$nat</td>
</tr>
</tbody>
</table>

**NOTE >>** All general registers r32-r127 may not be valid in a given stack frame.
IA-64 Processor Status Register Fields (PSR)

These fields control memory access alignment, byte-ordering, and user-configured performance monitors. It also records the modification state of floating-point registers.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>be</td>
<td>big-endian enable</td>
</tr>
<tr>
<td>2</td>
<td>up</td>
<td>user performance monitor enable</td>
</tr>
<tr>
<td>3</td>
<td>ac</td>
<td>alignment check</td>
</tr>
<tr>
<td>4</td>
<td>mfl</td>
<td>lower (f2-f31) floating point registers written</td>
</tr>
<tr>
<td>5</td>
<td>mfh</td>
<td>upper (f32-f127) floating point registers written</td>
</tr>
<tr>
<td>13</td>
<td>ic</td>
<td>interruption collection</td>
</tr>
<tr>
<td>14</td>
<td>i</td>
<td>interrupt bit</td>
</tr>
<tr>
<td>15</td>
<td>pk</td>
<td>protection key enable</td>
</tr>
<tr>
<td>17</td>
<td>dt</td>
<td>data address translation</td>
</tr>
<tr>
<td>18</td>
<td>dfl</td>
<td>disabled lower floating point register set</td>
</tr>
<tr>
<td>19</td>
<td>dfh</td>
<td>disabled upper floating point register set</td>
</tr>
<tr>
<td>20</td>
<td>sp</td>
<td>secure performance monitors</td>
</tr>
<tr>
<td>21</td>
<td>pp</td>
<td>privileged performance monitor enable</td>
</tr>
<tr>
<td>22</td>
<td>di</td>
<td>disable instruction set transition</td>
</tr>
<tr>
<td>23</td>
<td>si</td>
<td>secure interval timer</td>
</tr>
<tr>
<td>24</td>
<td>db</td>
<td>debug breakpoint fault</td>
</tr>
<tr>
<td>25</td>
<td>lp</td>
<td>lower privilege transfer trap</td>
</tr>
<tr>
<td>26</td>
<td>tb</td>
<td>taken branch trap</td>
</tr>
<tr>
<td>27</td>
<td>rt</td>
<td>register stack translation</td>
</tr>
<tr>
<td>33:32</td>
<td>cpl</td>
<td>current privilege level</td>
</tr>
<tr>
<td>34</td>
<td>is</td>
<td>instruction set</td>
</tr>
<tr>
<td>35</td>
<td>mc</td>
<td>machine check abort mask</td>
</tr>
<tr>
<td>36</td>
<td>it</td>
<td>instruction address translation</td>
</tr>
<tr>
<td>37</td>
<td>id</td>
<td>instruction debug fault disable</td>
</tr>
<tr>
<td>38</td>
<td>da</td>
<td>disable data access and dirty-bit faults</td>
</tr>
<tr>
<td>39</td>
<td>dd</td>
<td>data debug fault disable</td>
</tr>
<tr>
<td>40</td>
<td>ss</td>
<td>single step enable</td>
</tr>
<tr>
<td>42:41</td>
<td>ri</td>
<td>restart instruction</td>
</tr>
</tbody>
</table>
Current Frame Marker Register Fields (CFM)

Each general register stack frame is associated with a frame marker. The frame maker describes the state of the general register stack. The Current Frame Marker (CFM) holds the state of the current stack frame.

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>6:0</td>
<td>sof</td>
<td>Size of frame</td>
</tr>
<tr>
<td>13:7</td>
<td>sol</td>
<td>Size of locals portion of stack frame</td>
</tr>
<tr>
<td>17:14</td>
<td>sor</td>
<td>Size of rotating portion of stack frame (number of rotating registers is sor*8)</td>
</tr>
<tr>
<td>24:18</td>
<td>rrb.gr</td>
<td>Register rename base for general registers</td>
</tr>
<tr>
<td>31:25</td>
<td>rrb.fr</td>
<td>Register rename base for float registers</td>
</tr>
<tr>
<td>37:32</td>
<td>rrb.pr</td>
<td>Register rename base for predicate registers</td>
</tr>
</tbody>
</table>

Register Stack Configuration Register Fields (RSC)

The Register Stack Configuration (RSC) Register is a 64-bit register used to control the operation of the Register Stack Engine (RSE).

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:0 00</td>
<td>mode</td>
<td>enforced lazy</td>
</tr>
<tr>
<td>1:0 01</td>
<td></td>
<td>load intensive</td>
</tr>
<tr>
<td>1:0 10</td>
<td></td>
<td>store intensive</td>
</tr>
<tr>
<td>1:0 11</td>
<td></td>
<td>eager</td>
</tr>
<tr>
<td>3:2</td>
<td>pl</td>
<td>RSE privilege level</td>
</tr>
<tr>
<td>4</td>
<td>be</td>
<td>RSE endian mode (0=little endian, 1=big endian)</td>
</tr>
<tr>
<td>29:16</td>
<td>loadrs</td>
<td>RSE load distance to tear point</td>
</tr>
</tbody>
</table>
Previous Function State Register Fields (PFS)

The Previous Function State register (PFS) contains multiple fields: Previous Frame Marker (pfm), Previous Epilog Count (pec), and Previous Privilege Level (ppl). These values are copied automatically on a call from the CFM register, Epilog Count Register (EC), and PSR.cpl (Current Privilege Level in the Processor Status Register) to accelerate procedure calling.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>37:0</td>
<td>pfm</td>
<td>previous frame marker</td>
</tr>
<tr>
<td>57:52</td>
<td>pec</td>
<td>previous epilog count</td>
</tr>
<tr>
<td>63:62</td>
<td>ppl</td>
<td>previous privilege level</td>
</tr>
</tbody>
</table>

Floating Point Registers

The IA-64 contains 128 floating-point registers. The first two are read only.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>float register 0</td>
<td>$long</td>
<td>N</td>
<td>Y</td>
<td>$f0</td>
</tr>
<tr>
<td>f1</td>
<td>float register 1</td>
<td>$long</td>
<td>N</td>
<td>Y</td>
<td>$f1</td>
</tr>
<tr>
<td>f2-f31</td>
<td>lower float registers</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$f2-$f31</td>
</tr>
<tr>
<td>f32-f127</td>
<td>upper float registers</td>
<td>$long</td>
<td>Y</td>
<td>Y</td>
<td>$f32-$f127</td>
</tr>
</tbody>
</table>

Floating Point Status Register Fields

The Floating-Point Status Register (FPSR) contains the dynamic control and status information for floating-point operations. There is one main set of control and status information and three alternate sets.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>traps.vd</td>
<td>0</td>
<td>Invalid Operation Floating-Point Exception fault disabled</td>
</tr>
<tr>
<td>traps.dd</td>
<td>1</td>
<td>Denormal/Unnormal Operating Floating-Point Exception fault disabled</td>
</tr>
<tr>
<td>traps.zd</td>
<td>2</td>
<td>Zero Divide Floating-Point Exception trap disabled</td>
</tr>
<tr>
<td>traps.od</td>
<td>3</td>
<td>Overflow Floating-Point Exception trap disabled</td>
</tr>
<tr>
<td>traps.ud</td>
<td>4</td>
<td>Underflow Floating-Point Exception trap disabled</td>
</tr>
<tr>
<td>traps.id</td>
<td>5</td>
<td>Inexact Floating-Point Exception trap disabled</td>
</tr>
</tbody>
</table>
Here is a description of the FPSR status field descriptions.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sfo</td>
<td>18:6</td>
<td>main status field</td>
</tr>
<tr>
<td>sf1</td>
<td>31:19</td>
<td>alternate status field 1</td>
</tr>
<tr>
<td>sf2</td>
<td>44:32</td>
<td>alternate status field 2</td>
</tr>
<tr>
<td>sf3</td>
<td>57:45</td>
<td>alternate status field 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ftz</td>
<td>flush-to-zero mode</td>
</tr>
<tr>
<td>1</td>
<td>wre</td>
<td>widest range exponent</td>
</tr>
<tr>
<td>3:2</td>
<td>pc</td>
<td>precision control</td>
</tr>
<tr>
<td>5:4</td>
<td>rc</td>
<td>rounding control</td>
</tr>
<tr>
<td>6</td>
<td>td</td>
<td>traps disabled</td>
</tr>
<tr>
<td>7</td>
<td>v</td>
<td>invalid operation</td>
</tr>
<tr>
<td>8</td>
<td>d</td>
<td>denormal/unnorm operand</td>
</tr>
<tr>
<td>9</td>
<td>z</td>
<td>zero divide</td>
</tr>
<tr>
<td>10</td>
<td>o</td>
<td>overflow</td>
</tr>
<tr>
<td>11</td>
<td>u</td>
<td>underflow</td>
</tr>
<tr>
<td>12</td>
<td>i</td>
<td>inexact</td>
</tr>
</tbody>
</table>
Intel x86

This section contains the following information:

- “Intel x86 General Registers” on page 404
- “Intel x86 Floating-Point Registers” on page 405
- “Intel x86 FPCR Register” on page 406
- “Intel x86 FPSR Register” on page 407
- “Intel x86 MXCSR Register” on page 407

NOTE >> The Intel x86 processor supports the IEEE floating-point format.

Intel x86 General Registers

TotalView displays the Intel x86 general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>General registers</td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$eax</td>
</tr>
<tr>
<td>ECX</td>
<td></td>
<td>$long</td>
<td>yes</td>
<td></td>
<td>$ecx</td>
</tr>
<tr>
<td>EDX</td>
<td></td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$edx</td>
</tr>
<tr>
<td>EBX</td>
<td></td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$ebx</td>
</tr>
<tr>
<td>EBP</td>
<td></td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$ebp</td>
</tr>
<tr>
<td>ESP</td>
<td></td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$esp</td>
</tr>
<tr>
<td>ESI</td>
<td></td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$esi</td>
</tr>
<tr>
<td>EDI</td>
<td></td>
<td>$long</td>
<td>yes</td>
<td>yes</td>
<td>$edi</td>
</tr>
<tr>
<td>CS</td>
<td>Selector registers</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$cs</td>
</tr>
<tr>
<td>SS</td>
<td></td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$ss</td>
</tr>
<tr>
<td>DS</td>
<td></td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$ds</td>
</tr>
<tr>
<td>ES</td>
<td></td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$es</td>
</tr>
<tr>
<td>FS</td>
<td></td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fs</td>
</tr>
<tr>
<td>GS</td>
<td></td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$gs</td>
</tr>
<tr>
<td>EFLAGS</td>
<td></td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$eflags</td>
</tr>
</tbody>
</table>
Intel x86 Floating-Point Registers

TotalView displays the x86 floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIP</td>
<td>Instruction pointer</td>
<td>$code[]</td>
<td>no</td>
<td>yes</td>
<td>$eip</td>
</tr>
<tr>
<td>FAULT</td>
<td>$long</td>
<td>no</td>
<td>no</td>
<td></td>
<td>$fault</td>
</tr>
<tr>
<td>TEMP</td>
<td>$long</td>
<td>no</td>
<td>no</td>
<td></td>
<td>$temp</td>
</tr>
<tr>
<td>INUM</td>
<td>$long</td>
<td>no</td>
<td>no</td>
<td></td>
<td>$inum</td>
</tr>
<tr>
<td>ECODE</td>
<td>$long</td>
<td>no</td>
<td>no</td>
<td></td>
<td>$ecode</td>
</tr>
<tr>
<td>ST0</td>
<td>ST(0)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st0</td>
</tr>
<tr>
<td>ST1</td>
<td>ST(1)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st1</td>
</tr>
<tr>
<td>ST2</td>
<td>ST(2)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st2</td>
</tr>
<tr>
<td>ST3</td>
<td>ST(3)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st3</td>
</tr>
<tr>
<td>ST4</td>
<td>ST(4)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st4</td>
</tr>
<tr>
<td>ST5</td>
<td>ST(5)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st5</td>
</tr>
<tr>
<td>ST6</td>
<td>ST(6)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st6</td>
</tr>
<tr>
<td>ST7</td>
<td>ST(7)</td>
<td>$extended</td>
<td>yes</td>
<td>yes</td>
<td>$st7</td>
</tr>
<tr>
<td>FPCR</td>
<td>Floating-point control register</td>
<td>$int</td>
<td>yes</td>
<td>no</td>
<td>$fpacr</td>
</tr>
<tr>
<td>FPSR</td>
<td>Floating-point status register</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fpsr</td>
</tr>
<tr>
<td>FPTAG</td>
<td>Tag word</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fptag</td>
</tr>
<tr>
<td>FPIOFF</td>
<td>Instruction offset</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fpioff</td>
</tr>
<tr>
<td>FPISEL</td>
<td>Instruction selector</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fpiisel</td>
</tr>
<tr>
<td>FPDOFF</td>
<td>Data offset</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fpdoff</td>
</tr>
<tr>
<td>FPDSL</td>
<td>Data selector</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$fpdsl</td>
</tr>
<tr>
<td>MXCSR</td>
<td>SSE status and control</td>
<td>$int</td>
<td>yes</td>
<td>no</td>
<td>$mxcsv</td>
</tr>
<tr>
<td>MXCSR-MASK</td>
<td>MXCSR mask</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$mxcsr_mask</td>
</tr>
</tbody>
</table>
The Pentium III and 4 have 8 128-bit registers that are used by SSE and SSE2 instructions. TotalView displays these as 16 64-bit registers. These registers can be used in the following ways: 16 bytes, 8 words, 2 long longs, 4 floating point, 2 double, or a single 128-bit value. TotalView shows each of these hardware registers as two $long long registers. To change the type, dive and then edit the type in the data window to be an array of the type you wish. For example, cast it to "$char[16]", "$float[4]", and so on.

### Intel x86 FPCR Register

For your convenience, TotalView interprets the bit settings of the FPCR and FPSR registers.

You can edit the value of the FPCR and set it to any of the bit settings outlined in the next table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC=RN</td>
<td>0x0000</td>
<td>To nearest rounding mode</td>
</tr>
<tr>
<td>RC=R-</td>
<td>0x2000</td>
<td>Toward negative infinity rounding mode</td>
</tr>
<tr>
<td>RC=R+</td>
<td>0x4000</td>
<td>Toward positive infinity rounding mode</td>
</tr>
<tr>
<td>RC=RZ</td>
<td>0x6000</td>
<td>Toward zero rounding mode</td>
</tr>
<tr>
<td>PC=SGL</td>
<td>0x0000</td>
<td>Single-precision rounding</td>
</tr>
<tr>
<td>PC=DBL</td>
<td>0x0080</td>
<td>Double-precision rounding</td>
</tr>
<tr>
<td>PC=EXT</td>
<td>0x00c0</td>
<td>Extended-precision rounding</td>
</tr>
<tr>
<td>EM=PM</td>
<td>0x0020</td>
<td>Precision exception enable</td>
</tr>
<tr>
<td>EM=UM</td>
<td>0x0010</td>
<td>Underflow exception enable</td>
</tr>
<tr>
<td>EM=OM</td>
<td>0x0008</td>
<td>Overflow exception enable</td>
</tr>
<tr>
<td>EM=ZM</td>
<td>0x0004</td>
<td>Zero-divide exception enable</td>
</tr>
<tr>
<td>EM=DM</td>
<td>0x0002</td>
<td>Denormalized operand exception enable</td>
</tr>
<tr>
<td>EM=IM</td>
<td>0x0001</td>
<td>Invalid operation exception enable</td>
</tr>
</tbody>
</table>
Using the Intel x86 FPCR Register

You can change the value of the FPCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPCR to include 0x0004 so that TotalView traps the “divide-by-zero” bit. The string displayed next to the FPCR register should now include EM=(ZM). Now, when your program divides by zero, it receives a SIGFPE signal, which you can catch with TotalView. See “Handling Signals” in Chapter 3 of the TotalView Users Guide for information on handling signals. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the EF=(ZE) bit in the FPSR.

Intel x86 FPSR Register

The bit settings of the Intel x86 FPSR register are outlined in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP=&lt;i&gt;</td>
<td>0x3800</td>
<td>Register &lt;i&gt; is top of FPU stack</td>
</tr>
<tr>
<td>B</td>
<td>0x8000</td>
<td>FPU busy</td>
</tr>
<tr>
<td>C0</td>
<td>0x0100</td>
<td>Condition bit 0</td>
</tr>
<tr>
<td>C1</td>
<td>0x0200</td>
<td>Condition bit 1</td>
</tr>
<tr>
<td>C2</td>
<td>0x0400</td>
<td>Condition bit 2</td>
</tr>
<tr>
<td>C3</td>
<td>0x4000</td>
<td>Condition bit 3</td>
</tr>
<tr>
<td>ES</td>
<td>0x0080</td>
<td>Exception summary status</td>
</tr>
<tr>
<td>SF</td>
<td>0x0040</td>
<td>Stack fault</td>
</tr>
<tr>
<td>EF=PE</td>
<td>0x0020</td>
<td>Precision exception</td>
</tr>
<tr>
<td>EF=UE</td>
<td>0x0010</td>
<td>Underflow exception</td>
</tr>
<tr>
<td>EF=OE</td>
<td>0x0008</td>
<td>Overflow exception</td>
</tr>
<tr>
<td>EF=ZE</td>
<td>0x0004</td>
<td>Zero divide exception</td>
</tr>
<tr>
<td>EF=DE</td>
<td>0x0002</td>
<td>Denormalized operand exception</td>
</tr>
<tr>
<td>EF=IE</td>
<td>0x0001</td>
<td>Invalid operation exception</td>
</tr>
</tbody>
</table>

Intel x86 MXCSR Register

This register contains control and status information for the SSE registers. Some of the bits in this register are editable. You cannot dive in these values.
The bit settings of the Intel x86 MXCSR register are outlined in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FZ</td>
<td>0x8000</td>
<td>Flush to zero</td>
</tr>
<tr>
<td>RC=RN</td>
<td>0x0000</td>
<td>To nearest rounding mode</td>
</tr>
<tr>
<td>RC=R-</td>
<td>0x2000</td>
<td>Toward negative infinity rounding mode</td>
</tr>
<tr>
<td>RC=R+</td>
<td>0x4000</td>
<td>Toward positive infinity rounding mode</td>
</tr>
<tr>
<td>RC=RZ</td>
<td>0x6000</td>
<td>Toward zero rounding mode</td>
</tr>
<tr>
<td>EM=PM</td>
<td>0x1000</td>
<td>Precision mask</td>
</tr>
<tr>
<td>EM=UM</td>
<td>0x0800</td>
<td>Underflow mask</td>
</tr>
<tr>
<td>EM=OM</td>
<td>0x0400</td>
<td>Overflow mask</td>
</tr>
<tr>
<td>EM=ZM</td>
<td>0x0200</td>
<td>Divide-by-zero mask</td>
</tr>
<tr>
<td>EM=DM</td>
<td>0x0100</td>
<td>Denormal mask</td>
</tr>
<tr>
<td>EM=IM</td>
<td>0x0080</td>
<td>Invalid operation mask</td>
</tr>
<tr>
<td>DAZ</td>
<td>0x0040</td>
<td>Denormals are zeros</td>
</tr>
<tr>
<td>EF=PE</td>
<td>0x0020</td>
<td>Precision flag</td>
</tr>
<tr>
<td>EF=UE</td>
<td>0x0010</td>
<td>Underflow flag</td>
</tr>
<tr>
<td>EF=OE</td>
<td>0x0008</td>
<td>Overflow flag</td>
</tr>
<tr>
<td>EF=ZE</td>
<td>0x0004</td>
<td>Divide-by-zero flag</td>
</tr>
<tr>
<td>EF=DE</td>
<td>0x0002</td>
<td>Denormal flag</td>
</tr>
<tr>
<td>EF=IE</td>
<td>0x0001</td>
<td>Invalid operation flag</td>
</tr>
</tbody>
</table>
Sun SPARC

This section has the following information:

- SPARC General Registers
- SPARC PSR Register
- SPARC Floating-Point Registers
- SPARC FPSR Register
- Using the SPARC FPSR Register

**NOTE >>** The SPARC processor supports the IEEE floating-point format.

### SPARC General Registers

TotalView displays the SPARC general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>G0</td>
<td>Global zero register</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td>$g0</td>
</tr>
<tr>
<td>G1 - G7</td>
<td>Global registers</td>
<td>$int</td>
<td>yes</td>
<td>yes</td>
<td>$g1 - $g7</td>
</tr>
<tr>
<td>O0 - O5</td>
<td>Outgoing parameter registers</td>
<td>$int</td>
<td>yes</td>
<td>yes</td>
<td>$o0 - $o5</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer</td>
<td>$int</td>
<td>yes</td>
<td>yes</td>
<td>$sp</td>
</tr>
<tr>
<td>O7</td>
<td>Temporary register</td>
<td>$int</td>
<td>yes</td>
<td>yes</td>
<td>$o7</td>
</tr>
<tr>
<td>L0 - L7</td>
<td>Local registers</td>
<td>$int</td>
<td>yes</td>
<td>yes</td>
<td>$l0 - $l7</td>
</tr>
<tr>
<td>I0 - I5</td>
<td>Incoming parameter registers</td>
<td>$int</td>
<td>yes</td>
<td>yes</td>
<td>$i0 - $i5</td>
</tr>
<tr>
<td>FP</td>
<td>Frame pointer</td>
<td>$int</td>
<td>yes</td>
<td>yes</td>
<td>$fp</td>
</tr>
<tr>
<td>I7</td>
<td>Return address</td>
<td>$int</td>
<td>yes</td>
<td>yes</td>
<td>$i7</td>
</tr>
<tr>
<td>PSR</td>
<td>Processor status register</td>
<td>$int</td>
<td>yes</td>
<td>no</td>
<td>$psr</td>
</tr>
<tr>
<td>Y</td>
<td>Y register</td>
<td>$int</td>
<td>yes</td>
<td>yes</td>
<td>$y</td>
</tr>
<tr>
<td>WIM</td>
<td>WIM register</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>TBR</td>
<td>TBR register</td>
<td>$int</td>
<td>no</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>
Architectures / Sun SPARC

### SPARC PSR Register

For your convenience, TotalView interprets the bit settings of the SPARC PSR register. You can edit the value of the PSR and set some of the bits outlined in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET</td>
<td>0x00000020</td>
<td>Traps enabled</td>
</tr>
<tr>
<td>PS</td>
<td>0x00000040</td>
<td>Previous supervisor</td>
</tr>
<tr>
<td>S</td>
<td>0x00000080</td>
<td>Supervisor mode</td>
</tr>
<tr>
<td>EF</td>
<td>0x00001000</td>
<td>Floating-point unit enabled</td>
</tr>
<tr>
<td>EC</td>
<td>0x00002000</td>
<td>Coprocessor enabled</td>
</tr>
<tr>
<td>C</td>
<td>0x00100000</td>
<td>Carry condition code</td>
</tr>
<tr>
<td>V</td>
<td>0x00200000</td>
<td>Overflow condition code</td>
</tr>
<tr>
<td>Z</td>
<td>0x00400000</td>
<td>Zero condition code</td>
</tr>
<tr>
<td>N</td>
<td>0x00800000</td>
<td>Negative condition code</td>
</tr>
</tbody>
</table>

### SPARC Floating-Point Registers

TotalView displays the SPARC floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Data Type</th>
<th>Edit</th>
<th>Dive</th>
<th>Specify in Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0, F1, F0_F1</td>
<td>Floating-point registers (f registers), used singly</td>
<td>$float</td>
<td>no</td>
<td>yes</td>
<td>$f0, $f1, $f0_f1</td>
</tr>
<tr>
<td>F2 - F31</td>
<td>Floating-point registers (f registers), used singly</td>
<td>$float</td>
<td>yes</td>
<td>yes</td>
<td>$f2 - $f31</td>
</tr>
<tr>
<td>F0, F1, F0_F1</td>
<td>Floating-point registers (f registers), used as pairs</td>
<td>$double</td>
<td>no</td>
<td>yes</td>
<td>$f0, $f1, $f0_f1</td>
</tr>
<tr>
<td>F0/F1 - F30/ F31</td>
<td>Floating-point registers (f registers), used as pairs</td>
<td>$double</td>
<td>yes</td>
<td>yes</td>
<td>$2 - $f30_f31</td>
</tr>
</tbody>
</table>
TotalView allows you to use these registers singly or in pairs, depending on how they are used by your program. For example, if you use F1 by itself, its type is `$float`, but if you use the F0/F1 pair, its type is `$double`.

## SPARC FPSR Register

For your convenience, TotalView interprets the bit settings of the SPARC FPSR register. You can edit the value of the FPSR and set it to any of the bit settings outlined in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEXC=NX</td>
<td>0x00000001</td>
<td>Current inexact exception</td>
</tr>
<tr>
<td>CEXC=DZ</td>
<td>0x00000002</td>
<td>Current divide by zero exception</td>
</tr>
<tr>
<td>CEXC=UF</td>
<td>0x00000004</td>
<td>Current underflow exception</td>
</tr>
<tr>
<td>CEXC=OF</td>
<td>0x00000008</td>
<td>Current overflow exception</td>
</tr>
<tr>
<td>CEXC=NV</td>
<td>0x00000010</td>
<td>Current invalid exception</td>
</tr>
<tr>
<td>AEXC=NX</td>
<td>0x00000020</td>
<td>Accrued inexact exception</td>
</tr>
<tr>
<td>AEXC=DZ</td>
<td>0x00000040</td>
<td>Accrued divide by zero exception</td>
</tr>
<tr>
<td>AEXC=UF</td>
<td>0x00000080</td>
<td>Accrued underflow exception</td>
</tr>
<tr>
<td>AEXC=OF</td>
<td>0x00000100</td>
<td>Accrued overflow exception</td>
</tr>
<tr>
<td>AEXC=NV</td>
<td>0x00000200</td>
<td>Accrued invalid exception</td>
</tr>
<tr>
<td>EQ</td>
<td>0x00000000</td>
<td>Floating-point condition =</td>
</tr>
<tr>
<td>LT</td>
<td>0x00000400</td>
<td>Floating-point condition &lt;</td>
</tr>
<tr>
<td>GT</td>
<td>0x00000800</td>
<td>Floating-point condition &gt;</td>
</tr>
<tr>
<td>UN</td>
<td>0x00000c00</td>
<td>Floating-point condition unordered</td>
</tr>
<tr>
<td>QNE</td>
<td>0x00002000</td>
<td>Queue not empty</td>
</tr>
<tr>
<td>NONE</td>
<td>0x00000000</td>
<td>Floating-point trap type None</td>
</tr>
<tr>
<td>IEEE</td>
<td>0x00004000</td>
<td>Floating-point trap type IEEE Exception</td>
</tr>
<tr>
<td>UFIN</td>
<td>0x00008000</td>
<td>Floating-point trap type Unfinished FPop</td>
</tr>
<tr>
<td>UIMP</td>
<td>0x0000c000</td>
<td>Floating-point trap type Unimplemented FPop</td>
</tr>
<tr>
<td>SEQE</td>
<td>0x00010000</td>
<td>Floating-point trap type Sequence Error</td>
</tr>
</tbody>
</table>
Using the SPARC FPSR Register

The SPARC processor does not catch floating-point errors by default. You can change the value of the FPSR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPSR to include `0x01000000` so that TotalView traps the “divide by zero” bit. The string displayed next to the FPSR register should now include `TEM=(DZ)`. Now, when your program divides by zero, it receives a `SIGFPE` signal, which you can catch with TotalView. See “Handling Signals” in Chapter 3 of the TotalView User Guide for more information. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the `AEXC=(DZ)` bit.

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td>0x00400000</td>
<td>Nonstandard floating-point FAST mode</td>
</tr>
<tr>
<td>TEM=NX</td>
<td>0x00800000</td>
<td>Trap enable mask - Inexact Trap Mask</td>
</tr>
<tr>
<td>TEM=DZ</td>
<td>0x01000000</td>
<td>Trap enable mask - Divide by Zero Trap Mask</td>
</tr>
<tr>
<td>TEM=UF</td>
<td>0x02000000</td>
<td>Trap enable mask - Underflow Trap Mask</td>
</tr>
<tr>
<td>TEM=OF</td>
<td>0x04000000</td>
<td>Trap enable mask - Overflow Trap Mask</td>
</tr>
<tr>
<td>TEM=NV</td>
<td>0x08000000</td>
<td>Trap enable mask - Invalid Operation Trap Mask</td>
</tr>
<tr>
<td>EXT</td>
<td>0x00000000</td>
<td>Extended rounding precision - Extended precision</td>
</tr>
<tr>
<td>SGL</td>
<td>0x10000000</td>
<td>Extended rounding precision - Single precision</td>
</tr>
<tr>
<td>DBL</td>
<td>0x20000000</td>
<td>Extended rounding precision - Double precision</td>
</tr>
<tr>
<td>NEAR</td>
<td>0x00000000</td>
<td>Rounding direction - Round to nearest (tie-even)</td>
</tr>
<tr>
<td>ZERO</td>
<td>0x40000000</td>
<td>Rounding direction - Round to 0</td>
</tr>
<tr>
<td>PINF</td>
<td>0x80000000</td>
<td>Rounding direction - Round to +Infinity</td>
</tr>
<tr>
<td>NINF</td>
<td>0xc0000000</td>
<td>Rounding direction - Round to -Infinity</td>
</tr>
</tbody>
</table>
This section contains supplementary information, currently a single appendix, that provides information on creating startup files for various Message Passing Interface (MPI) implementations.
Appendix A

MPI Startup

Overview

Here you will find information that will allow you to create startup profiles for environments that Rogue Wave Software doesn't define. Any customizations made to your MPI environment will be available for later selection in the Sessions Manager where they will appear in the File > Debug New Parallel Program dialog's Parallel System Name list.

Rogue Wave Software products know about different Message Passing Interface (MPI) implementations. Because so many implementations are standard, our products usually do the right thing. Unfortunately, subtle differences in your environment or an implementation can cause difficulties that prevent our products from automatically starting your program. In these cases, you must declare what needs to be done.
Customizing Your Parallel Configuration

The File > Debug New Parallel Program dialog box (TotalView) or the Add parallel program screen (MemoryScape) let you select a parallel configuration. If the default configurations that Rogue Wave Software provides do not meet your needs, you can either overwrite these configurations or create new ones.

The default definitions for parallel configurations reside in the parallel_support.tvd file, located in your totalview/lib installation directory. For TotalView — and MemoryScape when used with TotalView — you can use the variable TV::parallel_configs to customize parallel configurations. For standalone MemoryScape, you need to instead add any new configurations directly to the parallel_support.tvd file. Both these methods are discussed here.

TotalView

If you are using TotalView, set the TV::parallel_configs variable, either local to your TotalView installation or globally:

- Globally, in your system's .tvdr file. If you set this variable here, everyone using this TotalView version will see the definition.
- Locally, in your .totalview/tvdr file. You will be the only person to see this definition when you start TotalView.

You can also directly edit the parallel_support.tvd file, located in the totalview/lib installation directory area, but reinstalling TotalView overwrites this file so this is not recommended.

For TotalView, if you are using a locally-installed MPI implementation, you should add it to your PATH variable. By default, both TotalView and MemoryScape use the information in PATH to find the parallel launcher (for example, mpirun, mpiexec, poe, srn, prun, dmpirun, and so on). Generally, if you can run your parallel job from a command line, TotalView can also run it.

If you have multiple installed MPI systems — for example, multiple versions of MPICH installed on a common file server — only one can be in your path. In this case, specify an absolute path to launch it, which means you will need to customize the TV::parallel_configs list variable or the parallel_support.tvd file contained within your installation directory so that it does not rely on your PATH variable.

The easiest way to create your own startup configuration for TotalView is to copy a similar configuration from the TV::private::parallel_configs_base variable (found in the parallel_support.tvd file, located in your installation directory at totalview/lib) to the TV::parallel_configs variable, and then edit it. Save the TV::parallel_configs variable in the tvdr file located in the .totalview subdirectory in your home directory. For standalone MemoryScape, please see Standalone MemoryScape.
When you add configurations, they are simply added to a list. This means that if TotalView supplies a definition named `foo` and you create a definition also named `foo`, both exist and your product chooses the first one in the list. Because both are displayed, be careful to give each new definition a unique name.

**Standalone MemoryScape**

For the standalone MemoryScape product, to customize the way an MPI program starts up, edit the `parallel-support.tvd` file, located in the `totalview/lib` installation directory area, using its existing syntax and definitions as a model for any new MPI implementations you add.

Note that this file is overwritten when you install a new TotalView or MemoryScape release. Be sure to make a backup copy of any customizations you make to this file. See Customizing Your Parallel Configuration for information on how to edit this file.

Make a backup copy of customizations you add to the `parallel_support.tvd` file, since the file is overwritten if you reinstall TotalView or MemoryScape.
Example Parallel Configuration Definitions

This section provides three examples of customized parallel configurations. See Customizing Your Parallel Configuration on page 415 for information on where to place these definitions.

NOTE >> Any customizations made to your MPI environment will be available for later selection in the Sessions Manager where they will appear in the File > Debug New Parallel Program dialog's Parallel System Name list.

Here are three examples:

```plaintext
dset TV::parallel_configs {
    #Argonne MPICH
    name: MPICH;
description: Argonne MPICH;
    starter: mpirun -tv -ksq %s %p %a;
    style: setup_script;
    tasks_option: -np;
    nodes_option: -nodes;
    env_style: force;
    pretest: mpichversion;

    #Argonne MPICH2
    name: MPICH2;
    description: Argonne MPICH2;
    starter: $mpiexec -tvsu %s %p %a;
    style: manager_process;
    tasks_option: -n;
    env_option: -env;
    env_style: assign_space_repeat;
    comm_world: 0x44000000;
    pretest: mpich2version

    # AIX POE
    name: poe - AIX;
    description: IBM PE - AIX;
    tasks_option: -procs;
    tasks_env: MP_PROCS;
    nodes_option: -nodes;
    starter: /bin/poe %p %a %s;
    style: bootstrap;
    env: NLS_PATH=/usr/lib/nls/msg/%L/%N/:
       /usr/lib/nls/msg/%L/N.cat;
    service_tids: 2 3 4;
    comm_world: 0;
    pretest: test -x /bin/poe
```
All lines (except for comments) end with a semi-colon (;). Add spaces freely to improve the readability of these definitions as TotalView and MemoryScape ignore them.

Notice that the MPICH2 definition contains the $mpiexec variable. This variable is defined elsewhere in the parallel_support.tvd file as follows:

```plaintext
set mpiexec mpiexec;
```

There is no limit to how many definitions you can place within the parallel_support.tvd file or within a variable. The definitions you create will appear in the Parallel system pulldown list in the File > New dialog box (TotalView) or the Add parallel program screen (MemoryScape) and can be used as an argument to the --mpi option of the CLI's dload command.

Note that for MemoryScape, you do not set this variable because the tvdrc file is not read. Rather, directly edit the parallel_support.tvd file.

The fields that you can set are as follows:

- **comm_world**
  Only use this option when style is set to bootstrap. This variable is the definition of MPI_COMM_WORLD in C and C++. MPI_COMM_WORLD is usually a #define or enum to a special number or a pointer value. If you do not include this field, TotalView and MemoryScape cannot acquire the rank for each MPI process.

- **description** (optional) A string describing what the configuration is used for. There is no length limit.

- **env** (optional) Defines environment variables that are placed in the starter program's environment. (Depending on how the starter works, these variables may not make their way into the actual ranked processes.) If you are defining more than one environment variable, define each in its own env clause.
  
The format to use is:

```plaintext
variable_name=value
```

- **env_option** (optional) Names the command-line option that exports environment variables to the tasks started by the launcher program. Use this option along with the env_style field.

- **env_style** (optional) Contains a list of environment variables that are passed to tasks.
  
  **assign:** The argument to be inserted to the command-line option named in env_option is a comma-separated list of environment variable name=value pairs; that is,

  ```plaintext
  NAME1=VALUE1, NAME2=VALUE2, NAME3=VALUE3
  ```

  This option is ignored if you do not use an env_option clause.
**assign_space_repeat**: The argument after **env_option** is a space-separated name/value pair that is assigned to an environment variable. The command within **env_option** is repeated for each environment variable; that is, suppose you enter:

```
-env NAME1 VALUE1 -env NAME2 VALUE2
-env NAME3 VALUE3
```

This mode is primarily used for the **mpiexec.py** MPICH2 starter program.

**excenv**

One of the following three strings:

- **export**: The argument to be inserted after the command named in **env_option**. This is a comma-separated list of environment variable names; that is,

  \[
  \text{name1, name2, name3}
  \]

  This option is ignored if you do not use the **env_option** clause.

- **force**: Environment variables are forced into the ranked processes using a shell script. TotalView or MemoryScape will generate a script that launches the target program. The script also tells the starter to run that script. This clause requires that your home directory be visible on all remote nodes. In most cases, you will use this option when you need to dynamically link memory debugging into the target. While this option does not work with all MPI implementations, it is the most reliable method for MPICH1.

- **none**: No argument is inserted after **env_option**.

**msq_lib**

(optional) Names the dynamically loaded library that TotalView and MemoryScape use when it needs to locate message queue information. You can name this file using either a relative or full pathname.

**name**

A short name describing the configuration. This name shows up in such places as the **File > New** dialog box and in the **Process > Startup Parameter**’s Parallel tab in TotalView and the **Add parallel program** screen in MemoryScape. TotalView and MemoryScape remember which configuration you use when starting a program so that they can automatically reapply the configuration when you restart the program.

Because the configuration is associated with a program’s name, renaming or moving the program destroys this association.

**nodes_option**

Names the command-line option (usually -**nodes**) that sets the number of node upon which your program runs. This statement does not define the value that is the argument to this command-line option.

Only omit this statement if your system doesn’t allow you to control the number of nodes from the command line. If you set this value to zero (“0”), this statement is omitted.

**pretest**

(optional) Names a shell command that is run before the parallel job is launched. This command must run quickly, produce a timely response, and have no side-effects. This is a test, not a setup hook.
TotalView or MemoryScape may kill the test if it takes too long. It may call it more than once to be sure if everything is OK. If the shell command `exit` is not as expected, TotalView or MemoryScape complains and asks for permission before continuing.

**pretext_exit**

The expected error code of the pretest command. The default is zero.

**service_tids**

(optional) The list of thread IDs that TotalView and MemoryScape marks as service threads. When using TotalView, you can use the `View > Display Managers` command to tell TotalView to display them.

A service thread differs from a system manager thread in that it is created by the parallel runtime and are not created by your program. POE for example, often creates three service threads.

**starter**

Defines a template that TotalView and MemoryScape use to create the command line that starts your program. In most cases, this template describes the relative position of the arguments. However, you can also use it to add extra parameters, commands, or environment variables. Here are the three substation parameters:

- `%a`: Replaced with the command-line arguments passed to rank processes.
- `%p`: Replaced with the absolute pathname of the target program.
- `%s`: Replaced with additional startup arguments. These are parameters to the starter process, not the rank processes.

For example:

```
starter: mpirun -tv -all-local %s %p %a;
```

When the user selects a value for the option indicated by the `nodes_option` and `tasks_options`, the argument and the value are placed within the `%s` parameter. If you enter a value of 0 for either of these, MemoryScape and TotalView omit the parameter. In MemoryScape, 0 is the default.

**style**

MPI programs are launched in two ways: either by a manager process or by a script. Use this option to name the method, as follows:

- **manager_process**: The parallel system uses a binary manager process to oversee process creation and process lifetime. Our products attach to this process and communicate with it using its debug interface. For example, IBM’s poe uses this style.

  ```
  style: manager_process;
  ```

- **setup_script**: The parallel system uses a script—which is often `mpirun`—to set up the arguments, environment, and temporary files. However, the script does not run as part of the parallel job. This script must understand the `-tv` command-line option and the `TOTALVIEW` environment variable.

- **bootstrap**: The parallel system attempts to launch an uninstrumented MPI by interposing TotalView or MemoryScape inside the parallel launch sequence in place of the target program. This does not work for MPICH and SGI MPT.
tasks_env
The name of an environment variable whose value is the expected number of parallel tasks. This is consulted when the user does not explicitly specify a task count.

tasks_option
(sometimes required) Lets you define the option (usually -np or -procs) that controls the total number of tasks or processes.

Only omit this statement if your system doesn't allow you to control the number of tasks from the command line. If you set this to 0, this statement is omitted.
Index

-serial device 350
- -add-gnu-debuglink command-line option 362

Symbols
- .debug gnu_debuglink copy 362
- .totalview/lib_cache subdirectory 47
- .tvd files 200
- @ symbol for action point 100
- * expr 308
- /proc file system 371
- # scoping separator character 38, 43, 99
- %A server launch replacement character 352
- %B server launch replacement character 352
- %C server launch replacement character 352
- %D path name replacement character 352
- %H hostname replacement character 352
- %L host and port replacement character 353
- %N line number replacement character 353
- %P password replacement character 353
- %S source file replacement character 353
- %t1 file replacement character 353
- %t2 file replacement character 353
- %V verbosity setting replacement character 354
- = symbol for PC of current buried stack frame 100
- > symbol for PC 100
- $mpiexec variable 418
- $newval variable in watchpoints 153
- $oldval variable in watchpoints 153
- $stop function 76, 117
- A -a option to totalview command 336
- ac, see actions command acquiring processes 270
- action point identifiers 28
- Action Point > Save All command 259
- action point identifiers 71
- action points
  - autoloading 259
  - default for newly created 253
  - default property 253
  - deleting 61, 176, 189, 192, 217
  - disabling 28, 29, 64
  - displaying 28
  - enabling 28, 29
  - identifiers 29
  - information about 28, 29, 169
  - loading 29
  - loading automatically 341
  - loading saved information 29
  - reenabling 71
  - saving 29
  - saving information about 29
  - scope of what is stopped 253
  - setting at location 28
  - sharing 252
  - stopping when reached 288
- actionpoint properties 176
- actionpoint command 176
- actions
  - list of disabled (ambiguous context) 28
  - list of enabled (ambiguous context) 28
- actions, see actions command
- activating type transformations 313
- Add parallel program screen 415
- adding group members 86
- adding groups 85
- address 201
- address property 176
- addressing_callback 221
- advancing by steps 221
- aggregate data 303
- AIX
  - compiling on 358
  - linking C++ to dbfork library 365
  - linking to dbfork library 364
  - swap space 373
  - aix_use_fast_trap command-line option 336
  - aix_use_fast_trap variable 256, 290
  - alias command 24
  - aliases
    - default 24
    - removing 173
  - append, see dlappend command
  - appending to CLI variable lists 98
- architectures 254
  - Intel IA-64 398
  - Intel-x86 386, 404
  - PowerPC 391
  - SPARC 409
- arenas 79, 111
- ARGS variable 248
- ARGS_DEFAULT variable 127, 132, 248
- ARGS(dpmid) variable 127, 132
- arguments
  - command line 132
  - default 248
  - for totalview command 334
  - for tvdsvr command 349
- arrays
automatic
dereferencing 256
gathering statistical
data 117
number of elements
displayed 256
arriving at barrier 39
as, see dassign command
ask on dlopen option 377
ask_on_dlopen option 378
ask_on_dlopen variable 256
assemble, displaying
symbolically 272
assign, see dassign command
assigning string values 33
assigning values 33
asynchronous execution 76
at, see dattach command
attach, see dattach command
attaching to parallel
processes 35
attaching to processes 35
attaching to ranks 35
attaching, using PIDs 36
auto_array_cast_bounds
variable 256
auto_array_cast_enabled
variable 256
auto_deref_in_all_c variable 257
auto_deref_in_all_fortran
variable 257
auto_deref_initial_c variable 257
auto_deref_initial_fortran
variable 258
auto_deref_nested_c
variable 258
auto_deref_nested_fortran
variable 258
auto_load_breakpoints
variable 259
auto_read_symbols_at_stop
variable 259
auto_save_breakpoints
variable 259
automatic dereferencing 257
automatic dereferencing of
arrays 256
automatically attaching to
processes 283
B
b, see dbreak command
ba, see dbarrier command
background command-line
option 336
back-tick analogy 26
barrier breakpoint 39
barrier is satisfied 249, 260
BARRIER_STOP_ALL variable 38,
40, 248
barrier_stop_all variable 260
BARRIER_STOP_WHEN_DONE
variable 38, 248
barrier_stop_when_done
variable 260
barrier, see dbarrier command
barriers 38, 39
arriving 39
creating 39
what else is stopped 38
base_name 202
Batch debugging, tvscript 230
baud rate, specifying 350
baw, see dbarrier command
bg command-line option
336
bkeepfile command-line
option 365
blocking command input 152
blocking input 152
Bluegene I/O interface 260
Bluegene launch string 261
Bluegene server timeout 261
bluegene_io_interface
variable 260
bluegene_launch_string
variable 261
bluegene_server_launch_time-
out variable 261
break, see dbreak command
breakpoints
automatically loading 259
barrier 38
default file in which set
defined 44
file 29, 259
popping Process
Window 300
setting at functions 44
setting expression 43
stopping all processes at 43
temporary 147
triggering 44
bt, see dbreak command
build_struct_transform
defined 308
example 307
lists 308
members argument 308
name argument 308
bulk_launch 353
bulk_launch_base_timeout
variable 261
bulkLaunch_enabled
variable 261
bulk_launch_incr_timeout
variable 261
bulk_launch_string variable 261
bulk_launch_tmpfile1_header_
line variable 262
bulk_launch_tmpfile1_host_
lines variable 262
bulk_launch_tmpfile1_trailer_
line variable 262
bulk_launch_tmpfile2_ head-
er_line variable 262
bulk_launch_tmpfile2_host_
lines variable 262
bulk_launch_tmpfile2_trailer_
line variable 262
buried stack frame 99
by_language_rules 198
by_path 198
by_type_index 198
C
C language
escape characters 33
using with C++View 325
C shell 372
c_type_strings 263
c_type_strings variable 263
C++
demangler 267, 338
including libdbfork.h 365
STL instantiation 304
C++View
and multithread safety 322
and programming design 321
and template classes 317
compiling and linking 330
described 314
element display function 316
elements of use with ReplayEngine 324
examples 331
use of with ReplayEngine 323

using with C 325

Caching cache, flushing 47

and Memory Debugger 294

Cache, see dcache command

Call stack 150
displaying 161
see also stack frame 17, 160
call tree saved position 297
callback command-line option 349
callback list 281
callback_host 349
callback_ports 350
callbacks 279
after loading a program 284
when opening the CLI 283
capture command 26, 165
case sensitive searching 285
cast subcommand 198
Cast to array with bounds checkbox 256
casting variables 120
ccq command-line option 337
cGROUP variable 249
changing CLI variables 134, 135
changing dynamic context 150
changing focus 79
changing value of program variable 33, 52, 95, 114, 129, 145, 147
chase_mouse variable 295

Checking interior pointers

Variable 293
class transformations 306
classes, transforming 308

CLI
activated from GUI flag 295
sourcing files 307
startup file 307

CLI variables
aix_use_fast_trap 256, 290
ARGS 248
ARGS_DEFAULT 248
ask_on_dlopen 256
auto_array_cast_bounds 25
auto_array_cast_enabled 25
auto_deref_in_all_c 257
auto_deref_in_all_fortran 257
auto_deref_inital_fortran 2
auto_deref_inital_c 257
auto_deref_nested_c 258
auto_deref_nested_fortran 258
auto_load_breakpoints 259
auto_read_symbols_at_stop 259
auto_save_breakpoints 259
BARRIER_STOP_ALL 38, 40, 248
barrier_stop_all 260
BARRIER_STOP_WHEN_DONE 38, 248
barrier_stop_when_done 260
blue_gene_launch_string 26
bluegene_io_interface 260
bluegene_server_launch_timeout 261
bulk_launch_base_timeout 261
bulk_launch_enabled 261
bulk_launch_incr_timeout 2
bulk_launch_string 261
bulk_launch_tmpfile1_header_line 262
bulk_launch_tmpfile1_host_lines 262
bulk_launch_tmpfile2_header_line 262
bulk_launch_tmpfile2_host_lines 262
bulk_launch_tmpfile2_trailer_line 262
c_type_strings 263
CROUP 249
changing 134, 135
chase_mouse 295
comline_patch_area_base 264
comline_path_area_length 264
COMMAND_EDITING 249
command_editing 265
compile_expressions 265
compiler_vars 265
control_c_quick_shutdown 266
copyright_string 266
current_cplusplus_demangler 267
current_fortran_demangler 267
data_format_double 268
data_format_ext 269
data_format_int16 269
data_format_int32 269
data_format_int64 270
data_format_single 270
data_format_singlen 270
dbfork 270
default value for 135
default_snippet_extent 293
default_stdout_append 270, 271
default_stderr_append 270, 271
default_stderr_filename 271
default_stdout_filename 271
deleting 134, 135
display_assembler_symbolically 272
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>display_bytes_kb_mb</td>
<td>295</td>
</tr>
<tr>
<td>display_font_dpi</td>
<td>295</td>
</tr>
<tr>
<td>dll_ignore_prefix</td>
<td>272</td>
</tr>
<tr>
<td>dll_read_all_symbols</td>
<td>272</td>
</tr>
<tr>
<td>dll_read_loader_symbols_only</td>
<td>272</td>
</tr>
<tr>
<td>dll_stop_suffix</td>
<td>273, 274, 275</td>
</tr>
<tr>
<td>dump_core</td>
<td>275</td>
</tr>
<tr>
<td>dwhere_qualification_level</td>
<td>275</td>
</tr>
<tr>
<td>dynamic</td>
<td>276</td>
</tr>
<tr>
<td>editor_launch_string</td>
<td>276</td>
</tr>
<tr>
<td>enabled</td>
<td>295</td>
</tr>
<tr>
<td>env</td>
<td>276</td>
</tr>
<tr>
<td>errorCodes</td>
<td>117</td>
</tr>
<tr>
<td>EXECUTABLE_PATH</td>
<td>36, 100, 250</td>
</tr>
<tr>
<td>EXECUTABLE_SEARCH_PATH</td>
<td>250</td>
</tr>
<tr>
<td>fixed_font</td>
<td>295</td>
</tr>
<tr>
<td>fixed_font_family</td>
<td>296</td>
</tr>
<tr>
<td>fixed_font_size</td>
<td>296</td>
</tr>
<tr>
<td>follow_clone</td>
<td>277</td>
</tr>
<tr>
<td>font</td>
<td>296</td>
</tr>
<tr>
<td>force_default_cplusplus_demangler</td>
<td>277</td>
</tr>
<tr>
<td>force_default_f9x_demangler</td>
<td>277</td>
</tr>
<tr>
<td>force_window_position</td>
<td>296</td>
</tr>
<tr>
<td>frame_offset_x</td>
<td>296</td>
</tr>
<tr>
<td>frame_offset_y</td>
<td>296</td>
</tr>
<tr>
<td>geometry_call_tree</td>
<td>297</td>
</tr>
<tr>
<td>geometry_cli</td>
<td>297</td>
</tr>
<tr>
<td>geometry_globals</td>
<td>297</td>
</tr>
<tr>
<td>geometry_help</td>
<td>297</td>
</tr>
<tr>
<td>geometry_memory_stats</td>
<td>298</td>
</tr>
<tr>
<td>geometry_message_queue</td>
<td>298</td>
</tr>
<tr>
<td>geometry_message_queue_graph</td>
<td>298</td>
</tr>
<tr>
<td>geometry_modules</td>
<td>298</td>
</tr>
<tr>
<td>geometry_process</td>
<td>298</td>
</tr>
<tr>
<td>geometry_ptset</td>
<td>299</td>
</tr>
<tr>
<td>geometry_root</td>
<td>299</td>
</tr>
<tr>
<td>geometry_thread_objects</td>
<td>299</td>
</tr>
<tr>
<td>geometry_variable</td>
<td>299</td>
</tr>
<tr>
<td>geometry_variable_stats</td>
<td>299</td>
</tr>
<tr>
<td>global_typenames</td>
<td>277</td>
</tr>
<tr>
<td>gnu_debuglink</td>
<td>278, 362</td>
</tr>
<tr>
<td>gnu_debuglink_checksum_flag</td>
<td>362</td>
</tr>
<tr>
<td>gnu_debuglink_global_directory</td>
<td>278, 363</td>
</tr>
<tr>
<td>gnu_debuglink_search_path</td>
<td>278</td>
</tr>
<tr>
<td>GROUP</td>
<td>250</td>
</tr>
<tr>
<td>GROUPS</td>
<td>102, 251</td>
</tr>
<tr>
<td>hia_allow_ibm_poe</td>
<td>293</td>
</tr>
<tr>
<td>ignore_control_c</td>
<td>279</td>
</tr>
<tr>
<td>ignore_snippets</td>
<td>293</td>
</tr>
<tr>
<td>image_load_callbacks</td>
<td>279</td>
</tr>
<tr>
<td>in_setup</td>
<td>280</td>
</tr>
<tr>
<td>kcc_classes</td>
<td>280</td>
</tr>
<tr>
<td>keep_expressions</td>
<td>300</td>
</tr>
<tr>
<td>keep_search_dialog</td>
<td>300</td>
</tr>
<tr>
<td>kernel_launch_string</td>
<td>281</td>
</tr>
<tr>
<td>kill_callbacks</td>
<td>281</td>
</tr>
<tr>
<td>leak_check_interior_pointer</td>
<td>293, 294</td>
</tr>
<tr>
<td>leak_max_cache</td>
<td>294</td>
</tr>
<tr>
<td>leak_maxแชมk</td>
<td>294</td>
</tr>
<tr>
<td>library_cache_directory</td>
<td>281</td>
</tr>
<tr>
<td>LINES_PER_SCREEN</td>
<td>251</td>
</tr>
<tr>
<td>local_interface</td>
<td>281</td>
</tr>
<tr>
<td>local_server</td>
<td>282</td>
</tr>
<tr>
<td>local_server_launch_string</td>
<td>283</td>
</tr>
<tr>
<td>MAX_LIST</td>
<td>99, 252</td>
</tr>
<tr>
<td>message_queue</td>
<td>282</td>
</tr>
<tr>
<td>nptl_threads</td>
<td>282</td>
</tr>
<tr>
<td>OBJECT_SEARCH_MAPPING</td>
<td>250, 252</td>
</tr>
<tr>
<td>OBJECT_SEARCH_PATH</td>
<td>250, 252</td>
</tr>
<tr>
<td>open_cli_callback</td>
<td>283</td>
</tr>
<tr>
<td>parallel</td>
<td>283</td>
</tr>
<tr>
<td>parallel_attach</td>
<td>283</td>
</tr>
<tr>
<td>parallel_configs</td>
<td>415</td>
</tr>
<tr>
<td>parallel_stop</td>
<td>283</td>
</tr>
<tr>
<td>platform</td>
<td>284</td>
</tr>
<tr>
<td>pop_at_breakpoint</td>
<td>300</td>
</tr>
<tr>
<td>pop_on_error</td>
<td>300</td>
</tr>
<tr>
<td>process_load_callbacks</td>
<td>284</td>
</tr>
<tr>
<td>PROMPT</td>
<td>252</td>
</tr>
<tr>
<td>PTSET</td>
<td>252</td>
</tr>
<tr>
<td>restart_threshold</td>
<td>285</td>
</tr>
<tr>
<td>save_window_pipe_or_filename</td>
<td>285</td>
</tr>
<tr>
<td>search_case_sensitive</td>
<td>285</td>
</tr>
<tr>
<td>server_launch_enabled</td>
<td>285</td>
</tr>
<tr>
<td>server_launch_string</td>
<td>286</td>
</tr>
<tr>
<td>server_launch_timeout</td>
<td>286</td>
</tr>
<tr>
<td>server_response_wait_time</td>
<td>286</td>
</tr>
<tr>
<td>SHARE_ACTION_POINT</td>
<td>253</td>
</tr>
<tr>
<td>share_action_point</td>
<td>286</td>
</tr>
<tr>
<td>shared_data_filters</td>
<td>294</td>
</tr>
<tr>
<td>show_startup_parameters</td>
<td>301</td>
</tr>
<tr>
<td>show_sys_thread_id</td>
<td>301</td>
</tr>
<tr>
<td>signal_handling_mode</td>
<td>287</td>
</tr>
<tr>
<td>single_click_dive_enabled</td>
<td>301</td>
</tr>
<tr>
<td>sourcePane_tab_width</td>
<td>288</td>
</tr>
<tr>
<td>SOURCE_SEARCH_MAPPING</td>
<td>253</td>
</tr>
<tr>
<td>SOURCE_SEARCH_PATH</td>
<td>253</td>
</tr>
<tr>
<td>spell_correction</td>
<td>288</td>
</tr>
<tr>
<td>stack_trace_qualification_level</td>
<td>288</td>
</tr>
<tr>
<td>STOP_ALL</td>
<td>43, 253</td>
</tr>
<tr>
<td>stop_all</td>
<td>288</td>
</tr>
<tr>
<td>stop_relatives_on_proc_error</td>
<td>289</td>
</tr>
<tr>
<td>suffix</td>
<td>289</td>
</tr>
<tr>
<td>TAB_WIDTH</td>
<td>100, 254</td>
</tr>
<tr>
<td>THREADS</td>
<td>254</td>
</tr>
<tr>
<td>toolbar_style</td>
<td>301</td>
</tr>
<tr>
<td>tooltips_enabled</td>
<td>301</td>
</tr>
<tr>
<td>TOTAL_VERSION</td>
<td>254</td>
</tr>
<tr>
<td>TOTALVIEW_ROOT_PATH</td>
<td>25</td>
</tr>
<tr>
<td>TOTALVIEW_TCLLIB_PATH</td>
<td>25</td>
</tr>
<tr>
<td>ttf</td>
<td>289</td>
</tr>
<tr>
<td>ui_font</td>
<td>301</td>
</tr>
<tr>
<td>ui_font_family</td>
<td>302</td>
</tr>
<tr>
<td>ui_font_size</td>
<td>302</td>
</tr>
<tr>
<td>user_threads</td>
<td>290</td>
</tr>
<tr>
<td>using_color</td>
<td>302</td>
</tr>
<tr>
<td>using_text_color</td>
<td>302</td>
</tr>
<tr>
<td>using_title_color</td>
<td>302</td>
</tr>
<tr>
<td>VERBOISE</td>
<td>255</td>
</tr>
<tr>
<td>version</td>
<td>290, 302</td>
</tr>
<tr>
<td>viewing</td>
<td>134, 135</td>
</tr>
<tr>
<td>visualizer_launch_enabled</td>
<td>291</td>
</tr>
<tr>
<td>visualizer_launch_string</td>
<td>291</td>
</tr>
<tr>
<td>visualizer_max_rank</td>
<td>291</td>
</tr>
<tr>
<td>warn_step_throw</td>
<td>291</td>
</tr>
<tr>
<td>WGROUP</td>
<td>255</td>
</tr>
</tbody>
</table>
closed loop, see closed loop
clones shared libraries 180
clusterid property 192
c, see dcont command
code snippets 293
code, displaying 99
color
foreground 339
command-line options
-ccq 337
comline_patch_area_base variable 264
comline_patch_area_length variable 264
command arguments 248
command focus 79
command input, blocking 152
command line arguments 132
command output 26
command prompt 252
command summary 8
command verb
actionpoint command 180
COMMAND_EDITING variable 249
command Editing variable 265
command, Tools > Dynamic Libraries 66
command-line options
-aix_use_fast_trap 336
-background 336
-bg 336
-ccq 337
-compiler_vars 336
- control_c_quick_shutdo wn 337
-cuda 337
dbfork 337
-debug_file 337
demangler 338
display 338
dll_ignore_prefix 338
dll_stop_suffix 338
dump_core 339
e 339
-ent 339
-env 339
-f9x_demangler 340
-fg 340
-foreground 339
global_types 340
-gnu_debuglink 340
-gnu_debuglink_checksum 340
-ipv6_support 340
-kcc_classes 341
-lib 341
-message_queue 341, 342
-mqd 342
-nccq 337
-nlb 341
-no_compiler_vars 337
-no_control_c_quick_shutdo wn 337
-no_cuda 337
-no_dbfork 337
-no_ent 339
-no_global_types 340
-no_gnu_debuglink 340
-no_gnu_debuglink_checksum 340
-no_ipv6_support 340
-no_kcc_classes 341
-no_message_queue 342
-no_mqd 342
-no_nptl_threads 343
-no_parallel 343
-no_startup_scripts 343
-no_team 346
-no_teamplus 346
-no_user_threads 336, 347
-nptl_threads 343
-parallel 343
-patch_area_base 344
-patch_area_length 344
-pid 344
-r 344
-remote 344
-s 344
-search_path 345
-serial 345
-shm 345
-signal_handling_mode 345
-stderr 346
-stderr_append 346
-stderr_is_stdout 346
-stdin 346
-stdout 346
-stdout_append 346
-team 346
-timeplus 346
tvsvr, syntax and use 348
user-defined 24
commands responding to 197
totalview 334
tvdsvr, syntax and use 348
user-defined 24
commands verb
actionpoint command 176
expr command 184
group command 189
process command 192
thread command 214
type command 217
compile_expressions variable 265
compiler property 221
compiler_vars command-line option 336
compiler_vars variable 265
compilers, KCC 280
compiling
debugging symbols 357
for C++View 330
-g compiler command-line option 357
on Bluegene 359
on HP IA-64 357
on IBM Power Linux 359
on itanium 361
on SunOS 361
on x86 360
on x86-64 360
options 357
compressed list of processes, defined. See plist.
Compressed List Syntax, defined 138
conditional watchpoints 153
connection directory 352
console output for tvdsvr 350
console output redirection 337
cont, see dcont command 214
continuing execution 83
control group variable 249
control list element 250
control_c_quick_shutdown command-line option 337
control_c_quick_shutdown variable 266
copyright_string variable 266
core dumping for TotalView 339
when needing to debug TotalView itself 275
core files, loading 36
count property 190
Cray pointers, example using Fortran with C++View 326
create subcommand 220
creating barrier breakpoints 39
creating commands 24
creating groups 85
creating new process objects 103
creating threads 83
creating type transformations 303
Ctrl+C, ignoring 279
Ctrl+D to exit CLI 164, 166
CUDA
CLI command detail 56
dcuda > block command 56
dcuda CLI command 56
support for GPU threads 23, 56
current data size limit 372
current frame marker register Intel IA-64 401
current list location 69
current_cplus_demangler variable 267
current_fortran_demangler variable 267
D
d, see ddown command 69
dactions command 29
dassign command 33
data format presentation styles 268
transforming withC++View 314
data representation simplifying using elision 320
data size 105
data size limit in C shell 372
data_format_double variable 268
data_format_ext variable 269
data_format_int16 variable 269
data_format_int32 variable 269
data_format_int64 variable 270
data_format_int8 variable 269
data_format_single variable 270
data_format_stringlen variable 270
datatype cast expr 309
datatype incompatibilities 33
dattach command 35
dbarrier command 38
dbfork command-line option 337
dbfork library 364
linking with 364
syntax 337
dbfork variable 270
dbreak command 43
setting expression in 43
dcache command 47
dcalltree command 48
dcont command 54
dcuda command 56
ddelete command 61
ddetach command 63
ddisble command 64
ddl_read_all_symbols variable 272
ddllopen command 66
ddown command 69
de, see ddelete command
deactivating action points 64
deadlocks at barriers 40
debug gnu_debuglink copy 362
debug_file command-line option 337, 350
debugger server 286, 348
debugging remote systems 47
debugging session, ending 164
dec2hex command 179
default aliases 24
default arguments 132, 248
modifying 132
default focus 79
default preferences, setting 136
default value of variables, restoring 146
default_snippet_extent 293
default_stdout_filename variable 271
default_stderr_filename variable 271
default_stderr_is_stdout variable 271
default_stdin_filename variable 271
default_stderr_append variable 270, 271
default_stderr_filenumber variable 270
default_stdin_filenumber variable 270
deferred reading, shared library symbols 196
defining MPI startup implementations 414
defining the current focus 252
delete verb, expr command 184
delete, see ddelete command
deleting action points 61, 176, 189, 192, 217
deleting cache 47
deleting CLI variables 134, 135
deleting groups 85, 86
deleting variables 146
demangler 267
C++ 267
forcing use 277
Fortran 267
overriding 338, 340
demangler command-line option 338
denable command 71
dereferencing 257, 258, 259
  C pointers
    automatically 257
  C structure pointers
    automatically 258
dereferencing values automatically 257
det, see ddetach command
detach, see ddetach command
detaching from processes 63
dexamine command 73
dflush command 76, 117
dfocus command 79
dga command 81
dgo command 83
dgroups command 85
  -add 86
  -delete 86
dhalt command 90
dheap command 91
dheap command, see also heap debugging
dhold command 95
di, see ddisable command
directory search paths 250
disable, see ddisable command
disabled action points list (ambiguous context) 28
disabling action points 28, 29, 64
display call stack 161
display command-line option 338
display_assembler_symbolically variable 272
display_bytes_kb_mb variable 295
display_font_dpi variable 295
displaying
code 99
  current execution location 161
  error message information 255
  help information 165
  information on a name lines 252
  values 117
  displaying expressions 117
  displaying memory 73
  displaying memory values 73
diving, single click 301
dkill command 97
dlappend command 98
dlist command 99, 252
dlist, number of lines displayed 252
dll command 180
  DLL Do Query on Load list 377
  DLL Don't Query on Load list 377
dll_ignore_prefix command-line option 338
dll_ignore_prefix variable 272
dll_read_loader_symbols_only 2
  dll_read_no_symbols variable 273
dll_stop_suffix command-line option 338
  dll_stop_suffix variable 273, 274, 275
dload command 102
dopen 67, 377
  ask when loading 256
dmstat command 105
dnext command 108
dnexti command 111
done property 184
double-precision data format 268
dout command 114
down, see ddown command
dpid 249
  dpid variable 249
  dpid property 215
dprint command 117
dptsets command 123
drerun - drun differences 127
drun command 126
drestart command 129
drun - drerun differences 127
drun command 97, 131
  poe issues 127, 132
  reissuing 132
dsession 134
dset command 134, 135
dstatus command 137
dstep command 141
  iterating over focus 141
duhtp, see dunhold command
duid property 193, 215
dump subcommand 198, 201
dump_core command-line option 339
dump_core variable 275
dunhold command 145
dunset command 146
duntil command 147
  group operations 148
dup command 150
dwait command 152
dwatch command 153
dwhat command 156
dwhere command 160
  levels 251
dwhere_qualification_level variable 275
dworker command 163
Dynamic Libraries page 377
dynamic library support limitations 382
dynamic linker 376
dynamic variable 276
dynamically linked program 376
dynamically loaded libraries 377
E
e command-line option 339
editor_launch_string variable 276
eliminating tab processing 100
elision, to simplify data representation 320
Emacs-like commands 265
en, see enable command
enable, see enable command
enabled action points list (ambiguous context) 28
enabled property 177
enabled variable 295
enabling action points 28, 29, 71
ending debugging session 164
ent command-line option 339
enum_values property 217
evaluated command-line option 339
evaluated property 177, 184
evaluated system
AIX 384
evaluation points, see dbreak
executing as one instruction 111
executing as one statement 108
executing source lines 141
execution
continuing 83
displaying location 161
halting 90
resuming 54
execve() 364
calling 337
catching 270
exit command 164
expr . expr 309
expr -> expr 309
expr command 117, 184
expression property 177, 184
expression system
AIX 384
expression values, printing 117
expressions in breakpoint 43
expressions, compiling 265
expressions, type transformation 308
extensions for file names 289
F
f, see dfocus command
f9x_demangler command-line option 340
fast_trap, setting 256, 290
fatal errors 372
fg command-line option 340
File > Preferences command 313
file name extensions 289
files
initialization 307
libdbfork.h 365
filter definition file 294
filters, sharing memory filters 294
fixed_font variable 295
fixed_font_family variable 296
fixed_font_size variable 296
floating point data format
double-precision 268
extended floating point 269
single-precision 270
floating point status register
Intel IA-64 402
flush, see dflush command
flushing cache 47
flushing suspended evaluations 76
focus
see also dfocus command
default 79
defining 252
temporarily changing 79
focus_groups command 186
focus_processes command 187
focus_threads command 188
focus_threads property 184
follow_clone variable 296
font variable 277
fonts 295
fixed 295, 296
ui 296, 301
ui font family 302
ui font size 302
force_default_cplus_demangler variable 277
force_default_f9x_demangler variable 277
force_window_position variable 296
foreground command-line option 339
fork()
about 364
calling 337
catching 270
formatting program data using C++View 314
Fortran
demangler 267
frame_offset_x variable 296
frame_offset_y variable 296
functions
evaluating 117
setting breakpoints at 44
G
g, see dgo command
GDP SPU Runtime System (SPU).
See spurs.
general registers
Intel IA-64 398
general_registers
Intel IA-64 398
general_registers
Intel IA-64 398
geometry_call_tree variable 297
geometry_cli_position 297
geometry_cli_variable 297
geometry_globals variable 297
geometry_memory_stats variable 298
geometry_message_queue variable 298
group
geometry_message_queue_group variable 298
geometry_modules variable 298
geometry_process variable 298
geometry_ptset variable 299
geometry_root variable 299
geometry_thread_objects variable 299
geometry_variable variable 299
geometry_variable_stats variable 299
get subcommand 198
get verb
  actionpoint command 176, 180
  expr command 184
  group command 189
  process command 192
  thread command 214
  type command 217
Global Arrays 81
  setting language for display 81
global_typenames variable 277
global_types, command-line option 340
GNU C++ STL instantiation 304
gnu_debuglink command-line option 340, 362, 363
gnu_debuglink files 362
gnu_debuglink variable 278, 362
gnu_debuglink_checksum command-line option 340, 362
gnu_debuglink_checksum variable 278
gnu_debuglink_checksum_flag variable 362
gnu_debuglink_global_directory variable 278
gnu_debuglink_global_directory command-line option 363
gnu_debuglink_global_directory variable 363
gnu_debuglink_search_path variable 278
go, see dgo command
goal breakpoint 142
GPU threads. See CUDA.
gr, see dgroups command
group command 189
group ID 255
group members, stopping flag 253
group of interest 142
GROUP variable 250
group width stepping behavior 141
groups
  accessing properties 189
  adding 85
  adding members 86
  creating 85
  deleting 85, 86
  intersecting 85
  listing 85
  naming 86
  placing processes in 36
  removing members 85
  returning list of 186
  setting properties 189
GROUPS variable 102, 251
groups, see dgroups command
H
  h, see dhalt command
  halt, see dhalt command
  halting execution 90
  handling signals 345
  handling user-level (M:N) thread packages 290
  heap size 105
  heap_size property 193
  held property 193, 215
  help command 165
  help window position 297
  hex2dec command 191
  hexadecimal conversion 179
  hia_allow_ibm_poe 293
  hiarc file 293
  hold, see dhold command
  holding processes 95
  holding threads 39, 95
  host ports 349
  hostname
    expansion 352
for tvdsvr 349
  property 193
  replacement 353
HP Tru64 UNIX
  /proc file system 371
hp, see dhold command
HP-UX
  shared libraries 376
  swap space 372
ht, see dhold command
htp, see dhalt command
I
I/O redirection 131
id property 177, 185, 190, 193, 215, 217, 221
ignore_control_c variable 279
ignore_snippets false 293
ignoring libraries by prefix 338
image browser window position 297
image file, stripped copy 362
Image information 105
image_id property 217
image_ids property 193
image_load_callbacks list 284
image_load_callbacks variable 279
in_setup variable 280
IndextermTEST 335
inet interface name 281, 341
infinite loop, see loop, infinite
info state 255
information on a name 156
initialization file 173, 307
initially_suspended_process property 185
input, blocking 152
inserting working threads 163
integer (64-bit) data format 270
integer data format
  16-bit 269
  32-bit 269
  8-bit 269
Intel IA-64
  architecture 398
current frame marker register 401
floating point status register 402
general registers 398
Intel IA-64 floating point registers 402
previous function state register 402
processor status register 400
register stack configuration register 401
Intel-x86 architecture 386, 404 floating-point registers 387, 405
FPCR register 388, 406 using 389, 407
FPSR register 389, 390, 407
general registers 386, 404
interface name for server 281, 341
interior pointers, checking 293
intersecting groups 85
ipv6_support command-line option 340
ipv6_support option 340
IRIX /proc file system 371

J
job_t::launch 371

K
k, see dkill command
-kcc_classes command-line option 341
kcc_classes command-line option 341
kcc_classes variable 280
keep_expressions variable 300
keep_search_dialog variable 300
kernel_launch_string variable 281
keys, remapping 383
keysym 383
kill_callbacks variable 281
kill, see dkill command
killing attached processes 266
killing processes 97

L
I, see dlstat command
language property 177, 218, 221
languages C, using with C++View 325
lappend, see dlappend command
launch string
for editor 276
for server (Sun only) 282
for Visualizer 291
Launch Strings page 291
launching
local server 282
processes 131
single process sever launch string 286
tvdsvr 348
Visualizer 291
lb command-line option 341
LD_BIND_NOW environment variable 376
LD_LIBRARY_PATH 364, 366
/dbfork linker option 364, 366
/dbfork option 364, 366
/dbfork_64 linker option 364
/dbfork_64 option 364, 366
leak_check_interior_pointers variable 293, 294
leak_check_interior_pointers variable 293, 294
leak_max_cache 294
leak_max_chunk 294
length property 177, 218
levels for dwhere 251
levels, moving down 69
libdbfork_64.a 366
libdbfork.a 364, 366
libdbfork.h file 365
libraries dbfork 337
ignoring by prefix 338
loading by suffix 273
loading symbols from 276
not loading based on prefix 272
shared 375
library cache data 281
library cache, flushing 47
library_cache_directory variable 281
line property 177
line_number 202
LINES_PER_SCREEN variable 251
linking
for C++View 330
linking to dbfork library 364
AIX 364
C++ and dbfork 365
SunOS 5 366
Linux swap space 373
list location 69
list_element_count_addressing_callback 222
list_element_data_addressing_callback 222
list_element_prev_addressing_callback 223
list_end_value property 223
list_first_element_addressing_callback 223
list, see dlstat command
listing groups 85
using a regular expression 87
listing lines 252
lo, see dlload command
load and loadbind 377
load, see dlload command
loader_name 202
loading
action point information 29
action points 29, 341
libraries 272, 273
programs 103
shared libraries 180
symbols from shared
libraries 276

tvd files 200

loading sessions dsession 134

local_interface variable 281
local_server variable 282
local_server_launch_string variable 282

lockstep list element 250

logical model 305

lookup subcommand 198

lookup_keys subcommand 199

loop, infinite, see infinite loop

lower_bounds_callback 222

M

manager property 215

manager threads, running 141

managing shared libraries 180

mangler, overriding 338, 340

mappings, search path 253

MAX_LIST variable 99, 252

maxdsiz_64 373

maximum data segment size 373

Maximum permissible rank field 291

mem_detect_leaks memory sub-option 238

mem_guard_blocks memory sub-option 238

mem_hoard Freed_memory memory sub-option 239

mem_paint_all memory sub-option 239

mem_paint_on Alloc memory sub-option 239

mem_paint_on_dealloc memory sub-option 239

member_type property 190

member_type_values property 190

members argument, build_struct_transform 308

members property 190

memory data size 105

heap 105

stack 105

text size 105

memory statistics window position 298

Memory Tracker, see dheap command

memory use 105

message queue graph window position 298

message queue window position 298

message verbosity variable 255

message_queue command-line option 341, 342

message_queue variable 282

mkswap command 373

modules window position 298

more processing 118

more prompt 165, 251

mounting /proc file system 371

MPI message queues 282

MPI startup 414

mqd command-line option 342

multiprocess programs, attaching to processes 36

multithread safety and C++View 322

N

N upcast expr 309

n, see dnext command

name argument, build_struct_transform 308

name property 218, 221

name, information about 156

namespaces 135

TV:: 135

TV::GUI:: 135

using wildcards 135

Namespaces, Symbol 210

naming the host 349

nccq command-line option 337

nested subroutines, stepping out of 114

cieval variable in watchpoints 153

next, see dnext command

nexti, see dnexti command

ni, see dnexti command

nil, see dnexti command

niw, see dnexti command

nl, see dnext command

nlb command-line option 341

no_ask_on_dlopen command-line option 378

no_compiler_vars command-line option 337

no_control_c_quick_shutdown command-line option 337

no_dynamic command-line option 375

no_ent command-line option 339

no_global_types command-line option 340

no_gnu debuglink command-line option 340

no_gnu debuglink_checksum command-line option 340

no_ipv6_support command-line option 340

no_kcc_classes command-line option 341

no_message_queue command-line option 342

no_mqd command-line option 342

no_nptl_threads command-line option 343

no_parallel command-line option 343

no_startup_scripts command-line option 343

no_team command-line option 346

no_teamplus command-line option 346

no_user_threads command-line option 346
option 336, 347
nodeid property 194
nodes_allowed command-line option 350
nptl_threads command-line option 343
nptl_threads variable 282
NVIDIA. See CUDA
nw, see dnext command

O
OBJECT_SEARCH_MAPPINGS variable 250, 252
OBJECT_SEARCH_PATH variable 252
oldval variable in watchpoints 153
Open (or raise) process window at breakpoint checkbox 300
Open process window on error signal check box 300
open_cli_window_callback variable 283
opening shared libraries 66
option 350
options
tvdsvr
-callback 349
-serial 349
-server 349
-set_pw 349
-user_threads 347
ou, see dout command
oun, see dout command
ouw, see dout command

P
p, see dprint command
p/t expressions 123
P/T set information 123
panes, width 288
parallel backtrace data, displaying 48
parallel command-line option 343
parallel jobs
  displaying state of data using dcalltree command 48
parallel processes
  attaching to 35
  displaying process and thread state using dcalltree command 48
parallel program runtime library support 283
parallel runtime libraries 283
parallel variable 283
parallel_attach variable 283
parallel_configs variable 102, 415
parallel_stop variable 283
parallel_support.tvd file 415
passwords 351
  checking generated by tvdsvr 349
patch space 264
patch_area_base command-line option 344
patch_area_length command-line option 344
PATH environment variable for tvdsvr 349
pc property 215
picking up threads 277
pid command-line option 344
Plant in share group checkbox 253, 286
platform variable 284
plist, defined 138
pop_at_breakpoint variable 300
pop_on_error variable 300
popping Process Window on error variable 300
port 4142 350
port command-line option 350
port number 350
  for tvdsvr 349
  replacement 353
  searching 350
ports on host 349
post_scope 199
post_symbol 199
PowerPC
architecture 391
Blue Gene registers 393
floating-point registers 395
FPSCR register 395
  using the 397
FPSCR register, using 397
general registers 391
MSR register 394
pre_scope 199
pre_sym 199
preferences, setting defaults for 136
previous function state register Intel IA-64 402
print, see dprint command
printing expression values 117
printing registers 120
printing slices 118
printing variable values 117
proc file system problems 371
Procedure Linkage Table (PLT) 376
Process > Startup command 83
process barrier breakpoint, see barrier breakpoint
process command 192
process counts, see ptlist
process groups, see groups
process information, saving 52
process list element 250
process objects, creating new 103
process statistics 105
process width stepping behavior 142
process window position 298
process_load_callbacks variable 284
process, attaching to existing CLI dattach command 35
  continuing or halting execution on attach 35
process/thread sets, changing 79
processes
  attaching to 35, 102
  automatically acquiring 270
  automatically attaching
to 283
current status 137
destroyed when exiting
  CLI 164, 166
detaching from 63
holding 95
killing 97
properties 192
releasing 145
releasing control 63
restarting 126, 131
returning list of 187
starting 126, 131
terminating 97
processor status register Intel IA-64 400
program control groups, placing processes in 36
program stepping 141
program variable, changing value 33, 52, 95, 114, 129, 145, 147
programs, loading 103
PROMPT variable 252
prompting when screen is full 118
properties
  address 176
  clusterid 192
  continuation_sig 214
  count 190
  done 184
dpid 215
  duid 193, 215
  enabled 177
  enum_values 217
  executable 193
  expression 177, 184
  focus_threads 184
  heap_size 193
  held 193, 215
  hostname 193
  id 177, 185, 190, 193, 215, 217
  image_id 217
  image_ids 193
  initially_suspended_process 185
  language 177, 218
  length 177, 218
  line 177
  manager 215
  member_type 190
  member_type_values 190
  members 190
  name 218
  nodeid 194
  pc 215
  prototype 218
  rank 218
  result 185
  satisfaction_group 177
  share 177
  sp 215
  stack_size 194
  stack_vm_size 194
  state 194, 215
  state_values 194, 215
  status 185
  stop_when_done 177
  stop_when_hit 178
  struct_fields 218
  symbol 202
  syspid 194
  systid 215
  text_size 195
  threadcount 195
  threads 195
  type 190
  type_values 178, 190
  vm_size 195
properties verb
  actionpoint command 176, 180
  expr command 184
  group command 189
  process command 192
  thread command 214
  type command 217
prototype property 218
ptlist
  and dcalltree 49
  and dstatus 137
  and dwwhere group-by
    property 160
    defined 138
    example 140
PTSET variable 252
ptsets, see dptsets
pxdb command 376
pxdb64 command 376
Q
qnofullpath command-line option 358
qualifying symbol names 99
quit command 166
quotation marks 33
R
r command-line option 344
r, see drun command
raising errors 182
rank property 218
ranks, attaching to 35
raw memory display 73
read_delayed subcommand 201
read_symbols command 196
reading action points file 29
reading symbols 196, 259, 272, 273
rebind subcommand 201
reenabling action points 71
register stack configuration register
  Intel IA-64 401
registers
  Blue Gene PowerPC 393
  BlueGene PowerPC 393
  floating-point
    Intel-x86 387, 405
    PowerPC 395
    SPARC 410
general
    Intel-x86 386, 404
    PowerPC 391
    SPARC 409
  Intel-x86 FPCR 388, 406
  using the 389, 407
  Intel-x86 FPSCR 389, 390, 407
  Power FPSCR 395
  Power MSR 394
  PowerPC FPSCR 395
  using 397
  PowerPC FPSCR, using 397
PowerPC MSR 394
printing 120
SPARC FPSR 411
SPARC FPSR, using 412
SPARC PSR 410
registers, using in evaluations 44
regular expressions within name argument 308
release 249
releasing control 63
releasing processes and threads 38, 145
remapping keys 383
remote command-line option 344
remote debugging, tvdsvr command syntax 348
remote systems, debugging 47
removing aliases 173
group member 85
variables 146
worker threads 163
replacement characters 352
replacing tabs with spaces 254
replay, see dhistory command
rerun, see drerun command
resolve_final subcommand 201
resolve_next subcommand 201
respond 197
restart_threshold variable 285
restart, see drestart command
restarting processes 126, 131
restoring variables to default values 146
result property 185
resuming execution 54, 83, 97
returning error information 182
root path 254
of TotalView 254
Root Window position 299
routines, stepping out of 114
rr, see drerun command
RS/6000, compiling on 358
RTLD_GLOBAL 66
RTLD_LOCAL 66
RTLD_NOW 66, 67
run, see drun command running to an address 147
S
s command-line option 344
s, see dstep command
satisfaction set 39, 248, 260
satisfaction_group property 177
save_window_pipe_or_filename variable 285
saved position
Call Tree Window 297
CLI Window 297
Help Window 297
Image Browser Window 297
Memory Statistics Window 298
Message Queue Graph Window 298
Message Queue window 298
Modules Window 298
Process Window 298
Root Window 299
Thread Objects Window 299
Variable Window 299
saving action point information 29
saving action points 29
saving process information 52
scope 202
scope command 198
scope of action point 253
screen size 251
search dialog, remaining displayed 300
search path 250
mappings 253
setting 250, 252, 253
search_case_sensitive variable 285
search_port command-line option 345
search_path command-line option 345
search_port command-line option 350
searching case sensitive 285
wrapping 291
serial command-line option 345, 349, 350
serial line connection 350
server command-line option 349, 350
server launch command 352
server_launch_enabled variable 286
server_launch_string variable 286
server_launch_timeout variable 286
server_response_wait_timeout variable 286
servers, number of 353
sessions dsession 134
set verb
actionpoint command 176
group command 189
process command 192
thread command 214
type command 217
set_pw command-line option 349, 351
set_pws command-line option 351
set, see dset command
setting default preferences 136
setting lines between more prompts 251
setting terminal properties 172
setting variables 135
SGROUP variable 252
share groups, share group variable 252
share list element 251
share property 177
SHARE_ACTION_POINT variable 253
share_action_point variable 286
share_in_group flag 252
shared libraries 375
closing 180
defered reading 196
HP-UX 376
information about 180
loading symbols from 276
managing 180
manually loading 66
reading deferred symbols 196
reading symbols 259, 272, 273
shared_data_filters 294
shm command-line option 345
show_startup_parameters 301
show_sys_thread_id variable 301
showing current status 137
showing Fortran compiler variables 265
SIGINT 279
signal_handling_mode command-line option 345
signal_handling_mode variable 287
signals, handling in TotalView 345
SILENT state 255
single process server launch 286
single_click_dive_enabled variable 301
sl, see dstep command
slices, printing 118
SLURM, control_c_quick_shutdown variable 266, 337
source code, displaying 99
source_pane_tab_width variable 288
source_process_startup 284
source_process_startup command 200
SOURCE_SEARCH_MAPPINGS variable 253
SOURCE_SEARCH_PATH variable 253
sourcing tvd files 200
sp property 215
spaces simulating tabs 254
SPARC architecture 409
floating-point registers 410
FPSR register 411
using 412
general registers 409
PSR register 410
spell_correction variable 288
spurs support for 23
st, see dstatus command
stack frame 99
moving down through 69
see also call stack 160
stack frame, see also call stack 17
stack memory 105
stack movements 150
stack_size property 194
stack_tracequalification_level variable 288
stack_vm size 106
stack_vm_size property 194
stack, unwinding 76
starting a process 126, 131
startup command 83
startup file 307
start-up file, tvdinit.tvd 24
startup options
-no_startup_scripts 343
start-up file, tvdinit.tvd
startup options
stop group breakpoint 44
stop_all property 260
STOP_ALL variable 43, 248, 253
stop_all variable 288
stop_group flag 253
stop_when_done command-line option 248
stop_when_done property 177, 260
stop_when_hit property 178
stopped process, responding to resume commands 39
stopping execution 90
stopping group members flag 253
stopping the control group 289
string length format 270
strings, assigning values to 33
stripped copy 362
struct_fields property 218
structure definitions in KCC 280
structure transformations 306
structures, transforming 308
stty command 172
suffixes variable 289
SunOS 5
/proc file system 371
key remapping 383
linking to dbfork library 366
swap space 374
sw, see dstep command
swap command 374
swap space 372
AIX 373
HP-UX 372
Linux 373
SunOS 374
swap command 373
symbol command 201
symbol name qualification 99
symbols
namespaces 210
properties 202
reading 272, 273
symbols, interpreting 33
syspid property 194
system variables, see CLI variables
systid property 215
T
tab processing 100
TAB_WIDTH variable 100, 254
tabs, replacing with spaces 254
target processes 90
terminating 97
target property 218
TCL library component search path 254
team command-line option 346
teamplus command-line option 346
templates
using with C++View 317
temporarily changing focus 79
terminal properties, setting 172
terminating debugging session 164
terminating processes 97
text size 105
text_size property 195
thread barrier breakpoint, see barrier breakpoint
thread command 214
thread counts, see ptlist
thread groups, see groups
thread list element 251
thread objects window position 299
thread of interest 142, 147
thread state display via ptlist 138
thread width stepping behavior 142
threadcount property 195
threads
barriers 40
creating 83
current status 137
destroyed when exiting CLI 164, 166
getting properties 214
holding 39, 95
list variable 254
picking up 277
property 195
releasing 145
returning list of 188
setting properties 214
THREADS variable 254
timeplus command-line option 346
toolbar_style variable 301
Tools > Command Line command 283
Tools > Dynamic Libraries command 66
Tools > Evaluate window 76
tooltips_enabled variable 301
totalview command 334
options 335
synopsis 334
syntax and use 334
TotalView Debugger Server 35
TotalView executable 254
TotalView GUI version 302
TotalView version 290
totalview_jobid variable 354
TOTALVIEW_ROOT_PATH variable 254
TOTALVIEW_TCLLIB_PATH command 313
TOTALVIEW_TCLLIB_PATH variable 254
TOTALVIEW_VERSION variable 254
TotalView.breakpoints file 29
totalview/lib_cache subdirectory 47
transformations
of types using C++View 314
using type 312
why type 304
transforming classes 308
transforming structures 306, 308
triggering breakpoints 44
troubleshooting 6
ttf variable 289
tv_data_display.h, and API for using C++View 315
TV_ttf_display_type function with C++View 324
TV_ttf_display_type function, writing for C++View 315
TV:: namespace 135
TV::actionpoint command 176
TV::dll command 67, 180
TV::dll_read_loader_symbols_only variable 196
TV::dll_read_no_symbols variable 196
TV::errorCodes command 182
TV::expr command 117, 184
TV::focus_groups command 186
TV::focus_processes command 187
TV::focus_threads command 188
TV::group command 189
TV::GUI:: namespace 135
TV::hex2dec command 191
TV::parallel_configs variable 102
TV::process command 192
TV::read_symbols command 196
TV::respond command 197
TV::scope command 198
TV::source_process_startup command 200
TV::symbol command 201
TV::thread command 214
TV::ttf variable 313
TV::type command 217
TV::type_transformation command 220
tvd files 200
TVD.breakpoints file 259
tvdinit.tvd start-up file 24
TVDSVRLAUNCHCMD environment variable 352
tvhome command-line option 346
tvscript 230
action point API 241
create_actionpoint command-line option 235
display_specifiers command-line option 238
evnet actions 236
Event API 242
event_action command-line option 235
event_action event types 235
example 239
example script file 243
external script files 241
logging functions API 241
maxruntime memory sub-option 239
memory debugging command-line options 238
memory debugging command-line sub-options 238
MPI programs 230
options 235
process functions API 241
script_file command-line option 239
script_log_filename command-line option 239
script_summary_log_filename command-line option 239
source location expression syntax 242
thread functions API 241
tvscript syntax for Blue Gene 231
tvscript syntax for Blue Gene/Q LoadLeveler job manager 233
tvscript syntax for Blue Gene/Q with ANL’s Cobalt 232
tvscript syntax for Blue Gene/Q with SLURM 232
tvscript syntax for Cray Xeon Phi 233, 234
type command 217
type names 277
type property 178, 190, 218
type transformation variable 289
Type Transformations why 304
type transformations activating 313
creating 303
expressions 308
preference 305
regular expressions 308
structures 306
using 312
using C++View 314
type_callback 221
type_index 202
type_transformation command 220
type_transformation_description property 221
type_values property 178, 190, 218
U
u, see dup command
uhp, see dunhold command
uht, see dunhold command
ui_font variable 301
ui_font_family variable 302
ui_font_size variable 302
un, see duntil command
unalias command 173
unconditional watchpoints 153
undefined symbols 66
unhold, see dunhold command
unl, see duntil command
unset, see dunset command
until, see duntil command
unw, see duntil command
unwinding the stack 76
up, see dup command
upper_bounds_callback 222
user_threads command-line option 336, 347
user_threads variable 290
user-defined commands 24
user-level (M:N) thread packages 290
using quotation marks 33
using type transformations 312
using_color variable 302
using_text_color variable 302
using_title_color variable 302
V
validate_callback 221
value for newly created action points 253
values, printing 117
Variable Window position 299
variables
assigning command output to 26
casting 120
changing values 33, 52, 95, 114, 129, 145, 147
default value for 134, 135
printing 117
removing 146
setting 135
watched 154
watching 153
vector transformation 304
VERBOSE variable 255
verbosity command-line option 347, 351
verbosity setting replacement character 354
version variable 290, 302
version, TotalView 254
vfork() calling 337
catching 270
viewing CLI variables 134, 135
visualizer_launch_enabled variable 291
visualizer_launch_string variable 291
visualizer_max_rank variable 291
vm_size property 195
vm_size size 106

W
w, see dwhere command
wa, see dwatch command
wait, see dwait command
walk subcommand 199
warn_step_throw variable 291
warning state 255
watch, see dwatch command
watchpoints 153
$newval 153
$oldval 153
conditional 153
information not saved 29
length of 154
supported systems 154
-WG,-cmpo=i option 359
WGROUP variable 255
wh, see dwhat command
what, see dwhat command
When barrier done, stop value 248
When barrier hit, stop value 248
where, see dwhere command

why type transformations 304
window position, forcing 296
worker group list variable 296
worker threads 255
inserting 163
removing 163
worker, see dworker command
workers list element 251
working_directory command-line option 351
wot, see dworker command
wrap_on_search variable 291, 292

X
xterm_name command-line option 347