

TotalView User Guide

Version 2024.3
September, 2024

PERFORCE

www.perforce.com



© 2024 Perforce Software, Inc. All rights reserved.
© 2007-2024 by Rogue Wave Software, Inc., a Perforce company ("Rogue Wave"). All rights reserved.
© 1998–2007 by Etnus LLC. All rights reserved.
© 1996–1998 by Dolphin Interconnect Solutions, Inc.
© 1993–1996 by BBN Systems and Technologies, a division of BBN Corporation.

Perforce and other identified trademarks are the property of Perforce Software, Inc., or one of its affiliates. Such trademarks are claimed and/or registered in the U.S. and other countries and regions. All third-party trademarks are the property of their respective holders. References to third-party trademarks do not imply endorsement or sponsorship of any products or services by the trademark holder. Contact Perforce Software, Inc., for further details.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of Rogue Wave.

Perforce has prepared this manual for the exclusive use of its customers, personnel, and licensees. The information in this manual is subject to change without notice, and should not be construed as a commitment by Perforce. Perforce assumes no responsibility for any errors that appear in this document.

TotalView and TotalView Technologies are registered trademarks of Rogue Wave. TVD is a trademark of Rogue Wave.

Perforce uses a modified version of the Microline widget library. Under the terms of its license, you are entitled to use these modifications. The source code is available at <https://rwkbp.makekb.com/>.

All other brand names are the trademarks of their respective holders.

ACKNOWLEDGMENTS

Use of the Documentation and implementation of any of its processes or techniques are the sole responsibility of the client, and Perforce 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 MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THE DOCUMENTATION. THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. ROGUE WAVE 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 PERFORCE 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.

TotalView by Perforce
<http://totalview.io>

Contents

Part 1: An Introduction to TotalView	1
Getting Started	
Introducing TotalView	3
An Initial Look at the Interface	4
Customizing the Interface.....	4
Preferences	4
Resizing	5
Drawers	5
Undocking and Docking	5
A Tour of the Interface.....	6
Central Area	6
Toolbars.....	12
Processes and Threads View	12
Call Stack View and Local Variables View	14
Data View	15
Lookup View	15
Action Points, CLI, and Logger Views	16
Input/Output View	17
Help.....	18
Starting TotalView and Creating a Debugging Session.....	21
Debugging Commands	23
Diving on Program Elements	24
Creating and Managing Sessions	
Setting up Debugging Sessions.....	28
Loading Programs from the Session Editor	28
Starting a Debugging Session	29
Debug a Program	31
Debug a Parallel Program	32
Attach to Process	35
Debug a Core or Replay Recording File.....	39
Load a Recent Session	41
Editing a Previous Session	41
Loading Programs Using the CLI	41
Options and Program Arguments	43
Debug Options.....	43

Program Environment	44
Working Directory	44
Environment Variables for the Program	44
Standard Input and Output	45
Modifying Arguments in an Open Session	46
Managing Sessions	49
Starting a Session from your Shell	53
Starting TotalView on a Script	54
Basic Debugging	
Program Load and Navigation	57
Load the Program to Debug	57
Initial Display	59
Program Navigation	61
Stepping and Executing	62
Simple Stepping	62
Setting and Running to a Breakpoint (Action Point)	65
Set and Control Breakpoints	65
Run Your Program and Observe the Call Stack	67
Examining Data	69
Viewing Variables in the Local Variables View	69
Viewing Variables in the Data View	71
Watching Data Values Update	72
Moving On	76
Program Navigation	
Navigating from within the Source Pane	78
Highlighting a String and the Find Function	79
The Lookup File or Function View	81
The Documents View	83
Part 2: Debugging Tools and Tasks	84
Setting and Managing Action Points (Breakpoints)	
About Action Points	86
Breakpoints	88
Setting Source-Level Breakpoints	88
Sliding Breakpoints	90
Breakpoints at a Specific Location	91
Setting Machine-Level Action Points	92

Pending Breakpoints	93
Pending Breakpoints on a Function	93
Pending Breakpoints on a Line Number	94
Conflicting Breakpoints	95
Breakpoints at Execution	95
Modifying a Breakpoint	95
Setting Breakpoints When Using the fork()/execve() Functions	97
Debugging Processes That Call the fork() Function	97
Debugging Processes that Call the execve() Function	98
Example: Multi-process Breakpoint	98
Evalpoints	100
Setting an Evalpoint	101
Creating a Pending Evalpoint	103
Modifying an Evalpoint	105
Creating Conditional Breakpoints	105
Patching Programs	106
Branching Around Code	107
Adding a Function Call	107
Correcting Code	108
Using Programming Language Constructs	108
Watchpoints	110
Creating Watchpoints	111
Displaying, Deleting, or Disabling Watchpoints	112
Modifying Watchpoints	113
Watching Memory	114
Triggering Watchpoints	115
Using Multiple Watchpoints	115
Performance Impact of Copying Previous Data Values	116
Using Watchpoint Expressions	116
Using Watchpoints on Different Architectures	117
Barrier Points	120
About Barrier Breakpoint States	120
Setting a Barrier Breakpoint	121
Creating a Satisfaction Set	123
Hitting a Barrier Point	124
Releasing Processes from Barrier Points	125
Changing Settings and Disabling a Barrier Point	125
Using Barrier Points	125
Barrier Point Illustration	126
Controlling an Action Point's Width	128
About an Action Point's Width: Group, Process or Thread	128
Setting the Action Point's Width	128
Action Point Width and Process/Thread State	129

Managing and Diving on Action Points	133
Sorting	133
Diving	134
Deleting, Disabling, and Suppressing	134
Saving and Loading Action Points	137
More on Action Points Using the CLI	139
Breakpoints	140
Evalpoints	140
Watchpoints	141
Barrier Points	143
Saving Action Points to a File Using the CLI	144
Suppressing and Unsuppressing Action Points	145
Examining and Editing Data	
Viewing Data in TotalView	146
About Expressions	147
Using C++	148
The Call Stack, Local Variables, and Registers Views	150
The Call Stack View	150
The Local Variables View	151
The Registers View	154
Edit or Cast a Register	155
Viewing Call Stack Data	155
Viewing Data in Fortran	157
Viewing Modules and Their Data	157
Common Blocks	159
Fortran 90 User-Defined Types	159
Fortran 90 Deferred Shape Array Types	160
Fortran 90 Pointer Types	161
Fortran Parameters	162
The Data View	164
Adding Variables to the Data View	164
Add to the Data View from the Local Variables View	165
Move a Variable from the Source View to the Data View	166
Create a New Expression from within the Data View	167
Diving on Variables	170
Working with Complex Variables in the Data View	170
Viewing Elements of Complex Variables	171
Diving on Complex Variables	172
Editing an Expression	174
Dereferencing a Pointer	174
Changing the Value of Data	174
Casting to Another Type	175

Displaying Arrays	178
Viewing Individual Elements in an Array of Structures	179
The Dive In All Command	179
Customizing the Data View	184
The Data View Drawer	185
The Array View	186
Adding Arrays to the Array View	186
The Array View Toolbar	187
Array Statistics and Visualization	187
Viewing Array Statistics	188
Visualizing Array Data	190
Configuring Arrays	195
Slicing Arrays	196
Casting to Another Type in the Array View	200
C++ STL Type Transformations	201
Supported C++ STL Type Transformations	203
Struct TTF "\$elide_" Members	204
Iterator Type Transformations	205
TTFs and TotalView Expressions	205
Type Transformations CLI Commands	206
Controlling Type Transformations	206
Querying Type Transformations	207
STL TTF Troubleshooting	207
Using the CLI to Examine Data	209
Changing the Display of Data	209
Displaying Variables	209
The Processes & Threads View	
Processes & Threads View Basics	212
Customize the Display	214
Customizing the Processes & Threads View Pane	215
Cycling Through Threads and Processes	217
The Processes & Threads View in Relation to Other Views	220
Displaying a Thread Name	221
Thread Names in the UI	221
Thread Properties	223
Thread Options on dstatus	223
Process and Thread Attributes	225
Debugging Python	
Overview	228

Python Debugging Requirements	229
Python Version	229
Limitations and Extensions:	229
Starting a Python Debugging Session	231
Debugging Python and C/C++ with TotalView	233
Transforming the Stack	234
Viewing and Comparing Python and C/C++ Variables	236
Leveraging Other Debugging Technologies for Python Debugging	238
Supported Python Extension Technologies for Stack Transformations	239
Using the Command Line Interface (CLI)	
Access to the CLI	242
Introduction to the CLI	244
About the CLI and Tcl	245
Integration of the CLI and the UI	245
Invoking CLI Commands	246
Starting the CLI in a Terminal Window	247
Startup Example	247
Starting Your Program	248
About CLI Output	250
‘more’ Processing	251
Using Command Arguments	252
Using Namespaces	253
About the CLI Prompt	254
Using Built-in and Group Aliases	255
How Parallelism Affects Behavior	256
Types of IDs	257
Controlling Program Execution Using CLI Commands	258
Advancing Program Execution	258
Using Action Points	259
Examples of Using the CLI	260
Setting the CLI EXECUTABLE_PATH Variable	260
Initializing an Array Slice	261
Printing an Array Slice	262
Writing an Array Variable to a File	263
Automatically Setting Breakpoints	263

Reverse Connections

About Reverse Connections	267
Reverse Connection Environment Variables	269
TV_REVERSE_CONNECT_DIR	269
TV_CONNECT_OPTIONS	269
Starting a Reverse Connect Session	271
Listening for Reverse Connections	272
Reverse Connect Examples	273
CLI Example	273
MPI Batch Script Example	273
Troubleshooting Reverse Connections	275
Stale Files in the Reverse Connect Directory	275
Directory Permissions	275
User ID Issues	275
Reverse Connect Directory Environment Variable	275

Preferences

About Preferences	277
Action Points	279
Display Settings	280
Tool Bar	282
Search Path	283
Parallel Configuration	286
Remote Connection Settings	288
Signal Actions	289
Default Signal Handling	291

Part 3: Parallel Debugging

About Parallel Debugging in TotalView

Parallel Program Execution Models	295
Viewing Process and Thread State	296
Controlling Program Execution	297
TotalView Groups	298
Synchronizing Execution with Barrier Points	299
Configuring TotalView for Parallel Debugging	300

Setting Up Parallel Sessions

Parallel Program Setup in the UI	303
Non-MPI Program Setup	304
The SLURM Resource Manager	304
Cray XT/XE/XK/XC Applications	305
Starting TotalView on Cray	305
Support for Cray Abnormal Termination Processing (ATP)	307
Special Requirements for Using ReplayEngine	307
Global Arrays Applications (Classic UI Only)	307
Shared Memory (SHMEM) Code	309
UPC Programs	310
Invoking TotalView	310
Viewing Shared Objects (Classic UI Only)	310
Displaying Pointer to Shared Variables (Classic UI Only)	312
CoArray Fortran (CAF) Programs	314
Invoking TotalView	314
Viewing CAF Programs (Classic UI Only)	314
Using CLI with CAF	315
MPI Program Setup	317
MPICH Applications	317
Starting TotalView on an MPICH Job	318
Attaching to an MPICH Job	319
Using MPICH P4 procgroup Files	320
MPICH2 Applications	321
Downloading and Configuring MPICH2	321
Starting TotalView Debugging on an MPICH2 Hydra Job	322
Starting TotalView Debugging on an MPICH2 MPD Job	322
Cray MPI Applications	323
IBM MPI Parallel Environment (PE) Applications	323
Preparing to Debug a PE -Application	324
Starting TotalView on a PE Program	324
Setting Breakpoints	325
Starting Parallel Tasks	325
Attaching to a PE Job	326
Open MPI Applications	327
QSW RMS Applications	327
Starting TotalView on an RMS Job	327
Attaching to an RMS Job	328
SGI MPI Applications	328
Starting TotalView on an SGI MPI Job	329
Attaching to an SGI MPI Job	329
Using ReplayEngine with SGI MPI	330
Sun MPI Applications	330
Attaching to a Sun MPI Job	331

Troubleshooting MPI Startup	331
Using ReplayEngine with Infiniband MPIs	332
MPI Startup Customizations	334
Customizing Your Parallel Configuration	334
Example Parallel Configuration Definitions	335
Debugging OpenMP Applications	
OpenMP and the OMPD API	341
OMP Requirements	341
OpenMP Setup and Configuration	342
Workarounds for Existing DWARF Debugging Problems	342
Compiling and Linking for OMPD Support	343
Enabling OpenMP Debugging	343
Enabling Stack Filtering	344
Running Your Program	346
The Call Stack	346
The OpenMP View	348
Hybrid Programming: Combining OpenMP with MPI	354
Controlling fork, vfork, and execve Handling	
The exec_handling and fork_handling Command Options and State Variables ..	358
Exec Handling	359
Fork Handling	359
Example	359
Group, Process, and Thread Control	
Overview	361
Groups in TotalView	363
What Is a Group?	363
Types of Groups Created by TotalView	364
How TotalView Creates Groups	364
Groups Created When a Program Calls fork()/exec()	365
Groups Created for MPI Programs	365
Groups Created for CUDA Programs	366
Executing a Single Share Group	367
Single Stepping While Focused on a Share Group	369
Arenas and P/T Sets	371
Arena Specifiers in a P/T Set	371
P/T Set and Arena Identifier Syntax	371
Process and Thread Width Specifiers in a P/T Set	372
Group Specifiers in P/T Sets	374
Identifying a Group Using a Letter	374

- Identifying a Group Using a Number 375
- Identifying a Group Using a Name 375
- Arena Specifier Examples 375
 - Naming Incomplete Arenas 375
 - Combining Arena and Group Specifiers 376
 - Naming Lists with Inconsistent Widths 378
 - Merging Focuses 379
- Using P/T Set Operators 379
- Setting and Creating Custom Groups 381
 - Using the g Specifier: An Extended Example 382
- Changing the P/T Using the dfocus Command 384
- Stepping and Program Execution 386
 - Individual Execution Commands 387
 - Executing at Group Width 388
 - Executing at Process Width 389
 - Executing at Thread Width 389
 - Synchronizing Processes and Threads 390
 - Holding and Releasing Processes and Threads 390
 - Using Run To and duntil 391
- CLI Stepping Examples 392
 - Execution Commands Using the “all” Arena Specifier 393

Scalability in HPC Computing Environments

- Configuring TotalView for Scalability 396
 - Disable User-Thread Debugging 396
 - Tune Dynamic Library Load Processing 396
 - Filtering dlopen Events 396
 - Handling dlopen Events in Parallel 397
- MRNet 398
- TotalView Infrastructure Models 398
- Using MRNet with TotalView 400
 - General Use 400
 - Using MRNet on Cray Computers 404

Part 4: Accessing TotalView Remotely 408

TotalView Remote Connections

- About Remote Connections 410
 - Connecting Remotely From the TotalView Remote Client 410
 - Connecting Remotely From a TotalView Debugger Installation 412
- Configuring a Remote Connection 413
- Debugging on a Remote Connection 416

TotalView Remote Display

Remote Display Supported Platforms	419
Remote Display Components	420
Installing the Client	421
Installing on Linux	421
Installing on Microsoft Windows	421
Installing on macOS	421
Client Session Basics	423
Working on the Remote Host	426
Advanced Options	427
Naming Intermediate Hosts	429
Submitting a Job to a Batch Queuing System	430
Setting Up Your Systems and Security	432
Session Profile Management	433
Batch Scripts	435
tv_PBS.csh Script	435
tv_LoadLeveler.csh Script	436

Part 5: GPU Debugging

Debugging CUDA Programs

NVIDIA CUDA Debugging Overview	440
Installing the CUDA SDK Tool Chain	440
Directive-Based Accelerator Programming Languages	441
CUDA Debugging Model and Unified Display	442
The TotalView CUDA Debugging Model	442
Pending and Sliding Breakpoints	444
Unified Source View and Breakpoint Display	444
CUDA Debugging Tutorial	446
Compiling for Debugging	447
Compiling for Fermi	447
Compiling for Fermi and Tesla	447
Compiling for Kepler	447
Compiling for Pascal	447
Compiling for Volta	448
Starting a TotalView CUDA Session	448
Controlling Execution	449
Viewing GPU Threads	449

Single-Stepping GPU Code	451
Halting a Running Application	452
Displaying CUDA Program Elements	452
GPU Assembler Display	452
GPU Variable and Data Display	452
Managed Memory Variables	454
About Managed Memory	454
How TotalView Displays Managed Variables	454
CUDA Built-In Runtime Variables	455
Type Casting	456
PTX Registers	458
The GPU Status View	459
The GPU Status View Focus Options	460
Configuring the GPU Status View	462
Enabling CUDA Memory Checker Feature	469
GPU Core Dump Support	471
GPU Error Reporting	471
CUDA Problems and Limitations	474
Hangs or Initialization Failures	475
CUDA and ReplayEngine	476
Sample CUDA Program	477
Debugging AMD ROCm Programs	
AMD ROCm Debugging Overview	482
Installing the AMD Tool Chain	482
AMD ROCm Debugging Model and Unified Display	483
The TotalView AMD ROCm Debugging Model	483
Disabling Deferred GPU Image Loading	485
Pending and Sliding Breakpoints	485
Unified Source View and Breakpoint Display	486
AMD ROCm Debugging Tutorial	488
Compiling for Debugging	488
Starting a TotalView ROCm Session	488
Controlling Execution	489
Viewing GPU Threads	490
Single-Stepping GPU Code	492
Halting a Running Application	493
Displaying ROCm Program Elements	493
GPU Variable and Data Display	493
ROCm Built-In Runtime Variables	496
GPU Error Reporting	497

AMD ROCm Problems and Limitations	498
Hangs or Initialization Failures	499
AMD GPU Debugging and ReplayEngine	500
Sample HIP Program	501
Part 6: Memory Debugging	505
About TotalView Memory Debugging	
Debugging Memory in TotalView	507
About Program Memory	508
How TotalView Intercepts Memory Data	512
Your Program's Data	514
The Data Section	514
The Stack	514
The Heap	519
Finding Heap Allocation Problems	519
Finding Heap Deallocation Problems	519
realloc() Problems	520
Memory Leaks	520
Running a Memory Debugging Session	
Starting Memory Debugging in TotalView	523
Memory Leak Detection	525
Using the Leak Report	526
Updating the Leak Report	528
MPI Programs and Leak Reports	528
Memory Heap Reports	531
Using the Heap Report	532
Updating the Heap Report	533
Corrupt Guard Block Reports	534
Memory Event Reports	537
The Memory Event Report View	538
Memory Block Notification	541
Memory Debugging Options	543
Option: Painting Memory	544
Enabling and Configuring Painting	545
Example: Viewing Painted Memory	546
Option: Hoarding Memory Blocks	546
Enabling and Configuring Hoarded Memory	548

Example: Hoarding Memory	549
Option: Guarding Allocated Memory	549
Enabling and Configuring Guard Blocks	550
Example: Viewing a Guard Corruption Event Report	550
Dangling Pointer Problems	554
Dangling Pointers in the Local Variables and Data Views	554
Memory Scripting	
display_specifiers Command-Line Option	556
event_action Command-Line Option	557
Other Command Line Options	558
memscript Example	559
Preparing Programs for Memory Debugging	
Compiling Programs for Memory Debugging	561
Linking Your Application with the HIA	562
Using env to Insert the HIA	565
Installing tvheap_mr.a on AIX	567
LIBPATH and Linking	567
Using TotalView in Selected Environments	569
MPICH	569
IBM PE	569
Mac OS	570
Background	570
Calls to system() on Mac OS	570
Setting the Environment Variable TV_MACOS_SYSTEM	570
Linux	571
dlopen and RTLD_DEEPCBIND	571
Part 7: Appendices	574
Appendix A More on Expressions	575
Calling Functions: Problems and Issues	576
Using Built-in Variables and Statements	577
Using TotalView Variables	577
Using Built-In Statements	578
Using Programming Language Elements	580
Using C and C++	580
Using Fortran	581
Fortran Statements	581
Fortran Intrinsics	582

Appendix B	Compiling for Debugging	584
	Compiling with Debugging Symbols	585
	Maintaining Debug Information Separate from an Executable	587
	Controlling Separate Debug Files	588
	Searching for the Debug Files	589
Appendix C	Platform-Specific Topics	590
	Swap Space	591
	Shared Libraries	591
	Changing Linkage Table Entries and LD_BIND_NOW	592
	Linking with the dbfork Library	593
	Linux or Mac OSX	593
Appendix D	Resources	594
	Classic TotalView Documentation	595
	Conventions	596
	Contacting Us	597
Appendix E	Open Source Software Notice	598
Appendix F	TotalView Glossary	599
Index	606

PART I An Introduction to TotalView

- **Getting Started**

An introduction to the primary features and interface of TotalView.

- **Creating and Managing Sessions**

How to create a new session or load a previous session.

- **Basic Debugging**

A tutorial based on a shipped example that illustrates basic debugging tasks.

- **Program Navigation**

Finding, searching, and navigation

Getting Started

- Introducing TotalView
- An Initial Look at the Interface
- Starting TotalView and Creating a Debugging Session

Introducing TotalView

TotalView features the next generation user interface of its debugger, supported on multiple platforms (See [TotalView Supported Platforms](#) for specifics.)

TotalView's new user interface continues to incorporate new features and functionality from the classic version of the UI in each release. See the release notes for detail. All TotalView functionality is fully available through the Command Line Interface (CLI) even if a feature has not yet been added to the new UI.

NOTE: For overviews, tutorials, and whitepapers on using the various features of TotalView, see the [TotalView website](#).

TotalView incorporates ReplayEngine technology. With this feature engaged, you can go backwards in the debugging session to find, for example, where an obviously incorrect variable value went wrong.

TotalView supports C++11 features for the GNU compiler, including support for lambdas, transformations for smart pointers, auto types, R-Value references, range-based loops, strongly-typed enums, initializer lists, user defined literals, and transformations for many of the containers such as array, forward_list, tuple and others.

UI Preferences

To launch the classic UI if necessary:

- **To change the default:**

Change the default Display preference under **File > Preferences > Display**, or

- **To launch the classic UI for a single instance of TotalView:**

Add the `-classicUI` switch after the `totalview` command, for example:

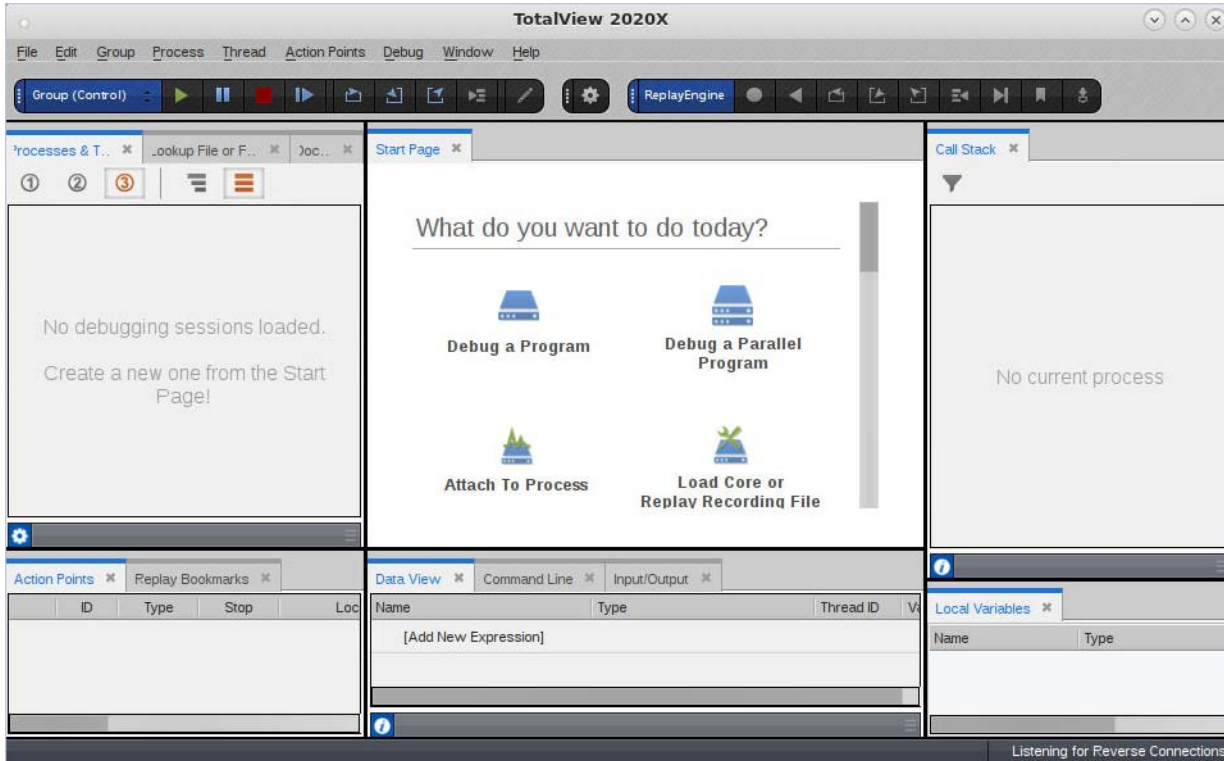
```
totalview -classicUI
```

For information on contacting Perforce, conventions used in the documentation, and documentation for the Classic UI, see *** 'Resources' on page 594 ***.

An Initial Look at the Interface

Starting TotalView without arguments launches the main screen:

Figure 1, The Initial Interface



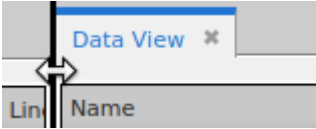
Customizing the Interface

Preferences

The Settings toolbar , when selected, displays the Preferences dialog. You can also select **File | Preferences**. For detail, see [Preferences](#).

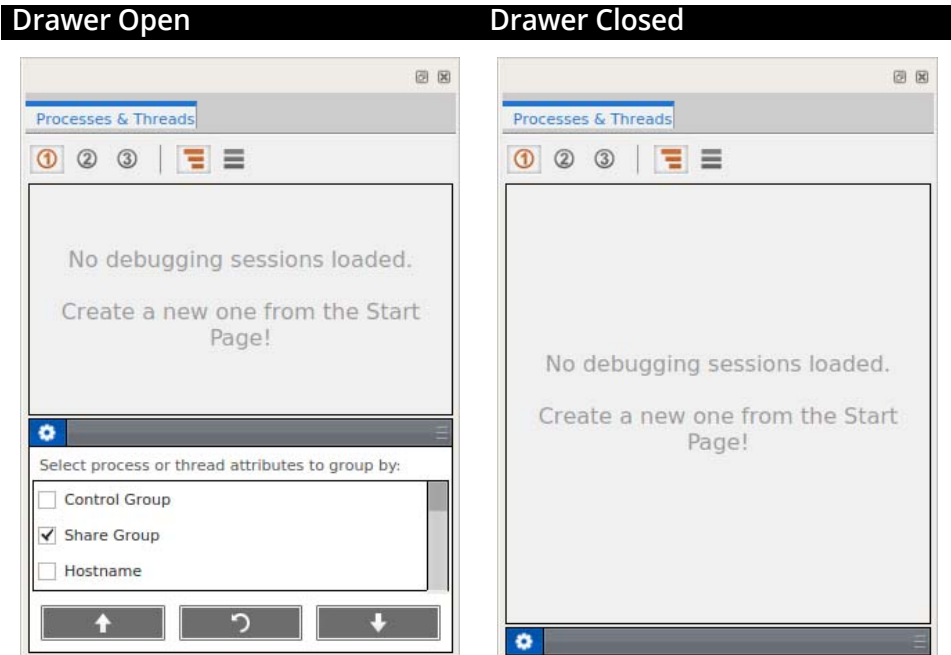
Resizing

To resize windows, hover your cursor over any dark dividing line between sections to display a two-way arrow that moves that boundary either up and down, or left and right.



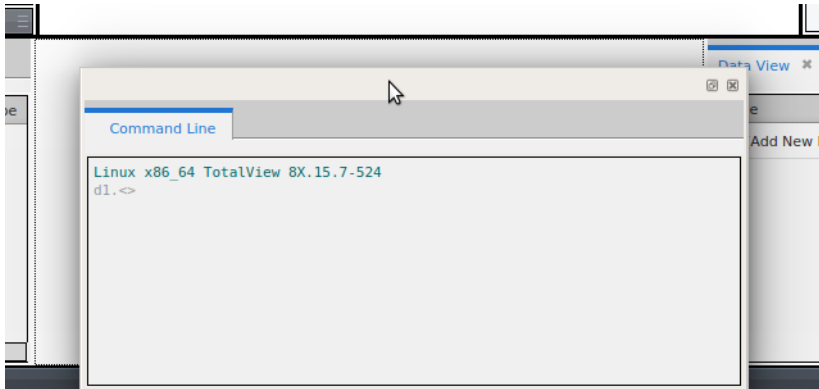
Drawers


Within certain views, you can display or hide a drawer, indicated by a dark gray banner that turns lighter gray at cursor hover. Double-click to close the drawer so only the banner is displayed; double-click again to re-open it. Click and drag the banner to move the banner up or down to resize the areas within the view.



Undocking and Docking

All views can be undocked into a separate, floating window by clicking on the top banner, dragging it a short distance, and releasing the mouse button. To dock the view elsewhere, click again and drag it to another location, wait for a blank area to display, then release the mouse button.

Figure 2, Docking a View in a New Location

To return a floating window to its default position, double-click on its banner or click the reattach icon  in the top-right corner next to the close (X) icon.

If you close the view, you can restore it using the Window | Views menu, or the context menu available by right-clicking in the toolbar area.

A Tour of the Interface

Here we introduce the main views that make up the interface. If a view is not visible, restore it through the Preferences dialog, the Window | Views menu, or the context menu available by right-clicking in the toolbar area.

Central Area

When you first start up TotalView, the central area contains either just the Start Page, or the Start Page and a Source view if you started TotalView with an executable name argument. This area is reserved for displaying the Start Page, Source views and Assembler views of code, the Help view, and other debug status and data specific views associated with your debugging session.

Central area views cannot be re-docked into the side or bottom secondary view areas, but they can be re-docked within the central area to create their own optimal debugging layout, such as a side-by-side layout.

Figure 3, Source View and Start Page in the central area

```

1092 #endif
1093 if (whoops)
1094 {
1095     printf("pthread_create failed; result=%d, errno=%d\n", whoops, errno);
1096     exit(1);
1097 }
1098 thread_ptids[total_threads++] = new_tid;
1099 printf ("%d: Spun off %ld.\n", (int)(getpid()), (long)new_tid);
1100
1101 forker (fork_count);           /* Never returns */
1102 } /* fork_wrapper */
1103
1104 /*****
1105
1106 int main (int argc, char **argv)
1107 {
1108     int fork_count = 0;
1109     int args_ok = 1;
1110     int arg_count = 1;
1111     char *arg;
1112     pthread_mutexattr_t mattr;
1113
1114     signal (SIGFPE, sig_fpe_handler);
1115     signal (SIGHUP, (void*)(int))sig_hup_handler);
1116
1117 #ifndef _linux
1118     /* The linux implementation of pthreads uses these signals, so we'd better not */
1119     signal (SIGUSR1, (void*)(int))sig_usr1_handler);
1120     signal (SIGUSR2, (void*)(int))sig_usr2_handler);
1121 #endif
1122
1123 #if defined ( _Lynx)
1124

```

The Source View

Viewing the Program Counter

- In normal debugging mode**, the diamond cursor and yellow highlighting identify the Program Counter (PC), i.e. the code location of the debugger. Clicking another line result in a blue highlight, indicating the target line if you use the Run To command. (There is no guarantee that the thread of focus will arrive at that line, of course, if it hits a breakpoint first, or never executes the line.)

```

674     bar = *foo;
675     *foo = bar + 1;
676 }
677 for (;;)
678 {
679     struct timeval timeout;
680     wait_a_while (&timeout);
681     if (verbose)
682         printf ("Thread %ld woke up in Snore()\n", (long)(pthread_self()));
683

```

- In Replay mode**, orange highlighting replaces yellow to identify where ReplayEngine is within the code. The red triangle shows the "Live" location, i.e., the last line executed. Once the PC hits the live location, it shifts from replay mode back to record mode.

```

576     timeout->tv_sec = sleep_time + get_random_time (vary_sleep);
577     timeout->tv_usec = sleep_time_usec;
578     YIELD();
579     errno = 0;
580     result = select (0, 0, 0, 0, timeout);

```

Source view actions

- **Create breakpoints** by clicking on bold line numbers in the gutter.

```

678   for (;;)
679   {
680       struct timeval timeout;
681       wait_a_while (&timeout);
682       if (verbose)
683           printf ("Thread %ld woke up in Snore()\n", (long)(pthread_self()));

```

- **See variable information** by hovering over a variable name.

```

680       struct timeval timeout;
681       wait_a_while (&timeout);
682       if (verbose)
683           printf ("Thread %ld woke up in Snore()\n", (long)(pthread_self()));
684       if (use_mut)
685       {

```

- **See function information** by hovering over function names.

```

680       struct timeval timeout;
681       wait_a_while (&timeout);
682       if (verbose)
683           printf ("Thread %ld woke up in Snore()\n", (long)(pthread_self()));
684       if (use_mut)
685       {

```

- **Search for text strings** with the Find function.

```

770
771     wait_a_while (&timeout);
772     pthread_rwlock_unlock (&rwlock);
773 #endif
774 }
775

```

Find: wait_a_while 8 matches

If you highlight the function name and select “Navigate to File or Function” from the context menu, TotalView finds and displays the source for the function, if the source is available. If there is more than one source location, displays the function name as a search in the [Lookup view](#).

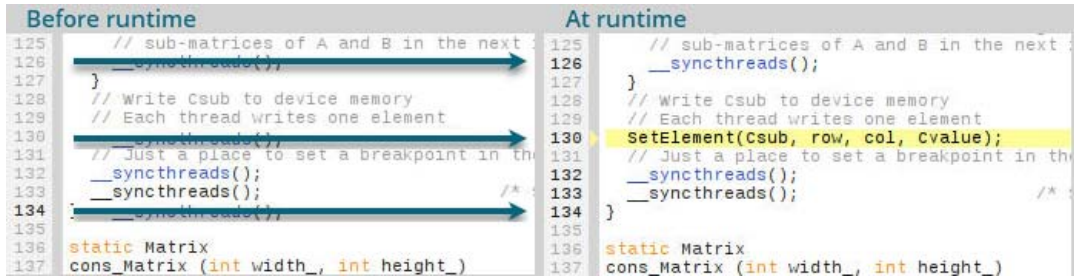
Unified Source View Display

The Source view provides a unified view of source-line breakpoints across all image files containing the source file, useful for programs in which the same source file or header file is compiled into multiple image files (e.g., executable and shared library files) used by the process.

Line numbers appear bold where TotalView has identified executable code, i.e., source code lines where the compiler has generated one or more line number symbols in the debug information.

For example, consider debugging a program that launches CUDA code running on a Graphics Processing Unit (GPU). When the host program is first loaded into TotalView, the CUDA threads have not yet launched, so the debugger has no symbol table information yet. [Figure 4](#) shows the Source view before and after a CUDA kernel launch. Before the CUDA threads exist (the left pane), only line 134 has been identified as having executable code.

Figure 4, Unified Source view display



Once the program is running and the CUDA threads have started (the right pane), lines 126, 130, 132, 133, and 134 are bold, so now TotalView has been able to identify line number symbols at those locations.

RELATED TOPICS

Using the Source view to set action points (breakpoints)

[Breakpoints](#)

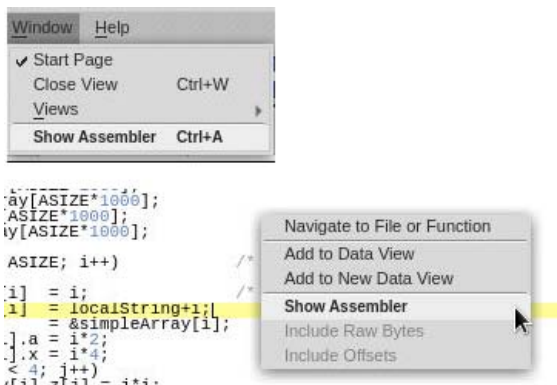
Source views and their relationship to data display

[The Processes & Threads View in Relation to Other Views](#)

The Assembler View

You can view not only your source code but the assembly code generated by the compiler. The assembly version of your code reveals the machine instructions behind an operation so you can clearly see what your code is doing.

To view assembly code, choose either the menu item **Window > Show Assembler (Ctrl+A)**, or right-click anywhere in the Source view and select **Show Assembler**.



The Source View divides into two parts.

Figure 5, Default Assembler View

```

60 }
61
62 for (i = 0; i < ASIZE; i++)
63     for (j = 0; j < ASIZE; j++)
64     {
65         twoDArray[i][j] = simpleArray[i];
66     }
67
68     for (i = 0; i < ASIZE; i++)
69         for (j = 0; j < ASIZE + 10; j++)
70             for (k = 0; k < ASIZE + 20; k++)
71                 threeDArray[i][j][k] = i*100+j*10+k;
72
73     for (i = 0; i < ASIZE; i++)
74         for (j = 0; j < ASIZE; j++)
75             charArray[i][j] = 'a' + i - j;
76
77     /* Add some odd ones to test sorting */
78     for (i = 0; i < ASIZE; i++)
79     {
80         charArray[ASIZE/2][i] = 0;
81         charArray[ASIZE/2+1][i] = 200 + i;
82     }

```

```

51 0x00401217: cmpl    $19, -4(%rbp)
52 0x0040121b: jle     main+0x2c
53 0x00401221: movl    $0, -4(%rbp)
54 0x00401228: jmp     main+0x164
55 0x0040122a: movl    $0, -8(%rbp)
56 0x00401231: jmp     main+0x15a
57 0x00401233: mov     -4(%rbp), %eax
58 0x00401236: cwtq
59 0x00401238: mov     -752(%rbp, %rax, 4), %ecx
60 0x0040123f: mov     -8(%rbp), %eax
61 0x00401242: movsxd %rax, %rsi
62 0x00401245: mov     -4(%rbp), %eax
63 0x00401248: movsxd %rax, %rdx
64 0x0040124b: mov     %rdx, %rax
65 0x0040124e: shll   $2, %rax
66 0x00401252: addl   %rdx, %rax
67 0x00401255: shll   $2, %rax
68 0x00401259: addl   %rsi, %rax
69 0x0040125c: mov     %ecx, -2352(%rbp, %rax, 4)
70 0x00401263: addl   $1, -8(%rbp)
71 0x00401267: cmpl   $19, -8(%rbp)
72 0x0040126b: jle     main+0x126
73 0x0040126d: addl   $1, -4(%rbp)
74 0x00401271: cmpl   $19, -4(%rbp)

```

The left pane contains the program's source code and the right, the assembler version. The Assembler view provides three viewing options:

- **The default view**, to view only the absolute address location and machine instructions, as shown in Figure 5.
- **The raw bytes**, to add a column with the actual values of the instruction parcels.
- **The offset**, to add a column with the hexadecimal offset from the start of the function.

The default Assembler view

The default Assembler view displays the absolute address and the machine instructions. In Figure 5, note line 62 where a breakpoint has been set; one line of source code equates to multiple machine instructions in the Assembler view. If a function contains a nested function, the contents of the nested function are also included; in this example, lines 63 and 65 represent a nested function within line 62.

Including raw bytes or offsets

To view bytes or offsets, use the context menu in the Assembler view by right-clicking in the view and selecting either **Include Raw Bytes** or **Include Offsets**. You can also choose to view both raw bytes and offsets.

```

62 0x0040121b: jle     main+0x2c
62 0x00401221: movl    $0, -4(%rbp)
62 0x00401228: jmp     main+0x164
63 0x0040122a: movl    $0, -8(%rbp)
63 0x00401231: jmp     main+0x15a
65 0x00401233: mov     -4(%rbp), %eax
65 0x00401236: cwtq
65 0x00401238: mov     -752(%rbp, %rax, 4), %ecx
65 0x0040123f: mov     -8(%rbp), %eax
65 0x00401242: movsxd %rax, %rsi
65 0x00401245: mov     -4(%rbp), %eax
65 0x00401248: movsxd %rax, %rdx
65 0x0040124b: mov     %rdx, %rax
65 0x0040124e: shll   $2, %rax
65 0x00401252: addl   %rdx, %rax
65 0x00401255: shll   $2, %rax
65 0x00401259: addl   %rsi, %rax
65 0x0040125c: mov     %ecx, -2352(%rbp, %rax, 4)
63 0x00401263: addl   $1, -8(%rbp)
63 0x00401267: cmpl   $19, -8(%rbp)
63 0x0040126b: jle     main+0x126
62 0x0040126d: addl   $1, -4(%rbp)
62 0x00401271: cmpl   $19, -4(%rbp)

```

■ Including Raw Bytes

Include the raw bytes to display the actual value of the instruction parcels. Some architectures have variable length instructions, so there is one byte per parcel.

63	0x0040068c:	c7 45 f8 00 00 00 00	ju	main+0x109
63	0x00400693:	83 7d f8 13	movl	\$0, -8(%rbp)
	0x00400697:	7f 36	cmpl	\$19, -8(%rbp)
65	0x00400699:	8b 45 fc	vg	main+0x163
	0x0040069c:	48 98	mov	-4(%rbp), %eax
	0x0040069e:	8b 94 85 10 fd ff ff	cwtd	
65	0x004006a5:	8b 45 f8	mov	-752(%rbp, %rax,
	0x004006a8:	48 63 f0	mov	-8(%rbp), %eax
	0x004006ab:	8b 45 fc	movsxd	%rax, %rsi
	0x004006ae:	48 63 c8	mov	-4(%rbp), %eax
	0x004006b1:		movsxd	%rax, %rcx

■ Including Offsets

Include offsets in the view to see the offset from the start of a function as a hexadecimal value. The function symbolic name displays at the top of the Assembler view, like so:

0x00400600 <main>

	0x0040056c:	<main>		
33	0x0040056c:	<+0x0>:	pushl	%rbp
	0x0040056d:	<+0x1>:	mov	%rsp, %rbp
	0x00400570:	<+0x4>:	subl	\$0xb4670, %rsp
	0x00400577:	<+0xb>:	mov	%edi, 0xffff4b99c(%rbp)
	0x0040057d:	<+0x11>:	mov	%rsi, 0xffff4b990(%rbp)
36	0x00400584:	<+0x18>:	movl	\$0x4009a0, -24(%rbp)
51	0x0040058c:	<+0x20>:	movl	\$0, -4(%rbp)
51	0x00400593:	<+0x27>:	cmpl	\$19, -4(%rbp)
	0x00400597:	<+0x2b>:	vg	main+0x113
53	0x0040059d:	<+0x31>:	mov	-4(%rbp), %eax
	0x004005a0:	<+0x34>:	cwtd	
	0x004005a2:	<+0x36>:	mov	-4(%rbp), %edx

When including offsets, each line contains an extra line showing its offset from `main`.

Adding the offset helps identify the target of an instruction. For example, looking at the selected “jump” instruction `jle` at line 58 makes it easier to find the offset it references at line 59.

	0x00401101:	<+0xc4>:	movss	%xmm0, (%rax)
58	0x004011d5:	<+0xc8>:	movl	\$0, -8(%rbp)
58	0x004011dc:	<+0xcf>:	jmp	main+0x100
59	0x004011de:	<+0xd1>:	mov	-8(%rbp), %eax
	0x004011e1:	<+0xd4>:	imull	-4(%rbp), %eax
	0x004011e5:	<+0xd8>:	mov	%eax, %ecx
59	0x004011e7:	<+0xda>:	mov	-8(%rbp), %eax
	0x004011ea:	<+0xdd>:	movsxd	%rax, %rsi
	0x004011ed:	<+0xe0>:	mov	-4(%rbp), %eax
	0x004011f0:	<+0xe3>:	movsxd	%rax, %rdx
	0x004011f3:	<+0xe6>:	mov	%rdx, %rax
	0x004011f6:	<+0xe9>:	addl	%rax, %rax
	0x004011f9:	<+0xec>:	addl	%rdx, %rax
	0x004011fc:	<+0xef>:	addl	%rax, %rax
	0x004011ff:	<+0xf2>:	addl	%rsi, %rax
	0x00401202:	<+0xf5>:	mov	%ecx, -504(%rbp, %rax, 4)
	0x00401209:	<+0xfc>:	addl	\$1, -8(%rbp)
58	0x0040120d:	<+0x100>:	cmpl	\$0, -8(%rbp)
	0x00401211:	<+0x104>:	jle	main+0xd1
51	0x00401213:	<+0x106>:	addl	\$1, -4(%rbp)
51	0x00401217:	<+0x10a>:	cmpl	\$19, -4(%rbp)

Setting breakpoints in the Assembler view

You can set breakpoints in either the Source view or the Assembler view.

RELATED TOPICS

Setting breakpoints in the Assembler view	Setting Machine-Level Action Points
The Source View	The Source View

Toolbars

The UI has the following toolbars:

Figure 6, TotalView Toolbars



NOTE: The ReplayEngine toolbar appears only on the supported platform Linux x86-64.

In the Settings (middle item), you can control which toolbars are displayed, and request that the toolbar items include descriptive text:



Another set of toolbars to support CUDA debugging is available when a CUDA program is being run. While you can display these at any time, they are responsive only when TotalView is debugging a CUDA program. See [CUDA Debugging Tutorial](#).

RELATED TOPICS

About the debugging commands

[Debugging Commands](#)

About the ReplayEngine commands

[Replaying Your Program in the *ReplayEngine User Guide*](#)

Controlling the scope of debugging commands

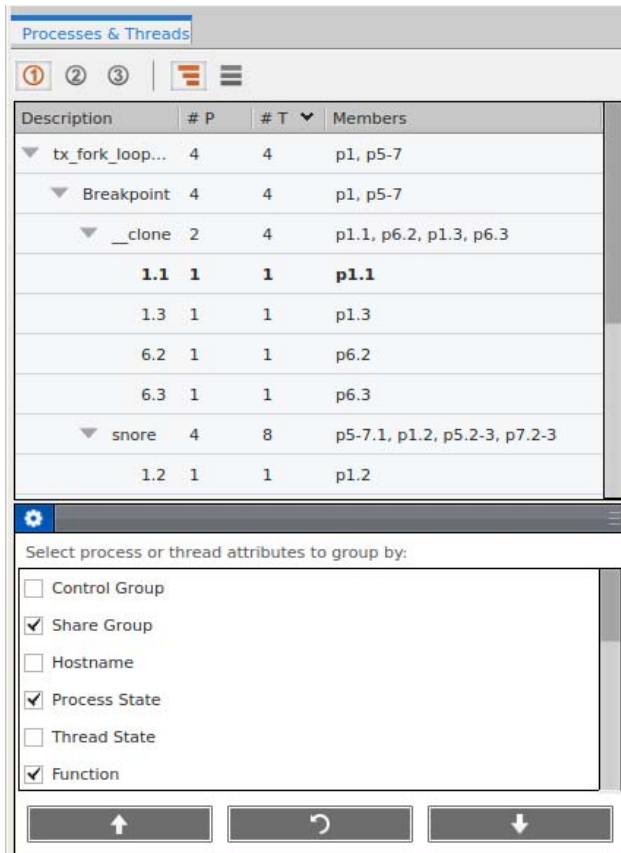
[Stepping and Program Execution](#)

Processes and Threads View

Once a program is running, the Processes and Threads view displays information about all of the processes and threads running in your program. The attributes list at the bottom lets you choose which information to display, and in what order. By manipulating these attributes you can create various views into your program.

One line is bold, indicating the process and thread that currently has the focus. You can double-click on a different line to change the focus to that process and thread. The thread of focus determines the data displayed in the Call Stack and Data View.

Figure 7, Processes and Threads View



RELATED TOPICS

Detailed information on this view

[The Processes & Threads View](#)

The thread/process of focus and its effect on the display of data

[The Processes & Threads View in Relation to Other Views](#)

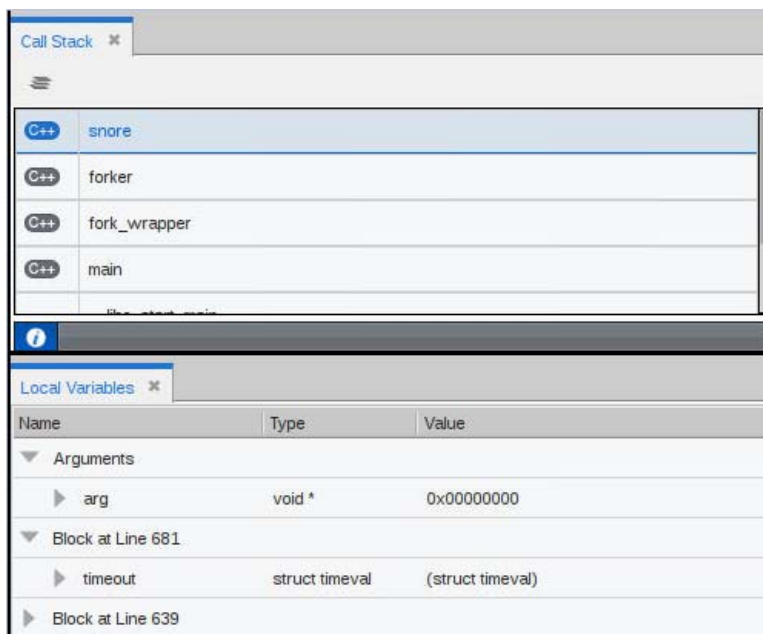
Call Stack View and Local Variables View

The display in the Call Stack view depends on which thread has the focus. That thread is highlighted in bold in the Processes and Threads view. You can double-click on a different line in the Processes and Threads view to change the focus to another thread.

The Call Stack view shows the stack trace for the thread in focus, allowing you to trace back through the execution of the thread. If the left column shows a language, source code is available and clicking on that stack entry displays the source code in a Source view at the location of the named function. If no language is shown, clicking on the stack entry still displays a Source view, but it simply says “No Source Available”.

The Local Variables view displays variables associated with the selected frame.

Figure 8, Call Stack View with Local Variables View



RELATED TOPICS

Detailed information on this view

[The Call Stack, Local Variables, and Registers Views](#)

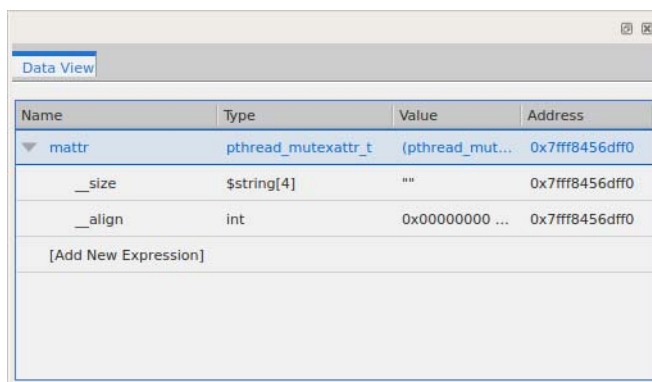
Data View

The Data View allows you to keep track of specific variables as you move around your program, and to manipulate the data in those variables in a number of ways. You add variables to the Data View by selecting them in the Local Variables view and either dragging them into the Data View, or right-clicking and selecting Add to Data View from the context menu.

Once data is in the Data View, you can do a number of things:

- Dereference pointers to access the data they point to
- Recast data to see different views of it, such as recasting pointers to the first value in an array into the actual array so you can see the contained values
- Changing data values to see the effect on the program, which you can also do in the Local Variables view.

Figure 9, The Data View



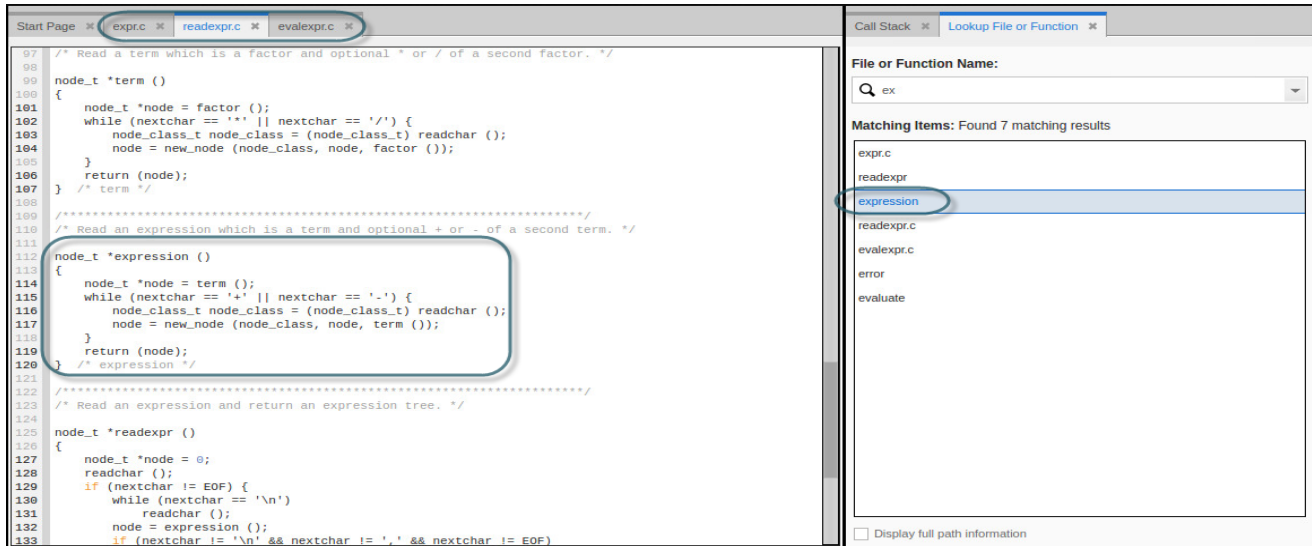
RELATED TOPICS

Examining, manipulating, and editing data [The Data View](#)

Lookup View

If your program is large and complex, finding functions or files can be challenging. The Lookup view allows you to search using any substring. Suppose you suspect a problem with the `expression` function. In [Figure 10](#), the string “`ex`” returns a number of file and function names, including the `expression` function. Clicking on the function name displays it in a Source view, with the desired function centered in the view.

Figure 10, Lookup View



RELATED TOPICS

Detailed information on this view

[The Lookup File or Function View](#)

Action Points, CLI, and Logger Views

The lower display area features a number of views.

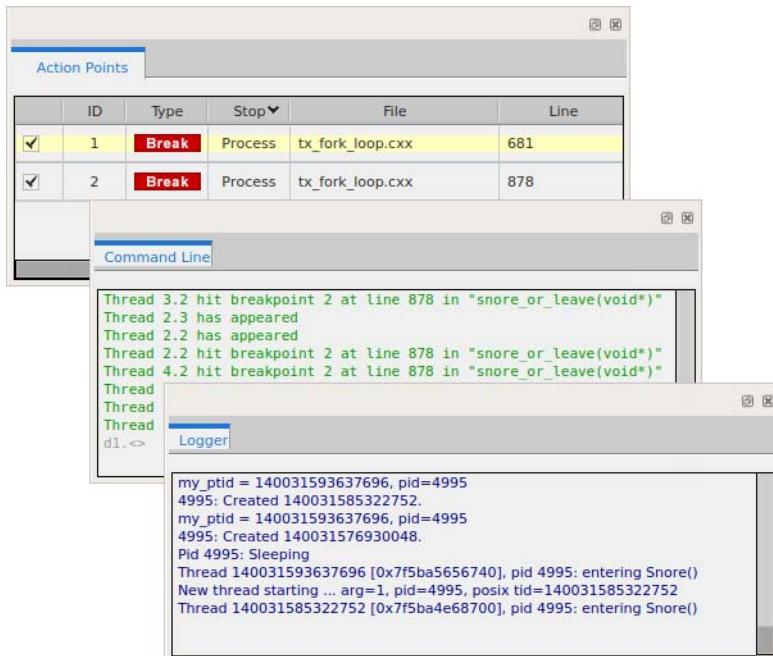
Action Points View. This view lists all of the action points — breakpoints, evalpoints, and barrierpoints — in your debugging session. You can add, delete, enable, and disable actions points in this view.

Command Line View. Although not yet fully supported in the UI, the full power of the Classic TotalView debug engine is available through the Command Line Interface (CLI). You can enter those commands in this view.

Logger. This view makes it easy to see the log messages that TotalView issues.

Note that the Command Line and Logger views allow text selection, cutting, and pasting of their contents.

Figure 11, Action Point, Command Line, and Logger Views



RELATED TOPICS

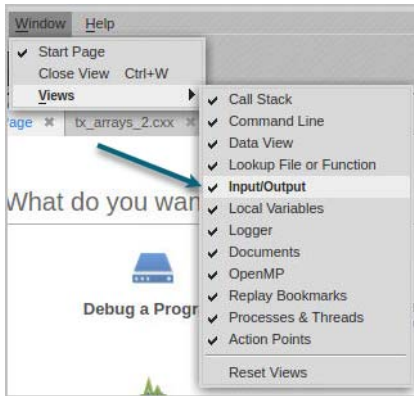
Detailed information about the Action Points view [Setting and Managing Action Points \(Breakpoints\)](#)

More information about the Command Line view [Access to the CLI](#)

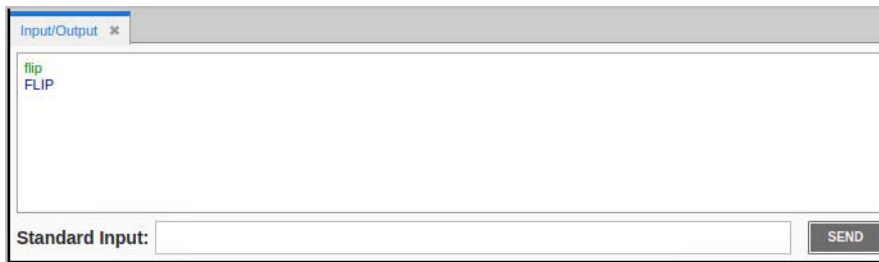
Input/Output View

The Input/Output view accepts user input and displays program output. This view is closed by default, but is available if your program requires user input or you want to enter or view program output in the UI, rather than in the terminal. You can also switch between the UI and the terminal; all input and output is reflected both in the UI *and* the terminal.

To open it, select **Window > Views > Input/Output**.



Input displays in green text, while output displays in blue text. Errors display in red.

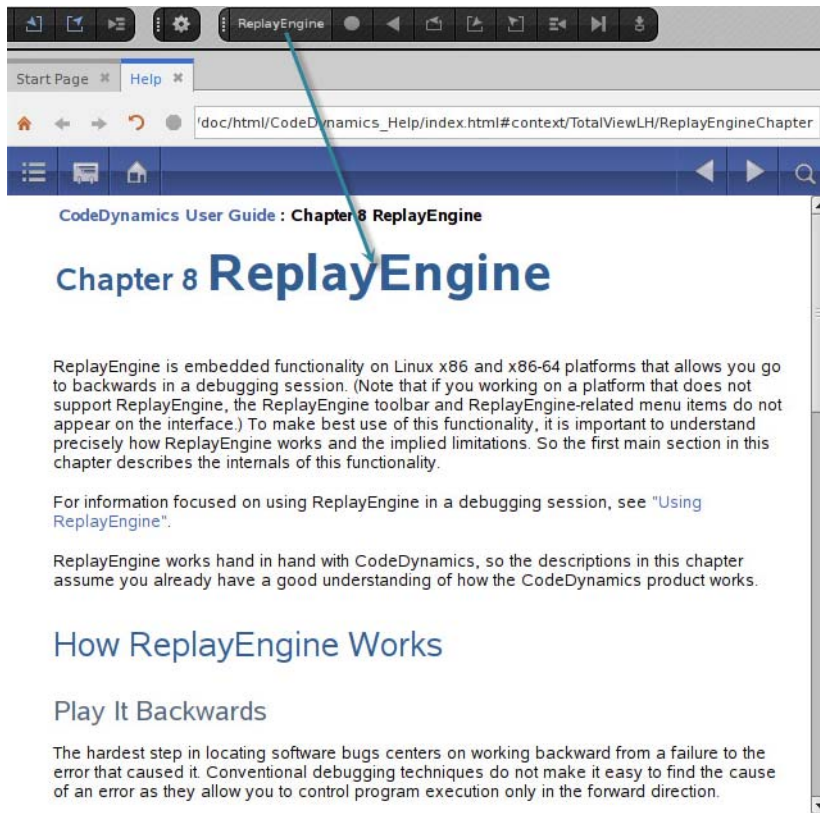


Help

One view that shows up in the main display area is the Help window. This can of course be displayed by selecting various items on the Help menu, such as Contents.

Context-sensitive information about parts of the interface can be obtained by placing the cursor over the area you are interested in and pressing **F1**. Information about that area appears in the Help window, or sometimes help about a parent container shows up, which usually contains the information you are seeking.

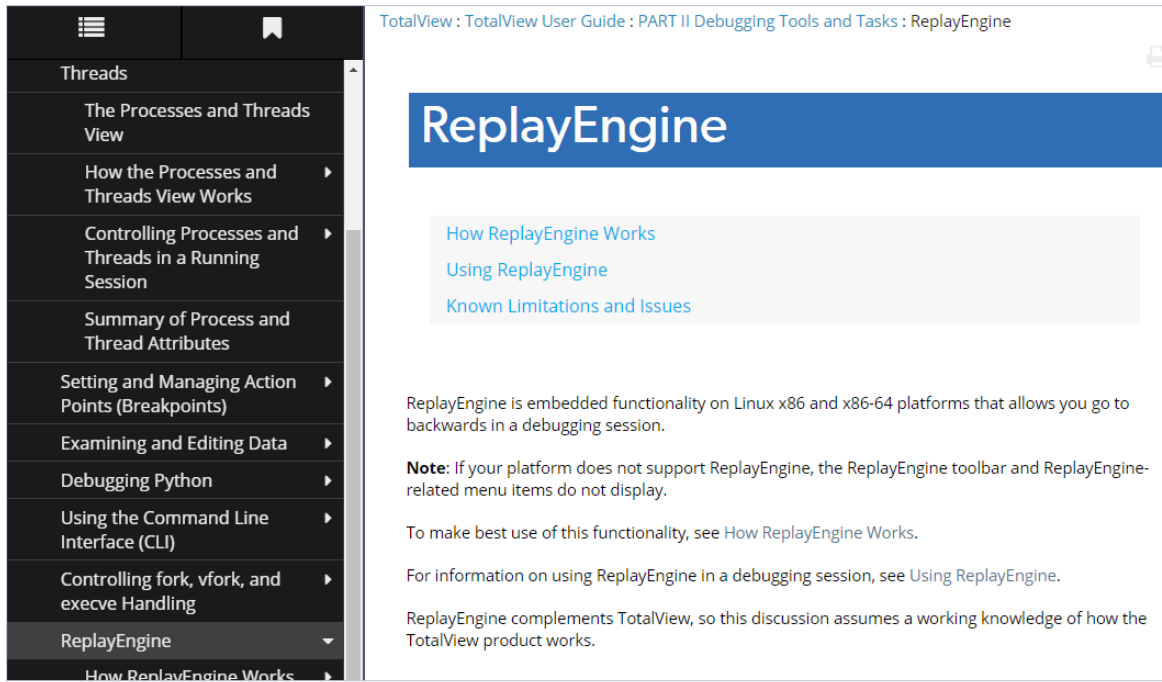
Figure 12, Obtaining Context-Sensitive Help



In Figure 12, F1 was pressed with the cursor over the ReplayEngine toolbar.

The information displayed for context sensitive help is from the full product documentation for TotalView. If you move the Help into a separate window and increase its size, at some point navigation for the full product documentation appears.

Figure 13, The Help Window with Full Product Documentation



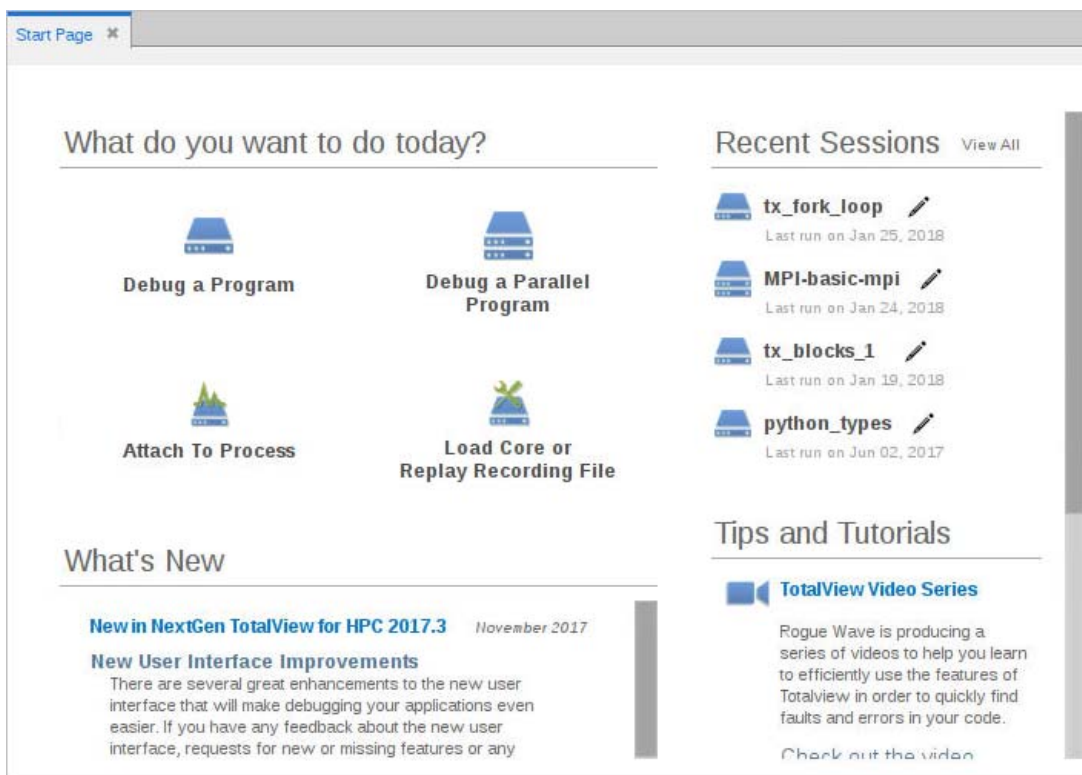
Starting TotalView and Creating a Debugging Session

Start TotalView in two primary ways: with no arguments to launch the Start Page, or with arguments to skip the Start Page and open the UI with the program loaded and ready to debug.

The Start Page

If you start TotalView without arguments, the UI displays the Start Page, from which you can access the Sessions Editor. This is the easiest way to load a program into TotalView. Once you configure a debugging session using the Sessions Editor, the settings are saved under Recent Sessions so you can access them later.

Figure 14, Starting TotalView at the Start Page



From this page you can:

- Specify a program to debug — **Debug a Program**
- Specify a parallel program to debug — **Debug a Parallel Program**

- Start debugging a running program — **Attach to Process**
- Specify a core file to debug, or a ReplayEngine recording file to load — **Load Core or Replay Recording File**
- Restart a previously defined session — **tx_fork_loop**

Loading a program directly into TotalView

Load a program into TotalView by entering:

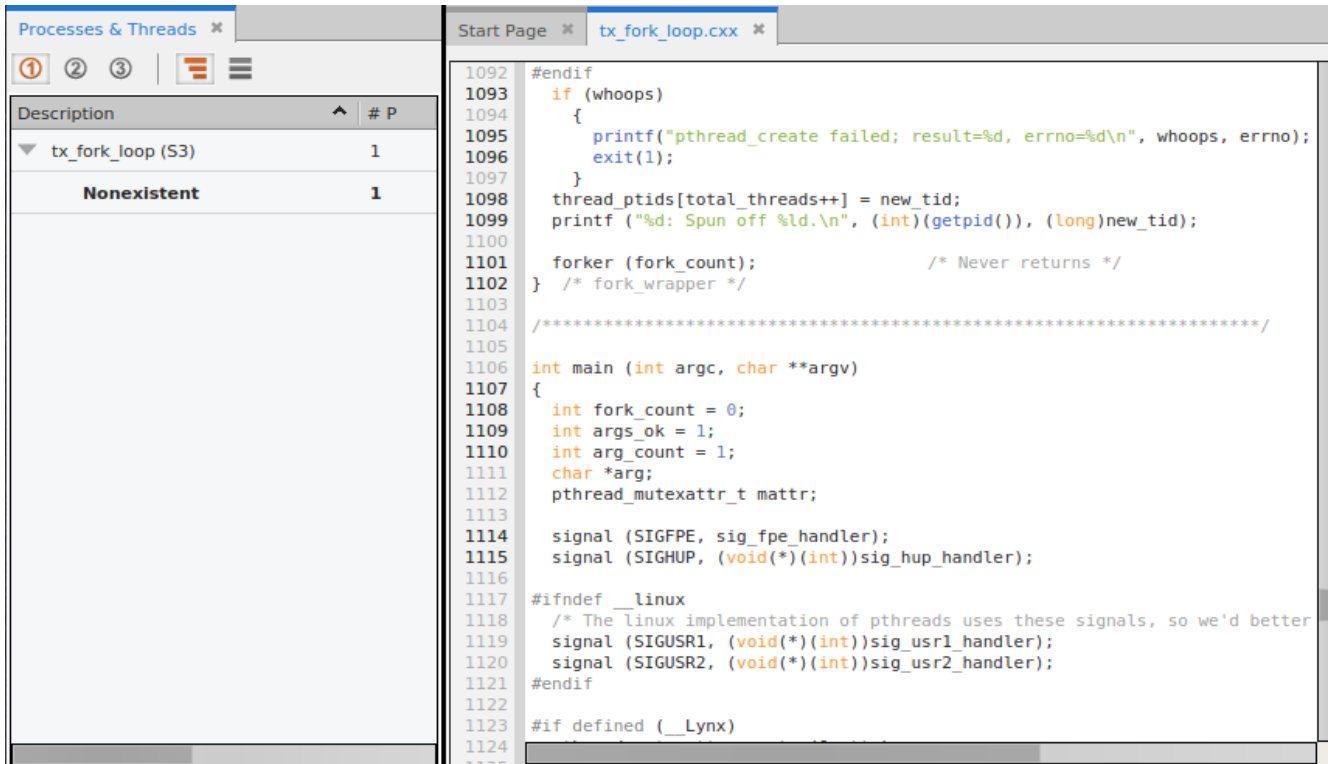
```
totalview executable_name [ argument argument ... ]
```

where *executable_name* is the executable of the program you want to debug, followed by any arguments the program takes. This opens the TotalView UI with the program loaded and ready to debug.

[Figure 15](#) shows the Source view for the program and the Processes and Threads view. These are just two of several available views.

NOTE: To run a program in TotalView, compile it for debugging, usually with the **-g** command-line option, depending on your compiler.

Figure 15, Starting TotalView with an Executable Name



RELATED TOPICS

[Defining, Editing, and Managing Sessions](#)

[Creating and Managing Sessions](#)

[More on compiling programs for debugging](#)

*** 'Compiling for Debugging' on page 584 ***

[More on starting TotalView](#)

[Starting a Session from your Shell](#)

Debugging Commands

The table below summarizes the behavior of the debugging commands available in TotalView. The descriptions assume that the command is being applied to a single thread, the thread with the focus. In fact, debugging commands are much more flexible than this. They can apply to threads, processes, or groups, or some collection of these. You select the different scopes for debugging commands from the menu on the left of the toolbar, or by selecting the commands from the Thread, Process, and Group menus.

See Related Topics below for the location of discussions about these extended capabilities.



Command	Description
Go	Sets the thread to running until it reaches a stopping point. Often this will be a breakpoint that you have set, but the thread could stop for other reasons.
Halt	Stops the thread at its current execution point.
Kill	Stops program execution. Existing breakpoints and other settings remain in effect.
Restart	Stops program execution and restarts the program from the beginning. Existing breakpoints and other settings remain in effect. This is the same as clicking Kill followed by Go.
Next	Moves the thread to the next line of execution. If the line the thread was on includes one or more function calls, TotalView does not step into these functions but just executes them and returns.
Step	Like Next, except that TotalView does step into any function calls, so the thread stops at the first line of execution of the first function call.
Out	If the thread is in a block of execution, runs the thread to the first line of execution beyond that block.
Run To	If there is a code line selected in one of the Source views, the thread will stop at this line, assuming of course that it ever makes it there. This operates like a one-time, temporary breakpoint.

RELATED TOPICS

Controlling the scope (width) of debugging commands	Stepping and Program Execution
Controlling what happens when a thread reaches a breakpoint (action point)	Controlling an Action Point's Width
Seeing the debugging commands in action	Stepping and Executing

Diving on Program Elements

Diving is integral to the TotalView UI and provides a quick, intuitive, and effective way to get more information about various program elements. Dive on an element either by just double-clicking on it or via a context menu, depending on the element. For example:

- Dive on a thread or function in the Processes & Threads view, (by double clicking on it), and the Source view switches its focus to that element.
- Navigate to a function in the Source pane to move its focus to that element.
- Dive on an expression or variable in the Data View to add it as a new expression in the Data View. This is helpful for examining one segment of a data structure or element of an array of data. See [Diving on Variables](#).

Creating and Managing Sessions

There are two primary ways to load programs into TotalView for debugging: the UI via the Start Page ([Loading Programs from the Session Editor](#)) or with CLI commands ([Loading Programs Using the CLI](#)). Both support all debugging session types.

There are also ways to start TotalView with arguments that set up a session when the program opens ([Starting a Session from your Shell](#)).

Setting up Debugging Sessions

- [Loading Programs from the Session Editor](#)
 - [Starting a Debugging Session](#)
 - [Debug a Program](#)
 - [Debug a Parallel Program](#)
 - [Attach to Process](#)
 - [Debug a Core or Replay Recording File](#)
 - [Load a Recent Session](#)
 - [Editing a Previous Session](#)
- [Loading Programs Using the CLI](#)

Additional Session Setup Options

- [Program Environment](#)
- [Standard Input and Output](#)

Managing Debug Sessions

- [Managing Sessions](#)

Starting TotalView with a Session Initiated

- Starting a Session from your Shell

Setting up Debugging Sessions

The easiest way to set up a new debugging session is to use the **Session Editor**, an easy-to-use interface for configuring sessions and loading programs into TotalView. Alternatively, you can use the CLI.

- Loading Programs from the Session Editor
- Loading Programs Using the CLI

Loading Programs from the Session Editor

TotalView can debug programs on local and remote hosts, and programs accessed over networks and serial lines. Use the **Session Editor** to configure a new debugging session or to access a previous session. Access the Session Editor via either the **Start Page** or the **File menu**. The Start Page provides access to all types of debug sessions ([Starting a Debugging Session](#)), while the File menu enables you to choose a specific debugging session, such as loading local and remote programs, core files, and processes that are already running.

Figure 16, Choosing a Specific Debug Session from the File Menu

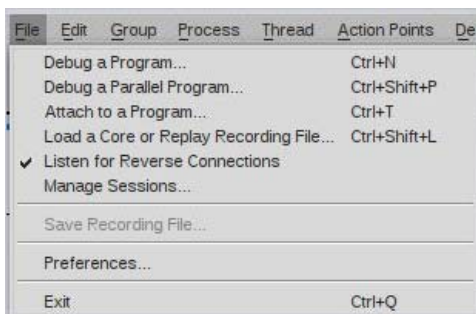
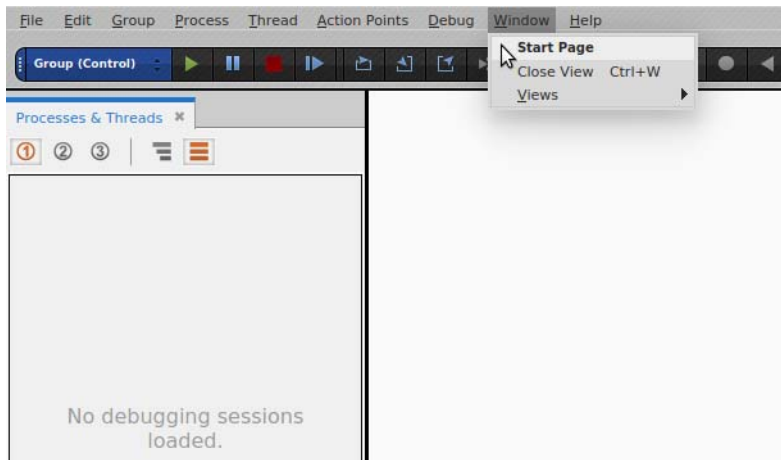


Figure 17, Opening the Sessions Editor from the Window Menu



If you are just starting TotalView, the Start Page automatically opens.

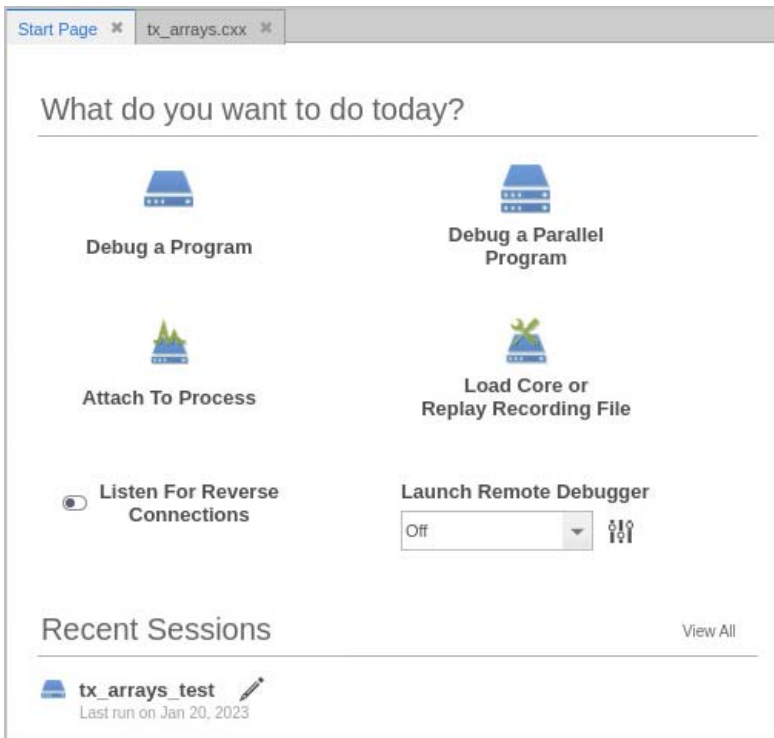
Starting a Debugging Session

Access the Start Page either directly from your shell by just entering

```
totalview
```

or by selecting **Window > Start Page** from within the UI if TotalView is already running.

Figure 18, Start Page Opening View



From this initial window, you can configure various types of debugging sessions:

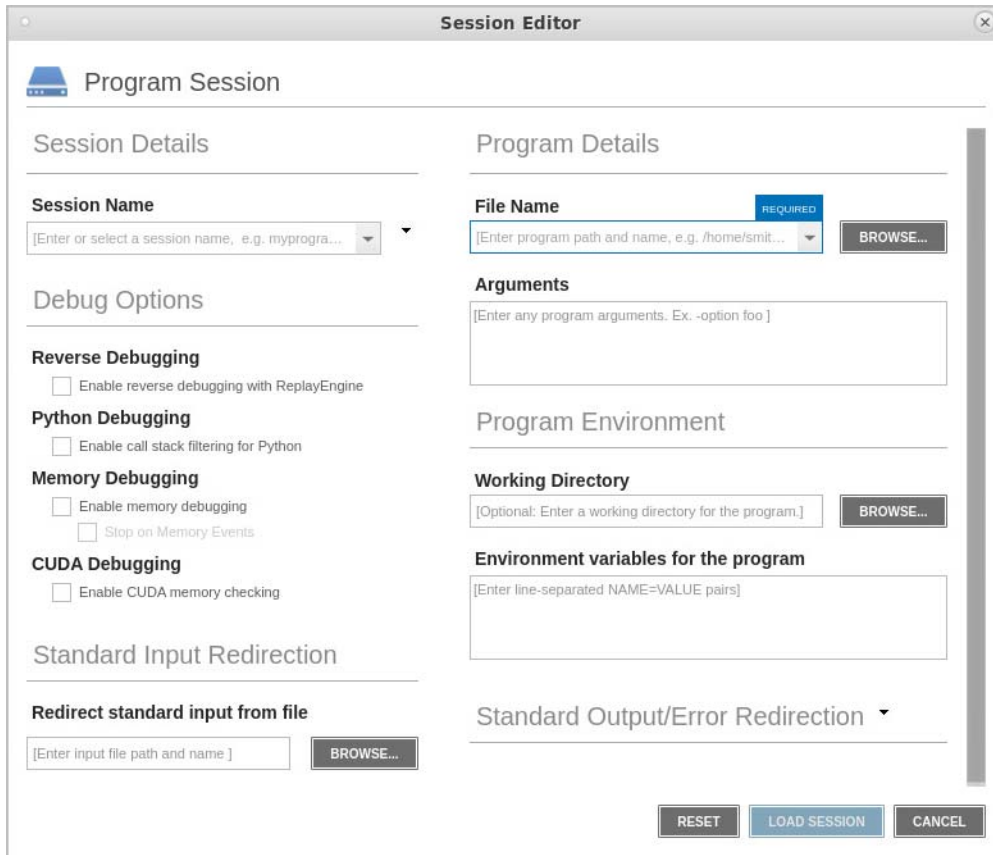
- [Debug a Program](#)
- [Attach to Process](#)
- [Debug a Parallel Program](#)
- [Debug a Core or Replay Recording File](#)
- [Load a Recent Session](#)

You can also control whether TotalView listens for reverse connections. ([Reverse Connections](#)) or launches a remote debugger ([About Remote Connections](#)).

Debug a Program

To configure a new debugging session, select **Debug a Program** to launch the Program Session dialog.

Figure 19, Program Session dialog



1. Enter a session name in the **Session Name** text box.
Note that any previously entered sessions of the same type are available from the Session Name dropdown box. See [Editing a Previous Session](#).
2. Enter the name of your program in the **File Name** box or press **Browse** to browse to and select the file. You can enter a full or relative path name. If you have previously entered programs here, they will appear in a dropdown list.
If you enter a file name and the UI cannot find it, it displays the path in red; however, TotalView searches for it in the list of directories listed in your **PATH** environment variable.

CLI: dset EXECUTABLE_PATH

3. (Optional) Add any custom configurations or options:

- **Program arguments:** Enter any program arguments into the Arguments field.
Because you are loading the program from within TotalView, you need to enter the command-line arguments that the program requires.
- **Debug Options:**
For detail, see [Debug Options](#).
- **Program Environment:**
[Working Directory](#).
[Environment Variables for the Program](#).
- **Standard Input and Output:** See [Standard Input and Output](#).

4. Click **Load Session**. The Load Session button is enabled once all required information is entered.

Debug a Parallel Program

TotalView supports the popular parallel execution models including MPI, OpenMP, SGI shared memory (shmem), Global Arrays, UPC, CAF, fork/exec, and pthreads.

Starting an MPI Program

MPI programs use a starter program such as **mpirun** to start your program. You can start these MPI programs in two ways:

- With the starter program under TotalView control. In this case, enter the name of the starter program on the command line.
- Using the UI, in which case the starter program is not under TotalView control. In this scenario, enter program information into the [Parallel Session Dialog](#) from within the Session Editor.

Programs started using the UI have some limitations: program launch does not use the information you set for single-process and bulk server launching, and you cannot use the Attach Subset command.

Starting MPI programs using the Session Editor is described here. For examples using a starter program, see [Starting a Session from your Shell](#).

Parallel Session Dialog

From the Start Page, select **Debug a Parallel Program** to launch the Parallel Session dialog.

1. Session and Program Details

Figure 20, Parallel Session: Session and Program Details

The screenshot shows a window titled "Session Editor" with a close button in the top right corner. Inside the window, there is a "Parallel Session" header with a small icon. Below this, the dialog is divided into three sections: "Session Details", "Program Details", and "Arguments".

- Session Details:** Contains a "Session Name" field with a dropdown arrow. The placeholder text reads: "[Enter or select a session name, e.g. myprogram with ReplayEngine]".
- Program Details:** Contains a "File Name" field with a dropdown arrow and a "Browse..." button. The placeholder text reads: "[Enter program path and name, e.g. /home/smith/myprogram]". A blue "REQUIRED" label is positioned above the field.
- Arguments:** Contains a text input field with the placeholder text: "[Enter any program arguments. Ex. -option foo]".

At the bottom of the dialog, there are three buttons: "RESET", "LOAD SESSION", and "CANCEL".

Session Details: Enter a session name in the **Session Name** field.

NOTE: Any previously entered sessions of the same type are available from the Session Name dropdown box. Once selected, you can change any session properties and start your debug session. See [Editing a Previous Session](#)

Program Details

- **File Name:** Enter the name of your program or press Browse to browse to and select the file. You can enter a full or relative path name. If you have previously entered programs here, they will appear in a dropdown list.

If you enter a file name and the UI cannot find it, it displays the path in red; however, TotalView searches for it in the list of directories listed in your **PATH** environment variable. See [Search Path](#).

CLI: dset EXECUTABLE_PATH

- **Arguments:** Enter any arguments to be sent to your program.

Because you are loading the program from within the UI, you need to enter the command-line arguments that the program requires.

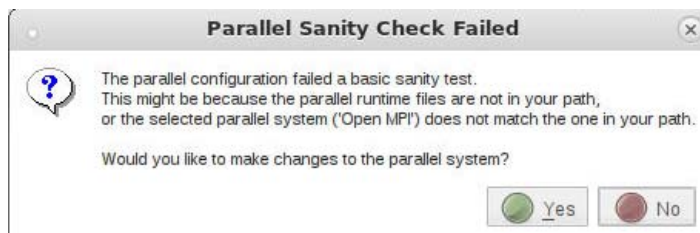
2. Parallel Details

Figure 21, Parallel Session: Parallel Details

- **Parallel System:** Select which parallel system profile TotalView should use when starting your program. This profile can be one that TotalView provides, one created for your site, or one that you create. (For information, see [MPI Startup Customizations](#).)
- **Tasks:** Enter a number indicating how many tasks your program should create.
If your system has a default value and you want to use it, enter a value of 0 (zero).
If your system has no default value or you want to override the default, enter a value of 1 or greater.
- **Nodes:** (System-dependent) Enter a number indicating how many nodes your program should use when running your program. (Not all systems use this value, so this field may not be visible.)
- **Additional Starter Arguments:** If your program's execution requires that you use arguments to send information to the starter process such as **mpirun** or **poe**, enter them in this field. (In contrast, if you need to use arguments to send information to your program, enter those arguments in the Arguments field under Program Details.)

3. **Debug Options:** See [Debug Options](#).
4. **Program Environment:** See [Program Environment](#).
5. **Standard Output/Error Redirection:** See [Standard Input and Output](#).
6. **Standard Input Redirection:** See [Standard Input and Output](#).
7. Select the **Load Session** button to launch the debugger.

NOTE: Note that any errors in the parallel configuration will launch an error pop-up:



If you continue with the session, additional errors launch, and your session may not run correctly.

Once created, a session named `my_foo` can be quickly launched later using the `-load` command line option, like so:

```
totalview -newui -load_session my_foo
```

RELATED TOPICS

Set up MPI debugging sessions for various environments and special use cases

[MPI Program Setup](#)

Set up non-MPI parallel debugging sessions for applications that use the parallel execution models that TotalView supports

[Non-MPI Program Setup](#)

Create MPI startup profiles for environments that TotalView doesn't define

[MPI Startup Customizations](#)

Tips for debugging parallel applications

Debugging Strategies for Parallel Applications" in the *Classic TotalView User Guide*

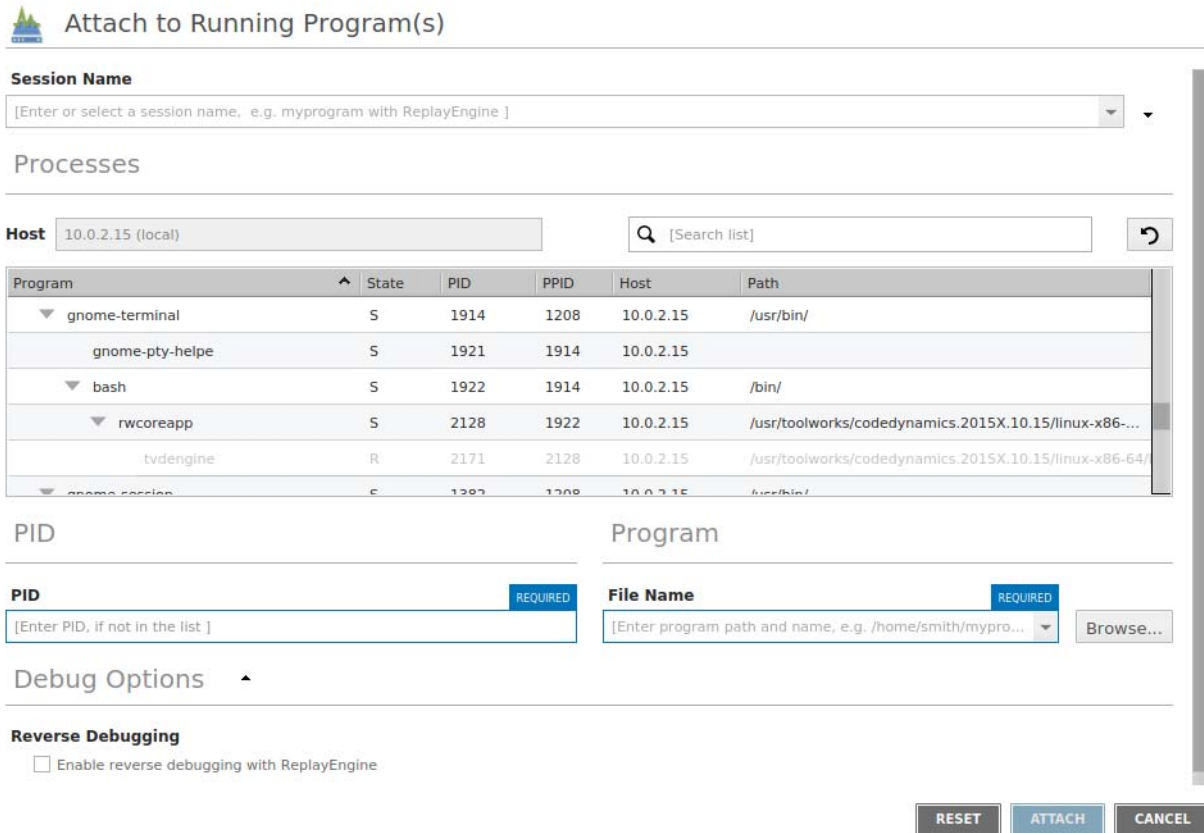
Attach to Process

If a program you're testing is hung or looping (or misbehaving in some other way), you can attach to it while it is running.

To open the Attach window, select **Attach to Process** on the Start Page.

A list of processes running on the local host displays in the **Attach to running program(s)** dialog.

Figure 22, Attach to a running program



In the displayed list, processes to which TotalView can attach are displayed in black text, while those to which TotalView has already attached or are not attachable for any reason are grayed out.

1. Enter a name for this session in the **Session Name** field.
 Any previously entered sessions of the same type are available from the Session Name dropdown box. Once selected, you can change any session properties and start your debug session. See [Editing a Previous Session](#).
2. Select the process under the Program column. For a single selected process, the **PID** and **File Name** fields are auto-populated. Alternatively, use these fields to specifically identify a process to attach to.
 To select multiple processes, hold down the **Ctrl** key and select them. (In this case, the **PID** and **File Name** fields are not used.)

3. Press **Attach**.

CLI: dattach executable pid

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.

RELATED TOPICS

The CLI **dattach** command

dattach in the *TotalView Reference Guide*

The CLI **ddetach** command

ddetach in the *TotalView Reference Guide*

Field Definitions

The Processes section displays these fields:

Program	State	PID	PPID	Host	Path
gnome-terminal	S	1914	1208	10.0.2.15	/usr/bin/
gnome-pty-helpe	S	1921	1914	10.0.2.15	
bash	S	1922	1914	10.0.2.15	/bin/
rwcoreapp	S	2128	1922	10.0.2.15	/usr/toolworks/codedynamics.2015X.10.15/linux-x86-...
tvengine	R	2171	2128	10.0.2.15	/usr/toolworks/codedynamics.2015X.10.15/linux-x86-64/
gnome-session	S	1208	1208	10.0.2.15	/usr/bin/

- **Program:** The name of the executing program. Notice that TotalView indents some names. This indentation indicates the parent/child relationship within the UNIX process hierarchy.
- **State:** A letter indicating the program's state, as follows:

Character and Meaning	Definition
I (Idle)	Process has been idle or sleeping for more than 20 seconds.
R (Running)	Process is running or can run.
S (Sleeping)	Process has been idle or sleeping for less than 20 seconds.
T (Stopped)	Process is stopped.
Z (Zombie)	Process is a "zombie"; that is, it is a child process that has terminated and is waiting for its parent process to gather its status.

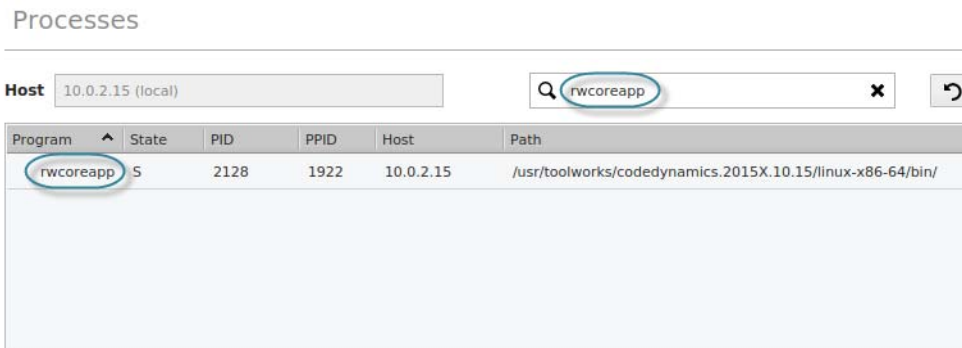
- **Host:** The name of the machine upon which the program is executing
- **PID:** The operating system program ID
- **PPID:** The parent program's ID
- **Path:** The program's path on the local machine, that is, the machine where TotalView is running.

If you attach to multiple processes, TotalView places all of them into the same control group, enabling you to stop and start them as a group.

Searching for Processes

Search for any process using the search box ().

If found, the process displays in the Processes pane.



In some cases, the name provided to TotalView by your operating system may not be the actual name of the program. In this case, you will not be able to simply select the name. Instead, you should

- Determine what its actual name is by using a command such as **ls** in a shell window.
- Select the name as this will fill in much of the program's name.
- Move to the **File Name** control, and type its actual name, then press **Enter**.

If you wish to attach to a multiprocess program, you can either select multiple processes here, or you can restart the program under TotalView control so that the processes are automatically picked up as they are forked. In most cases, this requires you to link your program with the **dbfork** library, as discussed in the section [Linking with the dbfork Library](#).

If the process you are attaching to is one member of a collection of related processes created with `fork()` calls, TotalView asks if you want to also attach to all of its relatives. If you answer yes, TotalView attaches to all the process's ancestors, descendants, and cousins.

NOTE: If some of the processes in the collection have called `exec()`, TotalView tries to determine the new executable file for the process. If TotalView appears to read the wrong file, you should start over, compile the program using the **dbfork** library, and start the program under TotalView control.

Debug Options

NOTE: Debug options are platform-specific, so your system may or may not include the options discussed in this section.

In the **Debug Options** section, you can enable ReplayEngine. (See [Reverse Debugging](#).)

Debug Options

Reverse Debugging

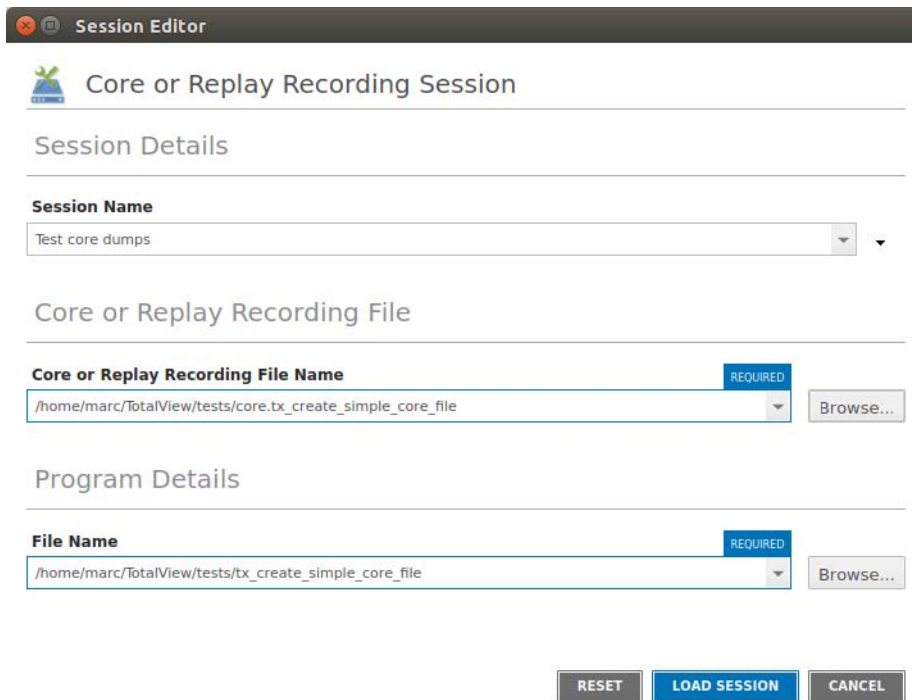
Enable reverse debugging with ReplayEngine

Debug a Core or Replay Recording File

To configure a core file or Replay Recording debug session, select **Load Core or Replay Recording File** from the Start Page. The “Core or Replay Recording Session” dialog launches.

1. Enter a name for the session in the **Session Name** field.
2. Select the core or Replay recording file to debug.
Use the **Browse** button to search the file system for the file.
3. Select the related program in the **File Name** field under the Program Details section.
4. Click **Load Session**.

Figure 23, Open a Core File



If your operating system can create multi-threaded core files (and most can), TotalView can examine the thread in which the problem occurred. It can also show you information about other threads in your program.

Similarly, TotalView can load previously saved replay recording session files to further debug applications.

When TotalView loads the core or replay recording session, it displays the core file/replay recording file, showing the state of the program. The status ribbon at the bottom of the window displays either the signal that caused the core dump, or “Recording File.” The yellow arrow and highlight in the Source Pane indicate the location of the program counter (PC) when the process encountered the error.

If you start a process while you’re examining a core file, TotalView stops using the core file and switches to this new process.

RELATED TOPICS

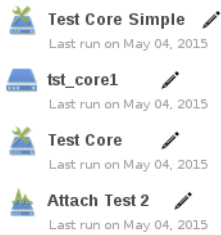
The CLI **dattach** command’s **-c *corefile-name*** | **dattach** in the *TotalView Reference Guide*
replayrecordingsessionfile option

Load a Recent Session

The Session Editor displays your most recent sessions on the Start Page so you can quickly continue a debugging session where you left off.

Figure 24, Start a Previous Debugging Session

Recent Sessions



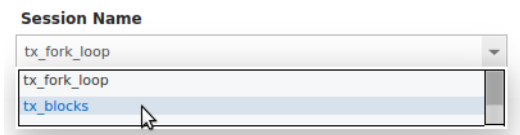
Click on a session to immediately load your previous session into TotalView.

To edit a previous session before loading it, see [Editing a Previous Session](#).

Editing a Previous Session

The **Session Name** field on each sessions window contains a dropdown that lists all previously created sessions of this type.

Figure 25, Sessions Name dropdown of a Program Session window



To edit a previous session, either select the previous session, or click the Pencil icon to open the Session Editor populated with session data. You can edit any session data, including the Session Name to create an entirely new session.

Loading Programs Using the CLI

When using the CLI, you can load programs in a number of ways. Here are a few examples.

Load a session

```
dsession -load session-name
```

Loads the session directly into TotalView.

Start a new process

dload -e *executable*

Open a core file

dattach -c *corefile* -e *executable*

If TotalView is not yet running, you can also provide the core file as a startup argument, like so:

totalview *executable corefile* [options]

Open a replay recording session file

dattach -c *replay-recording* -e *executable*

If TotalView is not yet running, you can also provide the replay recording file as a startup argument, like so:

totalview *executablereplay-recording-file* [options]

Attach to a program using its process ID

dattach *executable pid*

Load an MPI job using the POE configuration and the `hfiles` starter argument. In this example, two processes will be used across nodes.

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

RELATED TOPICS

CLI commands	"CLI Commands" in the <i>TotalView Reference Guide</i>
Loading sessions from the command line	Starting a Session from your Shell

Options and Program Arguments

The Session Editor supports setting options and program arguments either when first setting up a session or during a running session. These settings include:

- Debug Options
- Program Environment
- Standard input or output settings

See [Modifying Arguments in an Open Session](#) for how to set options during an existing session.

Debug Options

Debug program options, including for parallel programs, include:

Reverse Debugging

Reverse debugging records all program state while the program is running, then allows you to roll back your program to any point.

The reverse debugging check box is visible only on Linux-x86-64 platforms. If you do not have a license for ReplayEngine, enabling the check box has no effect, and TotalView displays an error message when your program begins executing. Selecting this check box tells TotalView that it should instrument your code so that you can move back to previously executed lines.

See the [ReplayEngine User Guide](#).

Python debugging

The Python language is easily extensible with C and C++ code. This enables Python applications to access legacy algorithms, specialized hardware, and to perform highly specialized computing. TotalView supports debugging Python extensions, shows a clean set of stack frames across the language barriers, and allows both Python and C/C++ variables to be examined and compared.

Python debugging is supported only on Linux-x86-64 platforms.

See [Starting a Python Debugging Session](#).

Memory debugging

Locate many of your program's memory problems, including leak detection, heap and event reports, and the ability to identify dangling pointers.

See [Starting Memory Debugging in TotalView](#).

CUDA debugging

Detect global memory addressing violations and misaligned global memory accesses by enabling the CUDA Memory Checker feature.

See [Enabling CUDA Memory Checker Feature](#).

Program Environment

Control the program environment by changing the working directory or setting environment variables.

Working Directory

The working directory option specifies a working directory for executing your target program. If not provided, the default is the directory from which you invoked TotalView.

This value can be entered into the UI or on the command line when starting TotalView. It can then be modified during a debug session using the **Process > Modify Arguments** menu.

Set or Modify the Working Directory in the UI

To set or modify the working directory in the Session Editor, enter the full path in the Working Directory field:



If the directory does not exist, "Directory not found locally" displays.

Set the Working Directory on the Command Line

When starting TotalView from a shell, set the working directory using the command line argument `-working_directory`, like so:

```
totalview -working_directory /tmp
```

Environment Variables for the Program

When loading the program from within TotalView, add any necessary environment variables into the **Environment variables for the program** field.

Figure 26, Setting Environment Variables

Program Environment

Environment variables for the program

[Enter line-separated NAME=VALUE pairs]

Either separate each argument with a space, or place each one on a separate line. If an argument contains spaces, enclose the entire argument in double-quotation marks.

At startup, TotalView reads in your environment variables to ensure that your program has access to them when the program begins executing. Use this field to add additional environment variables or to override values of existing variables.

TotalView does not display the variables that were passed to it when you started your debugging session. Instead, this field displays only the variables you added using this command.

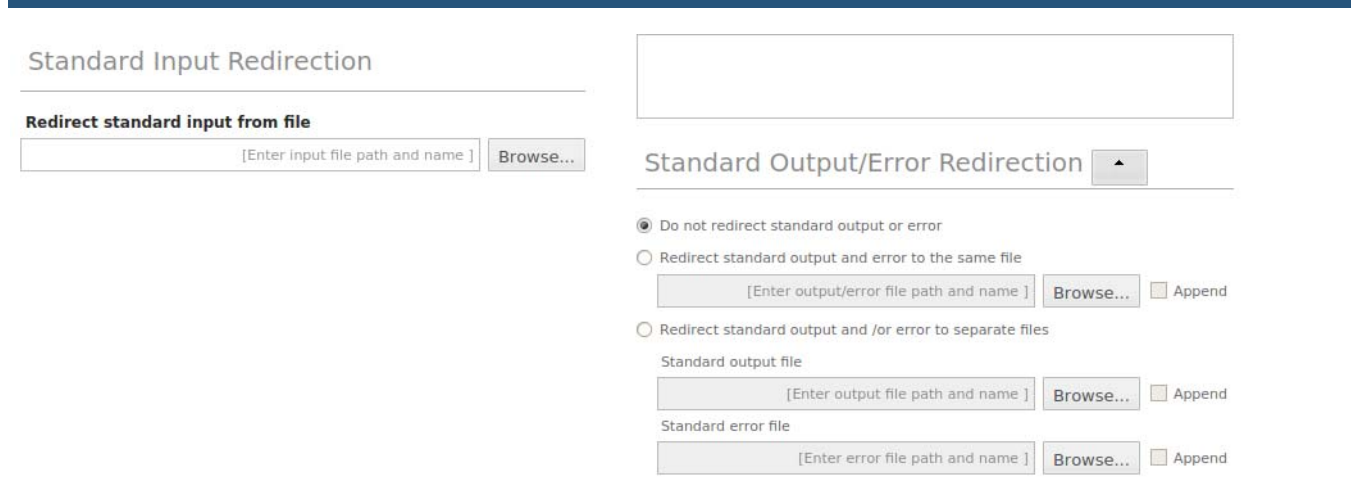
The format for specifying an environment variable is *name=value*. For example, the following definition creates an environment variable named **DISPLAY** whose value is **enterprise:0.0**:

```
DISPLAY=enterprise:0.0
```

Standard Input and Output

Use the controls in the **Standard Input Redirection** and **Standard Output/Error Redirection** sections to alter standard input, output, and error. In all cases, name the file to which TotalView will write or from which TotalView will read information. Other controls append output to an existing file if one exists instead of overwriting it or merge standard out and standard error to the same stream.

Figure 27, Debug Options for Standard Input



Modifying Arguments in an Open Session

All arguments or options that can be set while first configuring a session (see [Options and Program Arguments](#)) and can also be modified once the session has started.

NOTE: You can modify arguments in existing sessions only when debugging a program. You *cannot modify* arguments for existing sessions in which you have attached to a running process or are debugging a core or replay recording file.

Modify arguments in an existing session using either the UI or when loading a program from the command line, i.e. when entering **totalview <program_name>** into your shell. (See [Starting a Session from your Shell](#).)

To modify arguments in the UI:


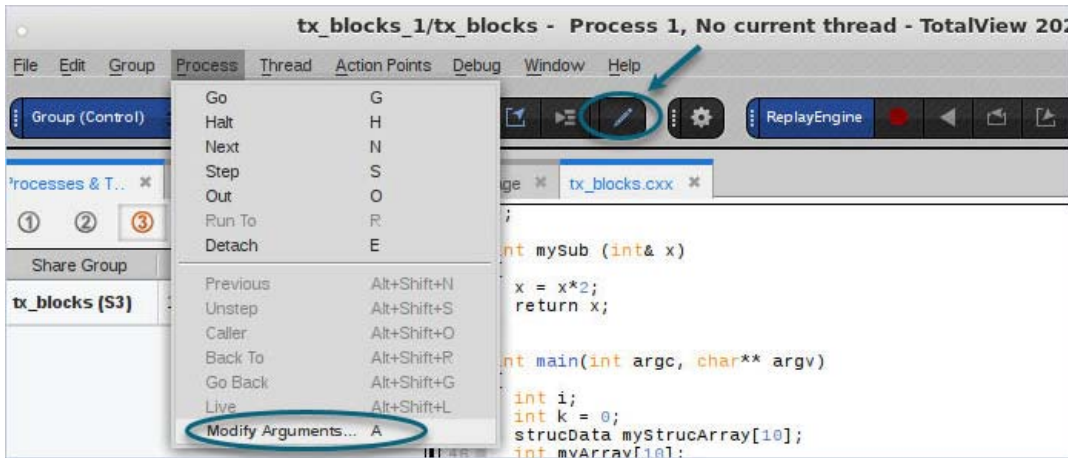
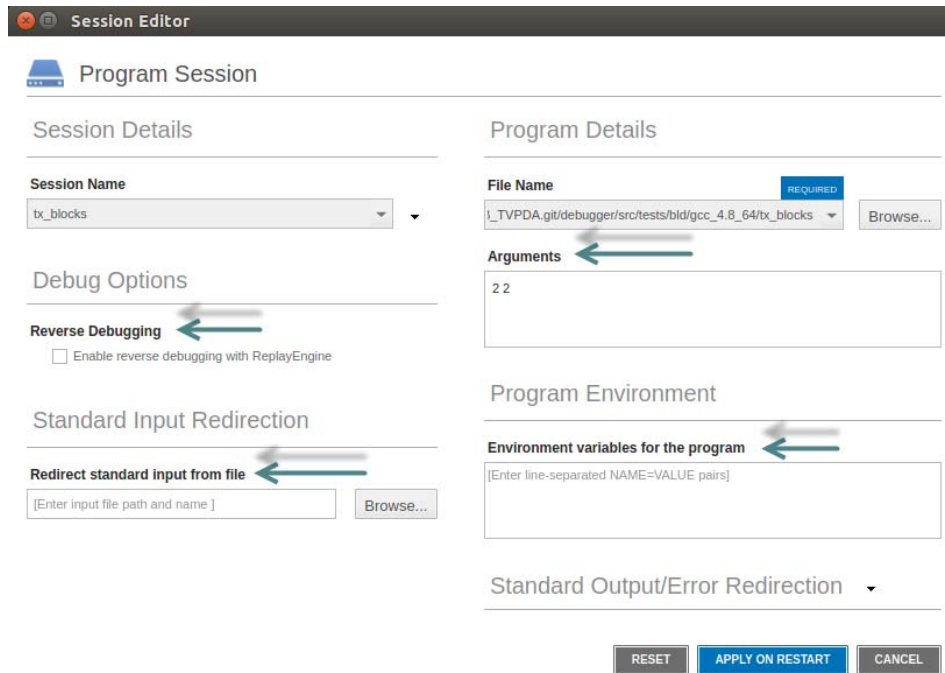
1. From within a debugging session, choose the **Process** menu, and then **Modify Arguments**. Alternatively, click the Modify Arguments () icon on the toolbar or press the shortcut key **A**.

Figure 28, Modify Arguments drop-down



The Session Editor launches.

Figure 29, Modifying Arguments in the Session Editor



2. Enter any modified arguments or options in either
 - **Reverse Debugging:** Toggle this on or off.
 - **Arguments:** Change any arguments to your program

- **Environment variables:** Enter or edit variables.
- **Standard input:** Enter or edit any input files.

NOTE: When modifying arguments within an open session, you cannot change the File Name or the Session Name, both of which are disabled.

3. Click **Apply on Restart**.



Modified arguments have no effect until you restart your program, selecting either **Go**, **Kill** or **Restart**.

Managing Sessions

TotalView saves the settings for each of your previously-entered debugging sessions, available in the Manage Debugging Sessions window of the Sessions Manager. Here, you can edit, duplicate or delete sessions as well as start a session and create new sessions.

You can also edit and create new sessions from any Sessions Window. See [Editing a Previous Session](#).

Access the Manage Sessions window, either from the **Start Page** by clicking **View All** to see all your sessions, or from **File > Manage Sessions**.

Figure 30, Accessing Manage Sessions from the Start Page

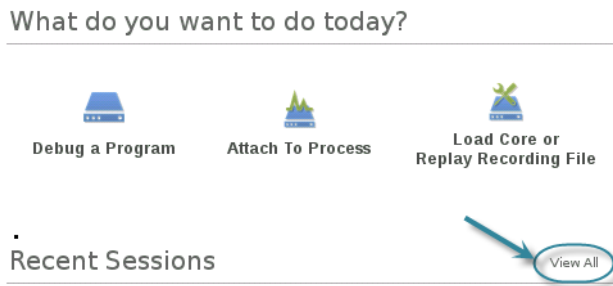
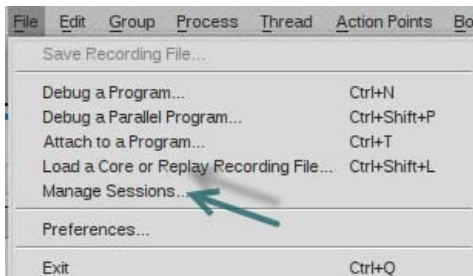
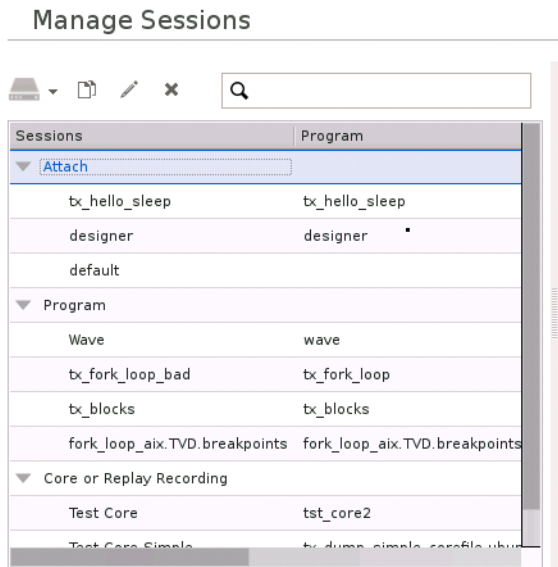


Figure 31, Accessing Manage Sessions from the File Menu



The Manage Sessions window launches listing all the sessions you have created.

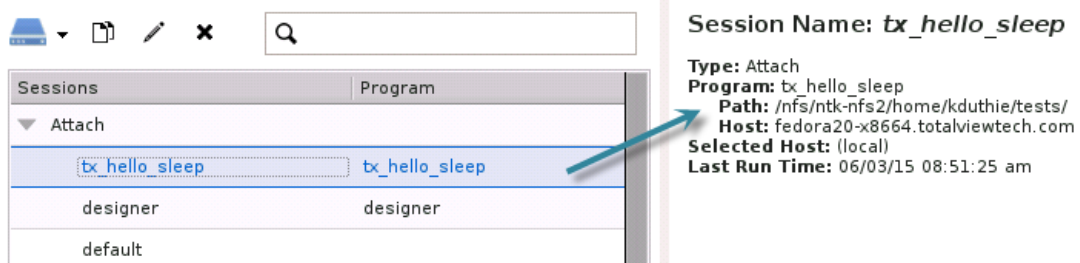
Figure 32, The Manage Sessions window



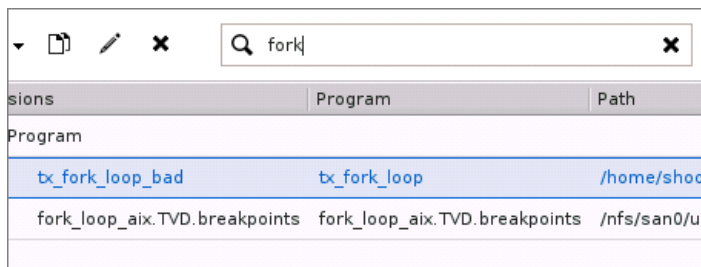
Use the Manage Sessions window to to:

- Display data about a session by selecting the session.

Manage Sessions



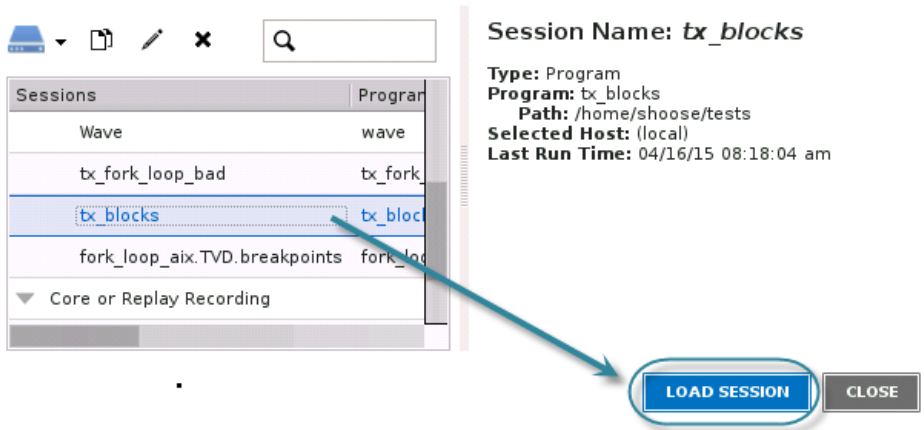
- Search for a session by entering a keyword in the search field.



- **Rename a session** by double-clicking on it and entering a new name.



- **Load a session** by clicking Load Session, which immediately opens that session in TotalView.



- **Edit, delete and duplicate sessions** using either the context-menu accessed by right-clicking on a session or the icons in the top toolbar:

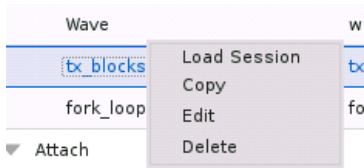




Table 1: Manage Sessions Icons

Icon	Action
	Creates a new debugging session, opening a drop-down menu for selecting: <ul style="list-style-type: none"> ■ Create a new Program Session ■ Set up an Attach Session ■ Create a session to load a Core or Replay Recording File
	Duplicates a session, naming it "<Session Name>Copy". You can rename and then edit this session.
	Edits a session, launching the appropriate window to change the session's configuration, either Program Session, Parallel Session, Attach to running programs, or Core or Replay Recording Session.
	Deletes a session.

Starting a Session from your Shell

There are a number of ways to start TotalView so a session is created and ready to begin when the debugger opens.

NOTE: If you need features currently not supported in the TotalView UI (see [Introducing TotalView](#)), you can launch Classic TotalView by invoking `totalview` with the flag `-classicui`. For example: `totalview -classicui`

Debugging a Program

`totalview executable`

Starts TotalView and loads the *executable* program.

Debugging a Parallel Program

`totalview -args mpirun -np 4 ./mpi_program`

Starts TotalView and loads a four-process MPI program.

Debugging a Core File

`totalview executable corefile`

Starts TotalView, loads the *executable* program, and an associated *corefile*. You can use wild cards in the core file name.

Debugging a Replay Recording File

`totalview executable replay-recording-file`

Starts TotalView, loads the *executable* program, and the *replay-recording-file* from a previous debugging session for which a ReplayEngine recording was saved to the named file.

Passing Arguments to the Program Being Debugged

`totalview executable -a args`

Starts TotalView and passes all the arguments following the **-a** option to the *executable* program. When using the **-a** option, it must be the last TotalView option on the command line. Delimit multiple arguments with spaces.

`totalview -args executable args`

Similar to above, but uses the command line option **-args** to specify that the executable program and arguments follow.

Loading a Session

`totalview -load_session session-name`

Starts TotalView and the named session.

RELATED TOPICS

Parallel preferences when debugging a parallel program [Parallel Configuration](#)

Starting TotalView on a Script

It is sometimes convenient to start TotalView on a shell script. For example, a typical use case might be a script that calls into a shared library, and you need to debug the shared library code; another case is a shell script that sets environment variables, then execs the application to debug.

Anywhere in the examples above that an executable can be specified, an interpreter script can be specified instead. The underlying interpreter, which must be a valid executable object file for the platform, is debugged, not the script itself.

On Unix, an interpreter script starts with a line that is similar to the following:

```
#!/ interpreter [ interpreter-arg ]
```

Where

- **#!** are the first two characters in the file.
- **interpreter** is the path to an executable object file or some other interpreter script.
- **interpreter-arg** is an optional argument to pass to the interpreter.

When the interpreter script is executed, the interpreter is invoked by the system as follows:

```
interpreter [ interpreter-arg ] script [ script-args ]
```

Here's a simple example:

```
% cat myscript.sh
#!/bin/sh -x
```

```
echo "$@"  
% ./myscript.sh a b c  
+ echo a b c  
a b c  
%
```

In the example above, the following command was executed:

```
/bin/sh -x ./myscript.sh a b c
```

Whenever TotalView is processing an executable file, it first checks to see if the file is an interpreter script. If the file starts with `#!`, it is treated as an interpreter script. The path to the interpreter is extracted from the script and used as the executable object file to debug. If the interpreter file is itself an interpreter script, TotalView repeats the procedure (up to 40 times) until it encounters an interpreter file that is not an interpreter script. If the procedure fails to find a valid executable object file for the platform, loading the script into the debugger will fail.

In most cases, the interpreter for the script does not directly contain the code you want to debug, and instead dynamically loads or executes the code to debug. TotalView contains several configuration settings that make it easier to plant breakpoints and stop in your code, as described by the Related Topics below. There are three common cases, where the interpreter script:

- Dynamically loads a shared library and calls into the code to debug (see the entries below regarding shared libraries and creating pending breakpoints)
- Excs the program containing the code to debug (relevant to exec handling)
- Runs the program containing the code to debug (relevant to fork handling)

RELATED TOPICS

Configuring dynamic library handling	Shared Libraries
Creating a pending breakpoint	Pending Breakpoints
Configuring exec handling	Exec Handling
Configuring fork handling	Fork Handling

Basic Debugging

This chapter illustrates some basic debugging tasks and is based on the shipped program, **expression**, located in the directory `installdir/toolworks/totalview.version/platform/examples`.

NOTE: We recommend that you follow the procedure in the README.TXT file in the examples directory to create a local copy of the examples and rebuild them in your environment.

This program takes expressions input by the user and evaluates them. For the purposes of this example, we'll instead redirect the standard input to read a file, **expr.tst**, also located in the examples directory. This file includes three simple expressions:

```
2+3
2*(4/5)
(1/2)-(3/4)
```

The first steps when debugging programs with TotalView are similar to those using other debuggers:

- Use the **-g** option to compile the program.
- Start the program under TotalView control.
- Start the debugging process, including setting breakpoints and examining your program's data.

The chapter introduces some of TotalView's primary tools, as follows:

- [Program Load and Navigation](#)
- [Stepping and Executing](#)
- [Setting and Running to a Breakpoint \(Action Point\)](#)
- [Examining Data](#)

Program Load and Navigation

This section discusses how to load a program and looks at the primary TotalView interface. It also illustrates some of TotalView's navigation tools.

Load the Program to Debug

Before starting TotalView, you must add TotalView to your PATH variable. For information on installing or configuring TotalView, see the *Classic TotalView Installation Guide*.

1. Start TotalView.

`totalview`

The Start Page launches.



2. Select **Debug a Program** to open the Program Session window.

The screenshot shows the 'Session Editor' window with the 'Program Session' tab selected. The interface is divided into several sections:

- Session Details:** Contains a 'Session Name' field with a dropdown menu showing 'Expression Example'.
- Debug Options:** Includes a 'Reverse Debugging' section with a checkbox labeled 'Enable reverse debugging with ReplayEngine' which is currently unchecked.
- Standard Input Redirection:** Features a 'Redirect standard input from file' section with a text input field containing '/home/marc/tvexamples/expr.tst' and a 'Browse...' button.
- Program Details:** Contains a 'File Name' field with a dropdown menu showing '/home/marc/tvexamples/expression' and a 'Browse...' button. A 'REQUIRED' label is present above the field. Below it is an 'Arguments' text area with the placeholder text '[Enter any program arguments. Ex. -option foo]'.
- Program Environment:** Includes an 'Environment variables for the program' section with a text area containing the placeholder '[Enter line-separated NAME=VALUE pairs]'.
- Standard Output/Error Redirection:** A dropdown menu is visible at the bottom of this section.

At the bottom right of the window, there are three buttons: 'RESET', 'LOAD SESSION', and 'CANCEL'.

3. Provide a name for the session in **Session Name** field. This can be any string.
4. In the **File Name** field, browse to and select the **expression** program, located in the directory *installdir/toolworks/totalview.version/platform/examples*.
5. In the **Standard Input Redirection** field, browse to and select the **expr.tst** file, also located in the examples directory. This provides the input required by the program. Leave all other fields and options as is.
6. Click **Load Session** to load the program into TotalView.

Note that this is the same as entering the path to the standard input file and program name as arguments when starting TotalView:

```
totalview -stdin expr.tst expression
```

(This invocation assumes that your **examples** directory is known to TotalView or that you are invoking TotalView from within the **examples** directory.)

RELATED TOPICS

[Loading programs](#)

[Loading Programs from the Session Editor](#)

RELATED TOPICS

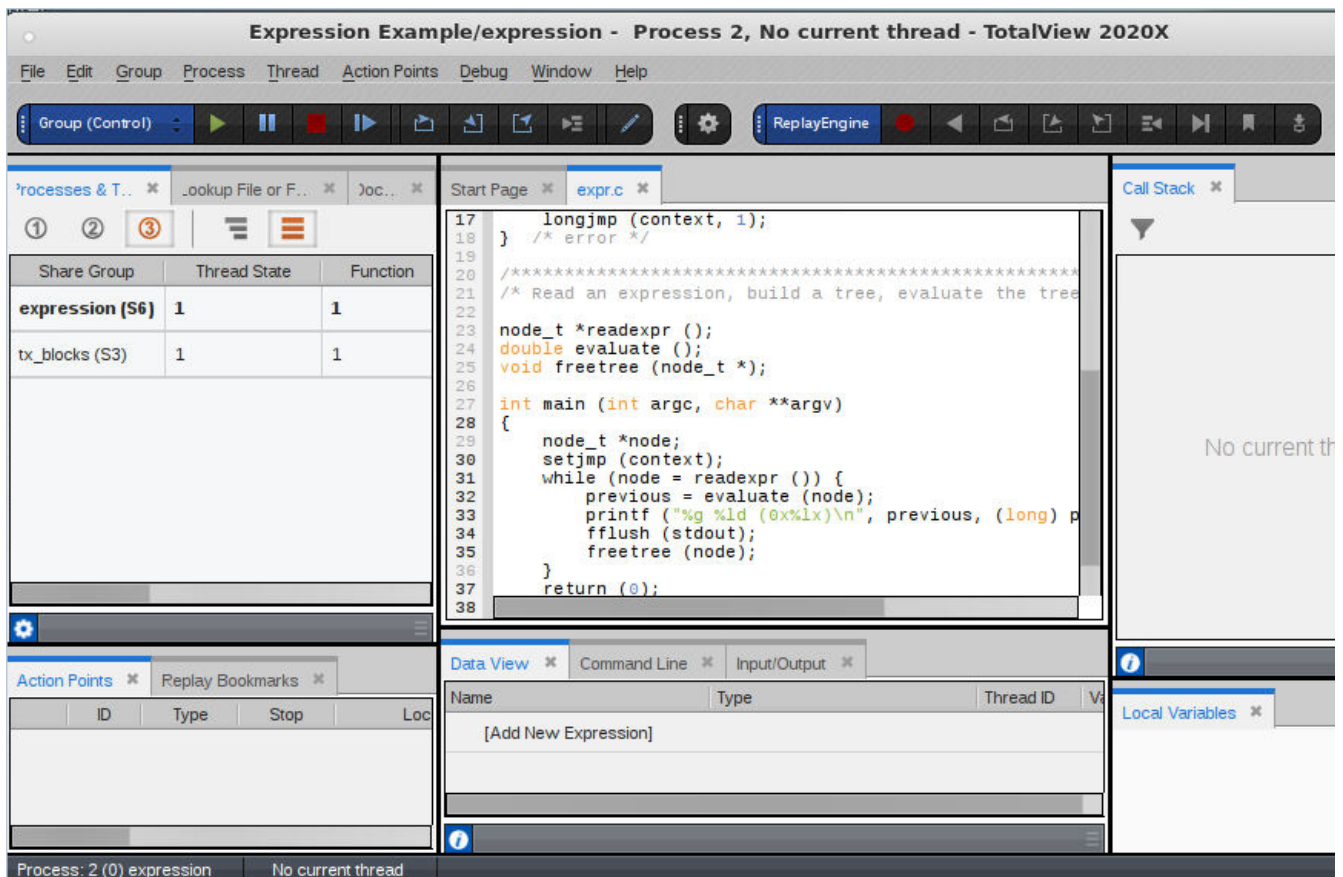
Starting a session from your shell [Starting a Session from your Shell](#)

Modifying arguments in an existing session [Modifying Arguments in an Open Session](#)

Initial Display

At startup, TotalView displays your program's code in the central area's Source pane, along with its default views: the Processes & Threads, Lookup File or Function, Action Points, Call Stack, Local Variables, Documents, and Data View.

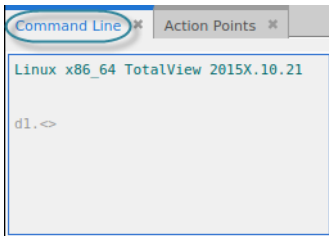
Figure 33, TotalView's Default Views



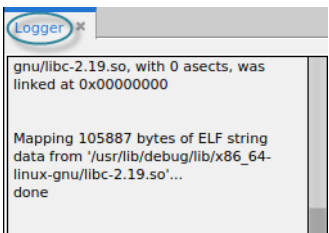
- The **Processes & Threads** view lists all processes and threads under TotalView control. You can use the **Window > Views** menu item to customize the displayed views.

Since the program has been created but not yet executed, there is no process or thread listed here.

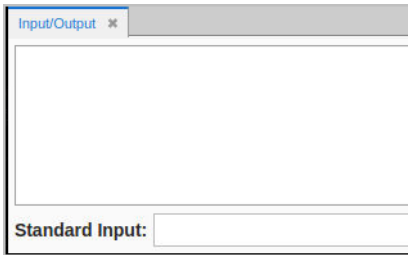
- The **Lookup File or Function** view takes any keyword search and returns a file or function from within your program's files.
- The **Documents** view displays all open files in the order in which their tabs appear in the central area.
- The **Replay Bookmarks** view displays bookmarks created to mark a point in program execution history.
- **Action Points** displays any breakpoints you set.
- The **Call Stack view** shows the backtrace of the thread that is currently in focus once the program is running.
- The **Local Variables view** displays information on the current thread's variables.
- The **Source view** in the central area displays your source code's **main()** function before execution.
- Several views are also visible at the bottom: Data View, Command Line, and Logger.
 - **Data View** enables you to evaluate expressions to observe your data while your program is running.
 - **Command Line**, which, when selected, displays a prompt for entering CLI commands:



- **Logger** which, when selected, displays logging output from TotalView:



- **Optional:** The **Input/Output** view displays **stdout** and **stderr**, and supports entering input directly into the user interface rather than only through the terminal. This view is turned off by default. To display it, select **Window > Views > Input/Output**.



RELATED TOPICS

Processes & Threads view	Customize the Display
Call Stack panel	The Call Stack, Local Variables, and Registers Views
Action Points	Setting and Managing Action Points (Breakpoints)
The CLI	Using the Command Line Interface (CLI)

Program Navigation

TotalView provides several ways to search your applications for text strings, files or functions. See [Program Navigation](#) for ways to navigate through your project.

Stepping and Executing

Here, we'll step through the program, using the buttons on the Debug toolbar.

Figure 34, Debug toolbar



The following sections explore how these work using the **expression** example.

NOTE: These procedures on stepping and execution can be performed independently of the other tasks in this chapter, but you must first load the program, as described in [Load the Program to Debug](#).

Simple Stepping

Here, we'll use the commands **Step**, **Run To**, **Next** and **Out**, and then note process and thread status.

1. Step

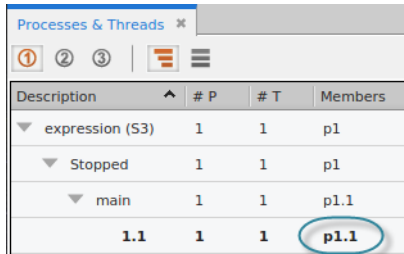
- Select **Step** () in the toolbar. TotalView stops the program just before the first executable statement, the call to **setjmp()**.

Note the yellow highlight and arrow show the current location of the Program Counter, or **PC**, in the Source pane.

```

27 int main (int argc, char **argv)
28 {
29     node t *node;
30     setjmp (context);
31     while (node = readexpr ()) {
32         previous = evaluate (node);
33         printf ("%g %ld (0x%lx)\n", previous, (long) previous, (long)
34                fflush (stdout);
35                freetree (node);
36     }
  
```

The process and thread status are displayed in the Processes & Threads pane:

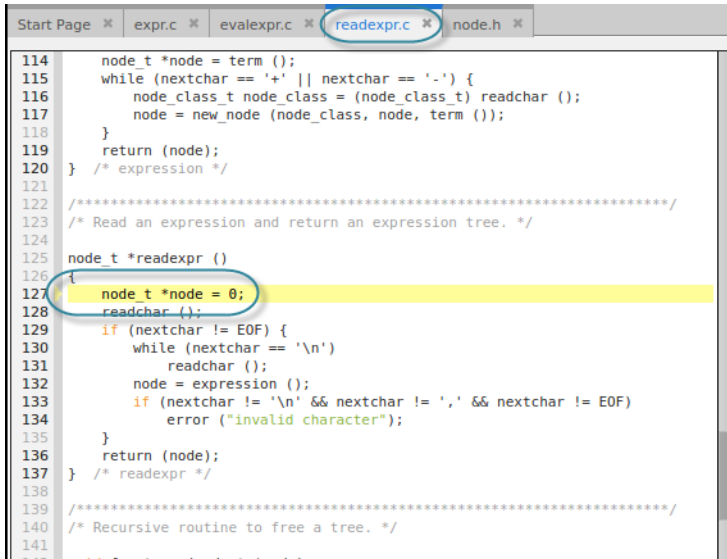


This program has just a single process **1** and thread, denoted by **1.1**, which reports that its status is **Stopped** in **main()**. The thread is in bold, because it is the active thread or the Thread of Interest (TOI). The status bar at the bottom also displays process/thread status, reporting that the TOI is in main().

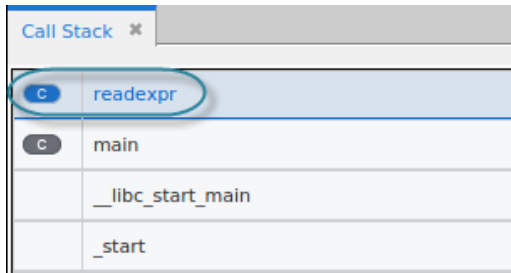


- Select **Step** again to advance to the **while** loop on line 31, and then select **Step** again to *step into* the **readexpr()** function. (**Next** would *step over*, or execute it.)

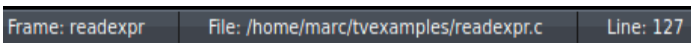
Because the **readexpr()** function is in a different file, TotalView opens the **readexpr.c** file and advances the PC to the first line of executable code in the **readexpr()** function.




Note that the **readexpr()** function now appears in the Call Stack view:



The status bar reports the location of the PC:



2. Run To

Select the **return()** statement at line 136, then click **Run To** () in the toolbar. The PC advances to line 136. Blue highlighting denotes a “run to” location.

```

134     error ("invalid character");
135     }
136     return (node);
137 } /* readexpr */
138

```

3. Out


Select **Out** () to execute the **return** statement and exit the function. The PC returns to the while condition inside **main()**:

```

29     node_t *node;
30     setjmp (context);
31     while (node = readexpr ()) {
32         previous = evaluate (node);
33         printf ("%g %ld (0x%lx)\n", previous, (long) previous, (long) previous);
34         fflush (stdout);
35         freetree (node);

```

4. Next

Click **Next** () on the toolbar. The **Next** command simply executes any executable code at the location of the PC. If that is a function call, it fully executes the function. If the PC is instead at a location within a function, it executes that line and then moves the PC to the next line.

In this case, the PC moves to the next executable line in **main()**, the assignment of the **evaluate()** function's return value on line 32:

```

30     setjmp (context);
31     while (node = readexpr ()) {
32         previous = evaluate (node);
33         printf ("%g %ld (0x%lx)\n", previous, (long) previous, (long) previous);
34         fflush (stdout);
35         freetree (node);

```

Now let's add some breakpoints and rerun the program.

Setting and Running to a Breakpoint (Action Point)

TotalView uses the term *action point*. A breakpoint is simply a type of action point that stops the execution of the processes and threads that reach it.

This section uses the **expression** example to set breakpoints.

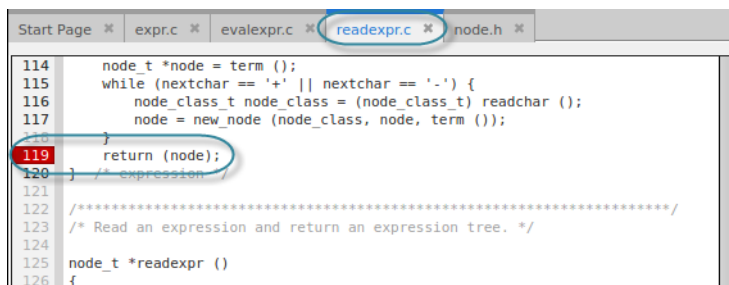
NOTE: These procedures on working with action points can be performed independently of the other sections in this chapter (which starts at [Basic Debugging](#)), but you must first load the program as described in [Load the Program to Debug](#).

Set and Control Breakpoints

1. Set a breakpoint.

Navigate to the function **readexpr()** on line 31 to open the file **readexpr.c**.

Set a breakpoint on line 119 by clicking on the line number in the Source pane. You can also set a breakpoint using the **Action Points > Set Breakpoint** menu item.



```

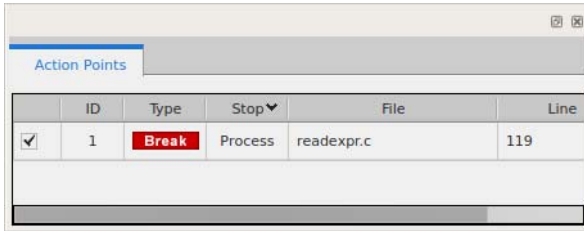
114     node_t *node = term ();
115     while (nextchar == '+' || nextchar == '-') {
116         node_class_t node_class = (node_class_t) readchar ();
117         node = new_node (node_class, node, term ());
118     }
119     return (node);
120 } /* expression */
121
122 /*****
123  /* Read an expression and return an expression tree. */
124
125 node_t *readexpr ()
126 {

```

The breakpoint will stop the program after executing the **expression()** function and just before returning the node object.

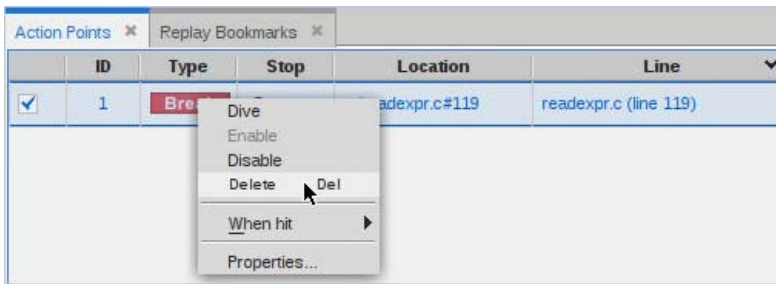
NOTE: Bold line numbers indicate known source locations. Breakpoints set on line numbers that are not bold are slid to the next valid source location and can become a valid source location if code is later loaded for that line.

The line number turns red in the Source pane and the action point is added to the Action Points view:

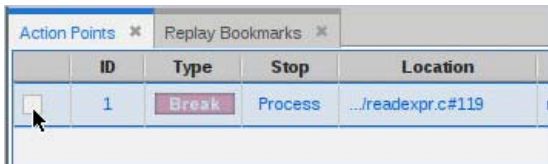


2. Delete/disable/enable a breakpoint.

- To delete the breakpoint, click the **red line number** in the Source Pane, and then re-add it by clicking again. You can also select it in the Action Points view, right-click for a context menu, and select **Delete** or simply hit the **Del** or **Delete** key on your keyboard.



- To disable a breakpoint, click the checkmark in the Action Points view. The icon dims to show it is disabled:



Click the checkmark again to re-enable it. Again, you can also disable or re-enable a breakpoint using the context menu.

NOTE: An action point also has a “When hit” option for controlling whether to stop all threads in a process or group, or just a single thread or process. See [Controlling an Action Point's Width](#) for more information.

RELATED TOPICS

Action points properties

About Action Points

Enabling/disabling action points

Managing and Diving on Action Points

Run Your Program and Observe the Call Stack

Run the program by clicking **Restart** (▶) on the toolbar.

The program halts at the breakpoint with the PC at line 119:

```

112 node_t *expression ()
113 {
114     node_t *node = term ();
115     while (nextchar == '+' || nextchar == '-') {
116         node_class_t node_class = (node_class_t) readchar ();
117         node = new_node (node_class, node, term ());
118     }
119     return (node);
120 } /* expression */
121

```

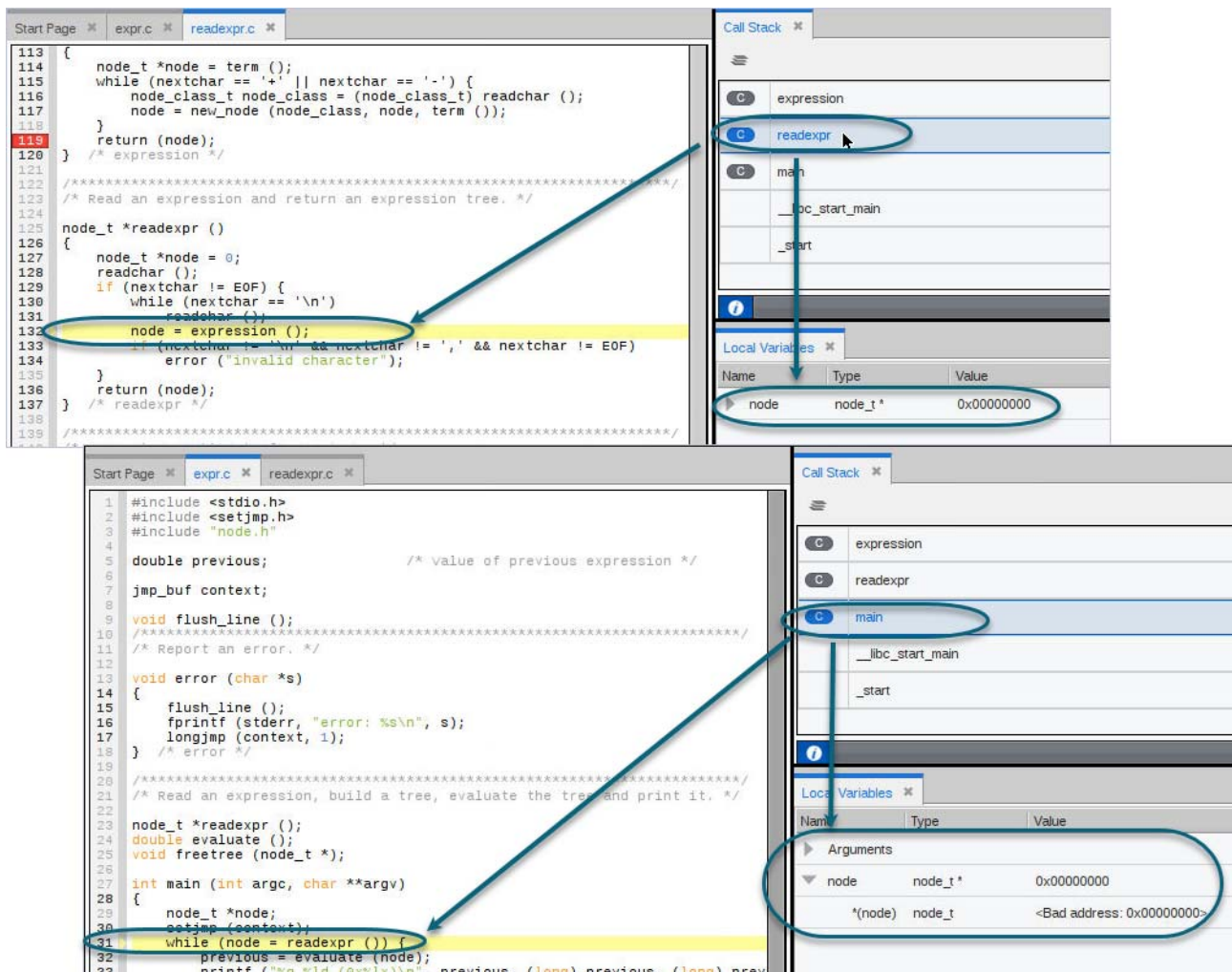
The Call Stack view shows that the program is stopped in the **expression()** function.

Call Stack *	
<input checked="" type="radio"/>	expression
<input type="radio"/>	readexpr
<input type="radio"/>	main
	__libc_start_main
	_start

The Local Variables view displays any local variables in scope.

Local Variables *		
Name	Type	Value
▶ node	node_t*	0x01396050 -> (node_t)

If you move the focus back up the call stack, the local variables in the Local Variables view update for the selected scope and the source related to that frame displays:



RELATED TOPICS

Action points, the Call Stack, and process/ thread state Action Point Width and Process/Thread State

Examining Data

Examining data is, of course, a primary focus of any debugging process. TotalView provides multiple tools to examine, display, and edit data.

You can quickly view data local to the selected call stack frame from within the Local Variables view. You can drill down by clicking on the left arrow to view compound data structures. To watch a variable's value change while the program runs, add it to the Data View where it remains even when no longer in scope.

In both the Local Variables view and the Data View, you can drill down on compound variables by clicking on the arrows to open and observe the data structures.

Add variables to the Data View where you can create expressions, cast data to another type, and perform other data-related tasks.

This section discusses viewing variables in the Local Variables view, and then using the Data View to look at global and compound data.

NOTE: These procedures on examining data can be performed independently of the tasks in other sections in this chapter, but you must first load the program ([Load the Program to Debug](#)).

Viewing Variables in the Local Variables View

First, we'll add a breakpoint so the program will stop execution and we can view data.

1. **Set a breakpoint.**

- Navigate to the word “**evaluate**” on line 32, in **main()**, to open **evalexpr.c**. Set a breakpoint on line 15 inside the **evaluate()** function at the assignment statement.

```

2  #include "node.h"
3
4  void error (char*);
5  /* Recursively evaluate an expression tree. */
6  /* Recursive routine to evaluate an expression tree. */
7
8  double evaluate (node_t *node)
9  {
10     double result;
11     if (!node)
12         error ("invalid expression tree");
13     switch (node->node_class) {
14     case nc_value:
15         result = (node->u.value);
16         break;
17     case nc_add:
18         result = (evaluate (node->u.node.left) +
19                 evaluate (node->u.node.right));
20         break;
21     case nc_subtract:
22         result = (evaluate (node->u.node.left) -
23                 evaluate (node->u.node.right));

```

NOTE: Disable any other breakpoints you have set, for this discussion.

- Click **Go** () on the toolbar. The program stops on the breakpoint. Now let's view some data.

2. View variables in the Local Variables view

The Local Variables view lists local variables. To view compound variables, click the left arrow.

Name	Type	Value
Arguments		
node	node_t *	0x01225010 -> (node_t)
...	node_t	(node_t)
	enum node_class	nc_value (46)
u	union <nameless...>	(union <nameless13>)
result	double	9.40007716767489e-317 <denormalized>

The Info view displays additional detail about the location of the stopped thread and the selected frame in the stack trace.

Function	evaluate
Source	...ild/rc/totalview/linux-x86-64/totalview/debugger/src/examples/evalexpr.c
Line	15
FP	0x7ffe776e30f0

3. View variables in a tooltip

In the Source pane or the Local Variables view, hover over the variable **result** to view a tooltip that displays its value:

```

8  double evaluate (node_t *node)
9  {
10     double result;
11     if (!node)
12         error ("invalid expression tree");
13     switch (node->node_class) {
14     case nc_value:
15         result = (node->u.value);
16         break;
17     case nc_result: result: 2.07376940296563e-317 <denormalized>
18         result = (evaluate (node->u.node.left) +
19                 evaluate (node->u.node.right));
20         break;

```

Name	Type	Value
Arguments		
node	node_t *	0x01225010 -> (node_t)
result	double	9.40007716767489e-317 <denormalized>

result
In scope:
File: /nfs/ntk-scratch/home/jenkins/build/rc/totalview/linux-x86-64/totalview/debugger/src/examples/evalexpr.c
Function: evaluate

Viewing Variables in the Data View

The Data View is a powerful tool that can list any variable in your program, along with its current or previous value and other information. This helps you to monitor variables as your program executes:

- View changing values of variables.
- Drill down into the nested structures of compound variables (which you can also do in the Local Variables view)
- Add global variables to the Data View by directly typing them in.

- Add expressions involving your program data.

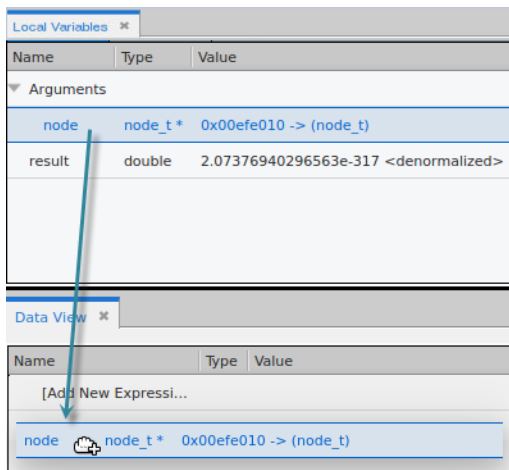
Watching Data Values Update

As you run your program, any data added to the Data View displays updated values.

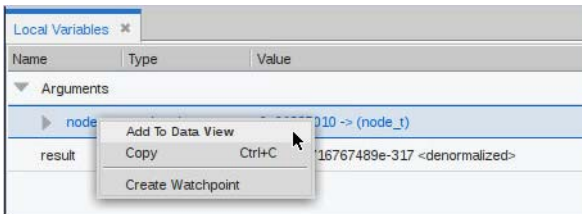
NOTE: This discussion assumes that you have set a breakpoint on line 15 in the `evaluate()` function and that you have clicked **Go**, as discussed in [Viewing Variables in the Local Variables View](#).

1. Add a Variable to the Data View

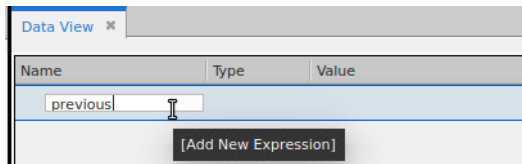
- From the Local Variables view, just drag a variable into the Data View.



Alternatively, right-click and select **Add to Data View**:



- **Add a global variable** by double clicking **Add New Expression** in the Data View and manually entering it:



Once entered, TotalView populates its type and value:

Name	Type	Value
previous	double	0
[Add New Expressi...		

2. View nested structures.

NOTE: You can also view compound structures in the Local Variables view.

The variable **node** is a compound type with several nested structures.

- To view any nested structure, click the right-arrow, which means that additional nested structures exist. Here, we've drilled into the node variable's union **u** to see that it contains a **left** and **right** struct, and a double **value**.

Name	Type	Value
previous	double	0
▼ node	node_t *	0x00efe010 -> (node_t)
▼ *(node)	node_t	(node_t)
node_class	enum nod...	nc_value (46)
▼ u	union <na...	(union <nameless11>)
▼ node	struct <n...	(struct <nameless10>)
left	struct nod...	0x4000000000000000 -> <Bad address: 0x4...
▶ right	struct nod...	0x00000000
value	double	2
[Add New Expressi...		

- Re-enable the breakpoint at line 119 in the **readexpr()** function by clicking on its checkbox in the Action Points view, or recreate it if necessary:

	ID	Type	Stop	File	Line
<input checked="" type="checkbox"/>	2	Break	Process	evalexpr.c	15
<input checked="" type="checkbox"/>	1	Break	Process	readexpr.c	119

- Click **Go** twice to run the program to the re-enabled breakpoint.

To see a variable's value, drill further down into the left or right variable:

node	struct <nameless10>	(struct <nameless10>)
left	struct node_t *	0x01ee5030 -> (struct node_t)
*(left)	struct node_t	(struct node_t)
node_class	enum node_class_t	nc_value (46)
u	union <nameless11>	(union <nameless11>)
node	struct <nameless10>	(struct <nameless10>)
left	struct node_t *	0x4010000000000000 -> <Bad address: 0x4010000000000000>
right	struct node_t *	0x00000000
*(right)	struct node_t	<Bad address: 0x00000000> (struct node_t)
value	double	4
right	struct node_t *	0x01ee5010 -> (struct node_t)

Remember that the data provided to the program consists of three simple expressions:

- 2+3
- 2*(4/5)
- (1/2)-(3/4)

At this point in the program's execution, the second expression is being read in. In the Data View, note that **left** has been assigned a value of 4. If you drill into **right**, it will have a value of 5, i.e. the input for the right side of the second expression.

3. View updated values.

- Click **Go** several times to run the program to the two breakpoints.

As the program reads in the expressions and evaluates them, the values change in the Data View:

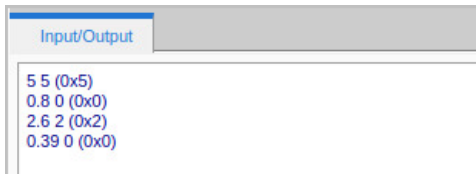
node	struct <nameless10>	(struct <nameless10>)
left	struct node_t *	0x010f90d0 -> (struct node_t)
*(left)	struct node_t	(struct node_t)
node_class	enum node_class_t	nc_value (46)
u	union <nameless11>	(union <nameless11>)
node	struct <nameless10>	(struct <nameless10>)
value	double	1
right	struct node_t *	0x010f90f0 -> (struct node_t)
*(right)	struct node_t	(struct node_t)
node_class	enum node_class_t	nc_value (46)
u	union <nameless11>	(union <nameless11>)
node	struct <nameless10>	(struct <nameless10>)
value	double	2
value	double	8.7930483525685e-317 <denormalized>
value	double	8.79303254246783e-317 <denormalized>
previous	double	0.8

4. View the output in the Input/Output view.

The output of the program goes to **stdout** when **fflush()** is called.

```
31 while (node = readexpr ()) {
32     previous = evaluate (node);
33     printf ("%g %ld (0x%lx)\n", previous, (long) previous, (long) previous);
34     fflush (stdout);
35     freetree (node);
36 }
37 return (0);
```

As each expression is evaluated and printed to **stdout**, when the **stdout** buffer is flushed, the Input/Output view shows the result of evaluating the expression. Run the program to the end to see the completed output.



RELATED TOPICS

[More on the Data View](#)

[The Data View](#)

[More on the Local Variables view](#)

[The Call Stack, Local Variables, and Registers Views](#)

Moving On

- For an overview on TotalView's new interface, see [An Initial Look at the Interface](#).
- For more information on ways to start and manage sessions in TotalView, see [Starting TotalView and Creating a Debugging Session](#) and [Creating and Managing Sessions](#).
- To use the Command Line Interface, see [Using the Command Line Interface \(CLI\)](#).
- To run your program backward, starting from the point of failure and working back in time to find the cause, see the [ReplayEngine User Guide](#).

Program Navigation

If your program is large or includes multiple source files, it may be difficult to find program elements you want to examine. TotalView provides several ways to search your applications for text strings, files or functions.

- **Navigating from within the Source Pane.** Navigate to a function from within the Source pane using the context menu.
- **Highlighting a String and the Find Function.** Search in the Source view by highlighting a string or using the Find function.
- **The Lookup File or Function View.** Use the **Lookup File or Function** view to search for files or functions.

You can also customize the system variables TotalView uses as part of the search path when searching for program elements. See [Search Path](#).

- **The Documents View** displays all open source files, updating automatically when files in the Source pane are opened or closed.

Navigating from within the Source Pane

You can navigate to a function in the Source pane.

Navigate to the **evaluate()** function call on **line 32**, by right-clicking and selecting **Navigate to File or Function** from the context menu.

```

27 int main (int argc, char **argv)
28 {
29     node_t *node;
30     setjmp (contxt);
31     while (node = readexpr ()) {
32         previous = evaluate (node);
33         printf ("%g %ld\n", previous, (long) previous);
34         fflush (stdout);
35         freetree (node);
36     }
37     return (0);
38 } /* main */
    
```

NOTE: If more than one result is found from the navigation operation, then all the results are shown in the Lookup File or Function view. You can easily click through the results to navigate to the location you want.

Note that since the **evaluate()** function is in a different file, **evalexpr.c**, that source file opens for viewing in addition to the source file already open containing the **main()** function.

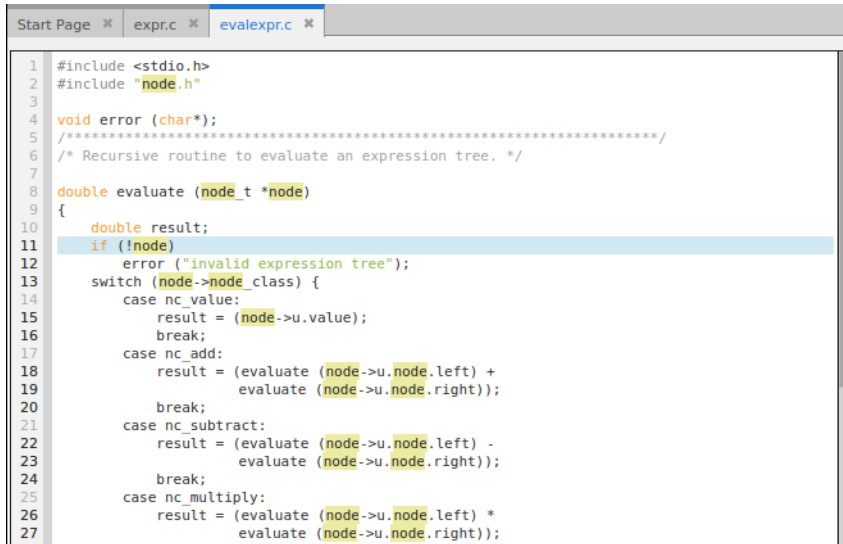
```

1 #include <stdio.h>
2 #include "node.h"
3
4 void error (char*);
5 /*****
6  /* Recursive routine to evaluate an expression tree. */
7
8 double evaluate (node_t *node)
9 {
10     double result;
11     if (!node)
12         error ("invalid expression tree");
13     switch (node->node_class) {
14     case nc_value:
15         result = (node->u.value);
16         break;
17     case nc_add:
18         result = (evaluate (node->u.node.left) +
19                 evaluate (node->u.node.right));
20         break;
21     case nc_subtract:
    
```

Highlighting a String and the Find Function

You can find specific text in Source views either by highlighting a string, or through the **Find** function.

When you click on some continuous string and it highlights, all other matching strings in that view are highlighted also. You can scroll through the text to find all other occurrences of the string. To remove the highlighting, simply click in any open space.

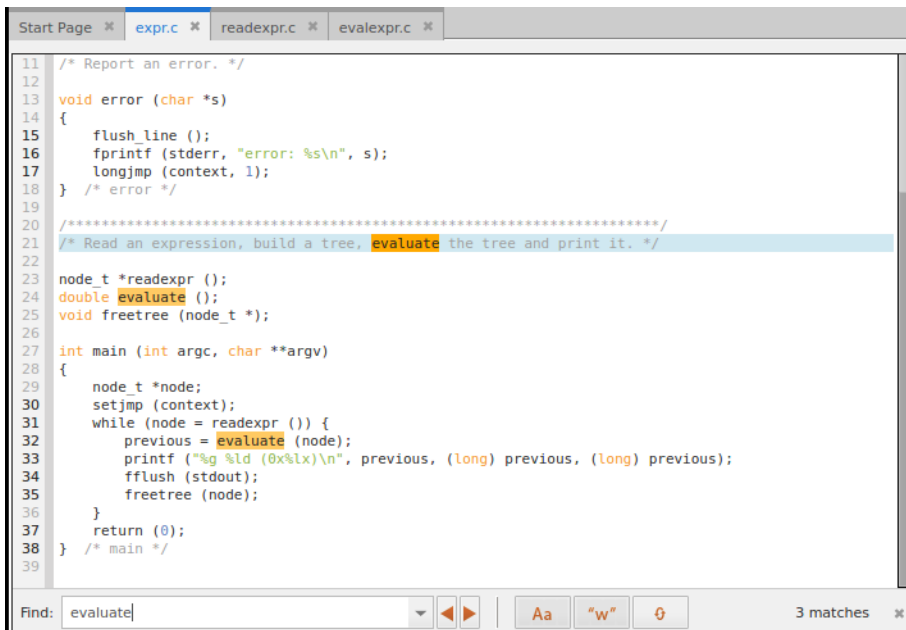


```

1 #include <stdio.h>
2 #include "node.h"
3
4 void error (char*);
5 /******
6 /* Recursive routine to evaluate an expression tree. */
7
8 double evaluate (node_t *node)
9 {
10     double result;
11     if (!node)
12         error ("invalid expression tree");
13     switch (node->node_class) {
14     case nc_value:
15         result = (node->u.value);
16         break;
17     case nc_add:
18         result = (evaluate (node->u.node.left) +
19                 evaluate (node->u.node.right));
20         break;
21     case nc_subtract:
22         result = (evaluate (node->u.node.left) -
23                 evaluate (node->u.node.right));
24         break;
25     case nc_multiply:
26         result = (evaluate (node->u.node.left) *
27                 evaluate (node->u.node.right));

```

To activate the **Find** function, enter **Ctrl-F** or select **Find** from the **Edit** menu.



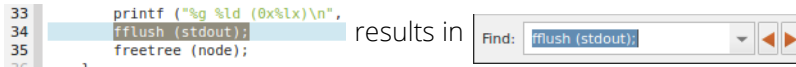
```





11 /* Report an error. */
12
13 void error (char *s)
14 {
15     flush_line ();
16     fprintf (stderr, "error: %s\n", s);
17     longjmp (context, 1);
18 } /* error */
19
20 /******
21 /* Read an expression, build a tree, evaluate the tree and print it. */
22
23 node_t *readexpr ();
24 double evaluate ();
25 void freetree (node_t *);
26
27 int main (int argc, char **argv)
28 {
29     node_t *node;
30     setjmp (context);
31     while (node = readexpr ()) {
32         previous = evaluate (node);
33         printf ("%g %ld (0x%x)\n", previous, (long) previous, (long) previous);
34         fflush (stdout);
35         freetree (node);
36     }
37     return (0);
38 } /* main */
39


```

Find: evaluate| 3 matches

If you select text in the Source view before activating the **Find** function, the selected string is loaded into the search text box.



The **Find** function tells you how many matching strings it has found in a given file, lets you easily move to the Next  (Ctrl-g) or Previous  (Ctrl-Shift-g) occurrence, and allows you to make the search case sensitive  or whole word .

You can also activate the **Wrap Search** button, , to wrap back to the beginning of the file after the last instance is reached.

The advantages of the **Find** function over simple highlighting are:

- In a large file, the Next and Previous controls save you tedious scrolling.
- The search can be refined using the case sensitive and whole word switches. Highlighting always applies to whole strings whereas **Find** can look for partial strings, such as “eval” rather than “evaluate”.
- If you move to another file in the Source view, the search is applied to that file so you can look for the string in the new file.

Previous search strings are saved in the dropdown menu at the end of the text field, and these are saved between debugging sessions, as is the state of the case sensitive and whole word buttons.

To close the Find function, press **Esc** or click the **X** at the right end of the window.

The Lookup File or Function View

The Lookup File or Function view takes any keyword search and returns a file or function from within your program's files.

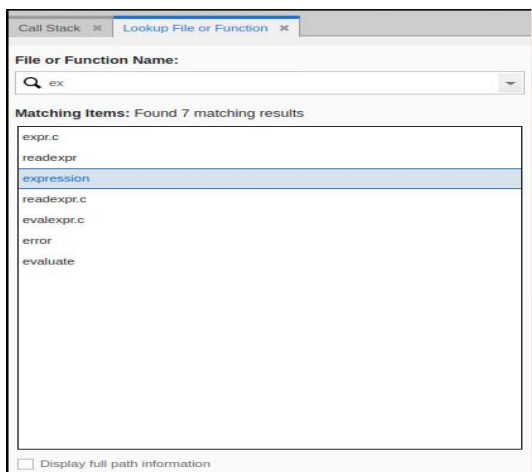
Open the **Lookup File or Function** view.

NOTE: If the Lookup view is not visible, select **Window > Views > Lookup File or Function** or use the keyboard shortcut **F**, to open it.



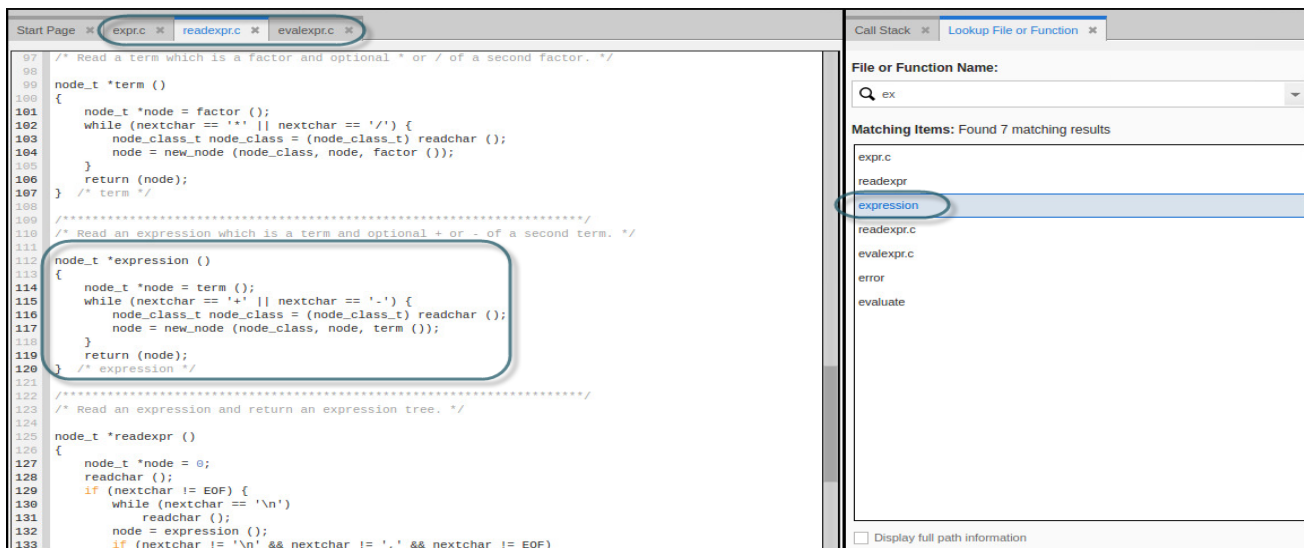
Searching for files or functions is based on keywords. The search encompasses the debugging symbols available in the executable files for the processes running in TotalView. This means that if your program links in shared libraries that were not compiled with debugging symbols, the search does not see files or functions related to these shared libraries. Also, if a dynamically shared library is not loaded because the program has not called that code, the debugging symbols from that library are not available.

For example, a search of "ex" returns a range functions and files:



Clicking on one of the results opens the source file in a tab in the Source pane. If the result is a function, the function definition is displayed in the source file. As you click through each returned result, the source appears in the same tab.

Double-click on a result to create a permanent tab for the source file.

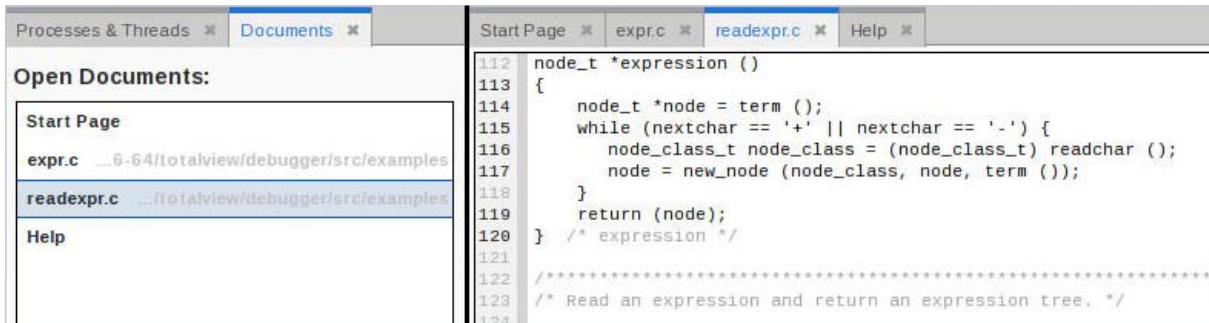


To display full path information in the results, select the checkbox at the bottom of the view.

The Documents View

The Documents view displays all active files or documents, useful when a program includes multiple files. The order of the displayed files matches that in the central area.

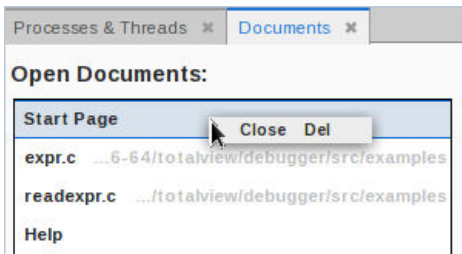
Figure 35, The Documents view



This view is open by default, but can be toggled off or on using the **Windows > Views > Documents** menu.

As you step through your code, the Documents view automatically displays any files that are opened, while removing files that are closed. It maintains the order in which the files appear in the central area, from left to right.

You can use the Documents view to close or delete files by right-clicking on the file to launch a context menu:



Closing a file also closes it in the central area.

PART II Debugging Tools and Tasks

- **Setting and Managing Action Points (Breakpoints)**

About TotalView's four types of action points: breakpoints, evalpoints, watchpoints, and barrier points.

- **Examining and Editing Data**

Using the Call Stack view, the VAR drawer, and the Data View.

- **The Processes & Threads View**

Using the Call Stack view, the VAR drawer, and the Data View.

- **Debugging Python**

Using TotalView to debug Python extensions.

- **Using the Command Line Interface (CLI)**

Using CLI commands via the Command Line view.

Setting and Managing Action Points (Breakpoints)

- About Action Points
- Breakpoints
- Evalpoints
- Watchpoints
- Barrier Points
- Controlling an Action Point's Width
- Managing and Diving on Action Points
- More on Action Points Using the CLI

About Action Points

TotalView employs the concept of *action points*, which specify an action to perform when a thread or process reaches a source line or machine instruction in your program.

TotalView supports four types of action points:

- A *breakpoint* stops execution of processes and threads that reach it. Other threads in the process also stop, and you can also indicate that you want other related processes to stop. Breakpoints are the simplest kind of action point.
- An *evalpoint* executes a code fragment when it is reached.
- A *watchpoint* monitors a location in memory and stops execution when it changes. A watchpoint can stop all the threads in a group or a process, or can include an expression to evaluate.
- A *barrier point* synchronizes a set of threads or processes at a location.

Action Point Properties





- You can independently enable or disable action points. A disabled action point isn't deleted; however, when your program reaches a disabled action point, TotalView ignores it.
- You can share action points across multiple processes or set them in individual processes.
- Action points apply to all the threads in a process. In a multi-process program, the action point's *width*, or scope, applies by default to all threads in all processes in a share group, i.e. those processes that share the same executable. You can narrow the width to stop just a single thread that executed to the breakpoint, or, conversely, broaden it to apply to all threads in all processes in the control group, which contains all share groups.
- TotalView assigns unique ID numbers to each action point. These IDs display in the **Action Points** view.

Figure 36, Action Points View

▼	ID	Type	Stop	Location	
☑	1	Break	Process	tx_blocks.cxx	48
☑	2	Break	Process	tx_blocks.cxx	54
☑	4	Watch	Group	4 bytes @ 0x7ffbeeb...	

Each type of action point is identified with a distinctive icon, as displayed in [Table 2](#).

Table 2: Action Point Types and Identifying Icons

Action Points Icons	Type of Action Point	How to Use
	Breakpoint	See Breakpoints
	Evalpoint	See Evalpoints
	Watchpoint	See Watchpoints
	Barrier point	See Barrier Points

NOTE: Conditional watchpoints can be created only in the CLI for this release.

Breakpoints

You can set breakpoints either directly in source by navigating to the location and clicking on the line, or by using the **At Location** dialog.

Setting Source-Level Breakpoints

Typically, you set and clear breakpoints before you start a process. To set a source-level breakpoint, select a line number in the Source view.

Source View Line Number Indicators

- A **bold line number** denotes that the compiler generated one or more line number symbols for the source line. Multiple symbols might be within a single image file, for example on a "for" loop statement. Or, the line number symbols might be spread across multiple image files if the source file was compiled into the executable, shared libraries, and/or CUDA code.
- **No bold** indicates that the compiler did not generate any line number symbols for the source line. However, you can still set a *sliding* or *pending* breakpoint at the line, which is useful if you know that code for that line will be dynamically loaded at runtime, for example, in a dynamically loaded shared library or a CUDA kernel launch.

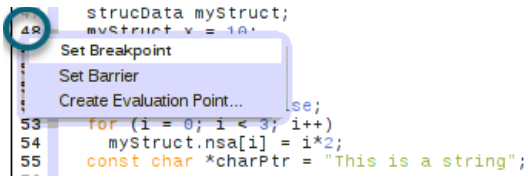
For example, [Figure 37](#) illustrates that source lines 48 and 49 both have line number symbols. Lines with no bold indicate that no executable code exists at those source lines yet (although you can set a sliding or pending breakpoint at those lines, discussed in [Pending Breakpoints](#) and [Sliding Breakpoints](#)).

Figure 37, Possible breakpoint locations in the Source view

```
45  structData myStrucArray[10];  
46  int myArray[10];  
47  structData myStruct;  
48  myStruct.x = 10;  
49  myStruct.y = 20;
```

Set a breakpoint either by:

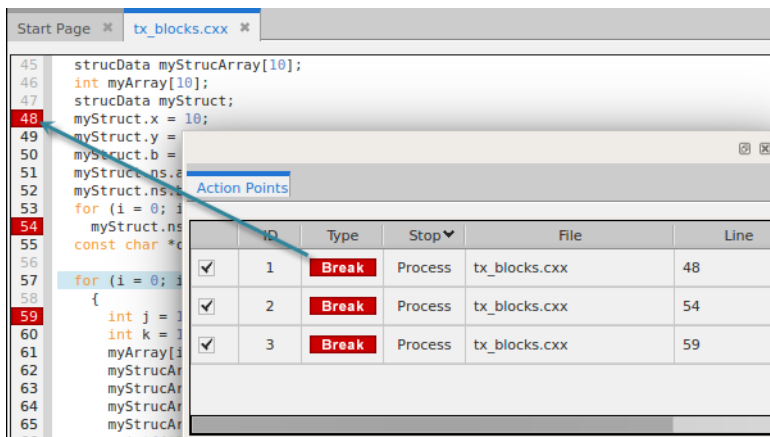
- Clicking directly on the line number in the Source view, *or*
- Right-clicking on the line number and using the context menu, *or*



- Clicking on a line in the Source View and then selecting the **Action Points > Set Breakpoint** menu item.

Once set, the breakpoint displays in the Action Points menu.

Figure 38, Set a breakpoint



Add any number of breakpoints before you run your program. (You can add or remove breakpoints at any point during your program's execution.)

NOTE: Setting a breakpoint on a line may cause that breakpoint to appear at many code locations. For example, setting a breakpoint on a line of templated code may cause the breakpoint to appear at all instances of that template.

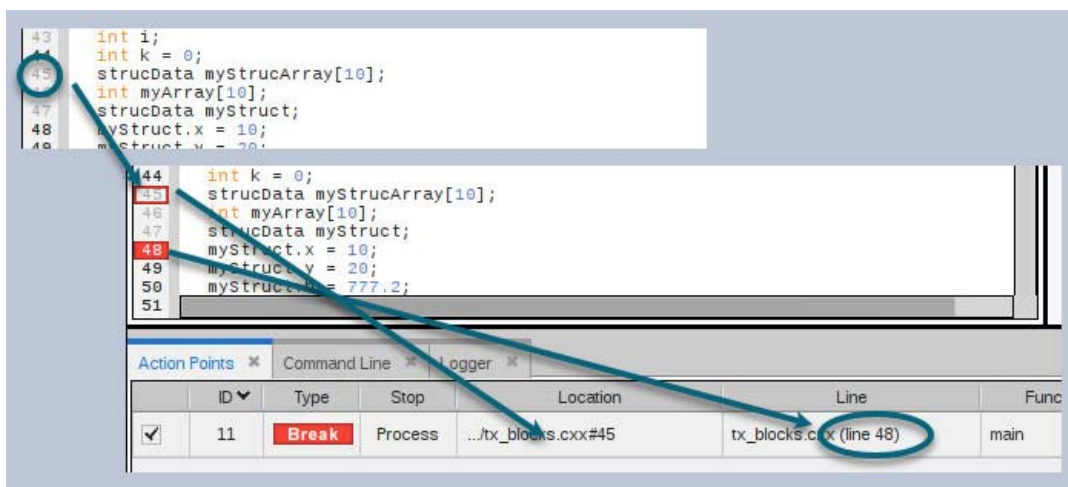
When you set a breakpoint or barrier point, it is defined by a *breakpoint expression*, also called a breakpoint specification, displayed in the Action Points tab for that breakpoint, or entered into the CLI (if created using the CLI). For more information, see [dbreak](#) in the *TotalView Reference Guide*.

Sliding Breakpoints

If you try to set a breakpoint in the Source view at a location with no bolded line, i.e., if there are no line number symbols for that source code line yet, TotalView automatically “slides” the breakpoint to the next line number in the source file that does have a line number symbol.

For example, in [Figure 39](#), a breakpoint was set at line 45 and slid to line 48 where there was a line number symbol. The Source view then displayed a hollow red box indicating that it slid, along with a solid red box at the slid location.

Figure 39, Sliding breakpoint



The Action Points Location column always displays the full breakpoint expression (in brackets). It also displays the “best” source file and line number it can currently find. TotalView does *not* change the original breakpoint expression, in the event that dynamically loaded code would be a better match later.

The breakpoint expression—pointing to line 45—is displayed in the Actions Points Location column as well as the location of the actual breakpoint at line 48. Retaining the original expression supports the situation in which a library that is dynamically loaded does have line number symbols at that location. As the program runs and dynamically loads code, TotalView reevaluates the breakpoint expressions, factoring in any new line number symbols it finds. If better-matching line number information is found, the address blocks in the breakpoint are updated to add the addresses of the new line number symbols, and possibly disable or invalidate old address blocks. This ensures that the breakpoint triggers for the most relevant source line.

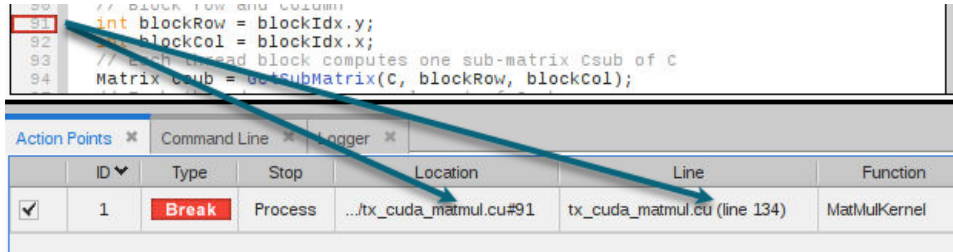
If TotalView cannot find a line number symbol following the line specified in the breakpoint expression, it creates a *pending* breakpoint. For example, this could occur when setting a breakpoint at the end of a source file. See [Pending Breakpoints](#) for information.

Dynamic Code Loading Example

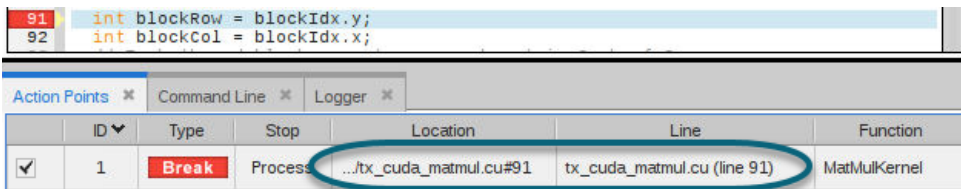
To see how this works, consider a program that will load code at runtime, such as when debugging CUDA code running on a GPU.

Figure 40 illustrates a breakpoint set at line 91 that has slid to line 134:

Figure 40, Sliding breakpoints when dynamically loading code



Once the program is running and the CUDA code is loaded, TotalView recalculates the breakpoint expression and is able to plant a breakpoint at line 91 in the CUDA code, which is an exact match for the breakpoint expression:



TotalView then disables the slid breakpoint at line 134 since it found a better match. Verify this using the **dactions** command in the CLI:

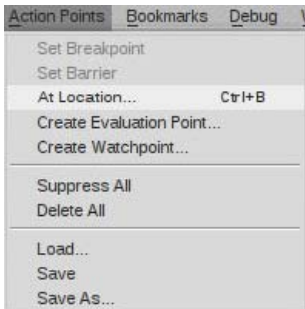
```
1.<> dactions -full -block_lines
1 shared action point for group 3:
1 [/home/totalview/cuda-example/tx_cuda_matmul.cu#91] Enabled
Address 0: [Disabled] MatMulKernel+0x18, tx_cuda_matmul.cu#134 (0x0040372d)
Address 1: [Enabled] MatMulKernel+0xae0, tx_cuda_matmul.cu#91 (Location not mapped)
Share in group: true
Stop when hit: process
```

Breakpoints at a Specific Location

You can quickly create breakpoints throughout your program using the **At Location** dialog, providing a convenient way to enter a valid breakpoint expression. Typical breakpoint expressions include a file and line number location (`myFile.cxx#35`), or a function signature (`main`). Use the **Create a pending breakpoint** option to create a pending breakpoint that becomes a breakpoint when TotalView finds the function or file.

For detailed information about the kinds of information you can enter in this dialog box, see the *BreakPoint Expressions* section in [dbreak](#) in the *TotalView Reference Guide*.

To enter a breakpoint expression, select **At Location** from the **Action Points** menu, or press **Ctrl-B**.



This launches the **At Location** dialog for entering a breakpoint expression. Here, a breakpoint is created at line 119 in the file `readexpr.c`.



NOTE: TotalView does not support ambiguous breakpoints in the UI, meaning that if it cannot find a location to set a breakpoint (or a barrier point), the breakpoint cannot be set.

Once you click **Create Breakpoint**, TotalView sets a breakpoint at the location. If you enter a function name, TotalView sets the breakpoint at the function's first executable line. If you check the **Create a pending breakpoint** box, TotalView creates the breakpoint as soon as it finds the function or file. See [Pending Breakpoints](#).

Setting Machine-Level Action Points

To set a machine-level action point, first display assembler code by selecting the menu item **Window > Show Assembler** or right-clicking in the Source View and selecting **Show Assembler** from the context menu.

Select an instruction and click the line number to set a breakpoint. Alternatively, right-click on a line number to set an eval or barrier point.

Action points set in the Assembler View have the same functionality and properties as those set in the Source View.

RELATED TOPICS

The Assembler view

[The Assembler View](#)

Source-level breakpoints

[Setting Source-Level Breakpoints](#)

Pending Breakpoints

TotalView supports pending breakpoints, useful when setting a breakpoint on code contained in a library that has not yet been loaded into memory.

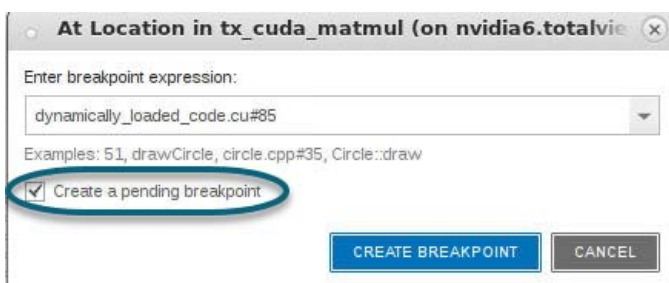
A pending action point is a breakpoint, barrier point, or evalpoint created with a breakpoint expression that does not yet correspond to any executable code. For example, a common use case is to create a pending function breakpoint with a breakpoint expression that matches the name of a function that will be loaded at runtime via **dlopen()**, CUDA kernel launch, or anything that dynamically loads executable code.

All four types of breakpoints can be pending (this includes line, function, methods in a class, and virtual function breakpoints). Further, a breakpoint may transition between pending to non-pending as image files are loaded, breakpoint expressions are reevaluated, address blocks are added, and invalid address blocks are nullified.

Set a pending breakpoint either on a function using the At Location dialog, or on a line number in the Source view.

Pending Breakpoints on a Function

When creating a breakpoint on a function using the **Action Points > At Location** dialog box, you are prompted to choose whether to set the breakpoint as pending if TotalView can't find the function:



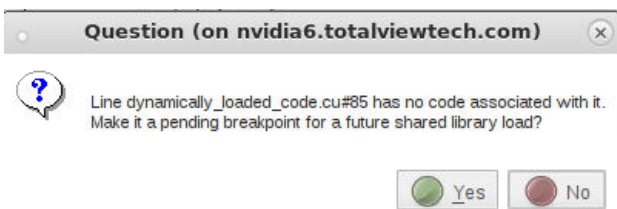
To immediately set a pending breakpoint, click **Create a pending breakpoint** directly in the **At Location** dialog. This is useful if you are sure that the function name you are entering is correct (even if TotalView can't find it) because it will be dynamically loaded at runtime. The breakpoint is set as pending:

ID	Type	Stop	Location
8	Break	Group	dynamically_loaded_code.cu#85 (pending)

(Note that, if you click the pending box when TotalView *can* find the function, it ignores the “Create Pending” request.)

Pending breakpoint prompt

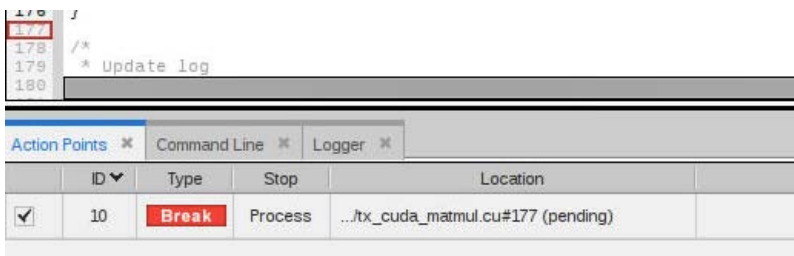
If you didn't select to create a pending breakpoint and the name you entered was not similar to any existing function, TotalView prompts to set a pending breakpoint.



Pending Breakpoints on a Line Number

Because TotalView “slides” a line number breakpoint to the next valid location (see [Sliding Breakpoints](#)), explicitly setting a line number pending breakpoint is rarely necessary. If, however, you know that there will be code at that spot, you can explicitly set a pending breakpoint in only these ways:

- By creating a line number breakpoint at a line near the end of a source file where the following lines have no line number symbols, but where you expect there to be dynamically loaded code at runtime. For example, here is a breakpoint set at line 177 just before the end of a file:



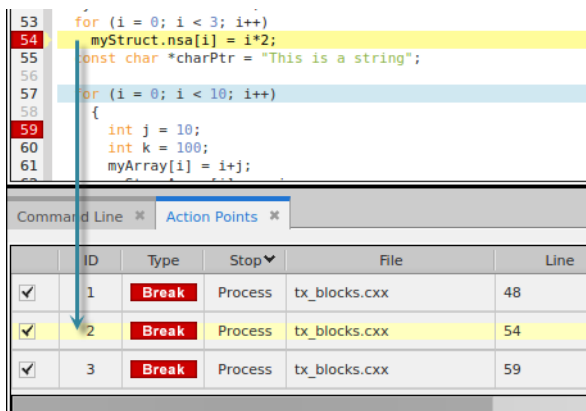
- In the **At Location** dialog box, type the file name and line number of a source file that has not been loaded yet. For example, `dynaloaded.c#42` where `dynaloaded.c` is compiled into a dynamically loaded shared library. TotalView posts a dialog box to confirm, unless "Create a pending breakpoint" is selected.

Conflicting Breakpoints

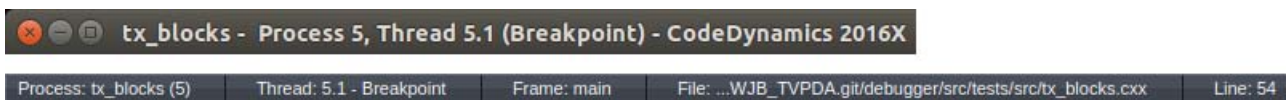
TotalView can place only one action point on an address. Because the breakpoints you specify are actually expressions, the locations to which these expressions evaluate can overlap or even be the same. Sometimes, and this most often occurs with pending breakpoints in dynamically loaded libraries, TotalView cannot predict when action points will overlap. If they do, TotalView enables only one of the action points and disables all others that evaluate to the same address. The action point that TotalView enables is that with the lowest actionpoint ID. The other overlapping action points are marked as "conflicted" in the Action Points pane and **dactions** output.

Breakpoints at Execution

Once you have added all your breakpoints, run or step through your program. When a breakpoint is hit, the Action Points view highlights the breakpoint that stopped execution.



Both the window title bar and the status bar at the bottom of the interface display information when a breakpoint is reached:

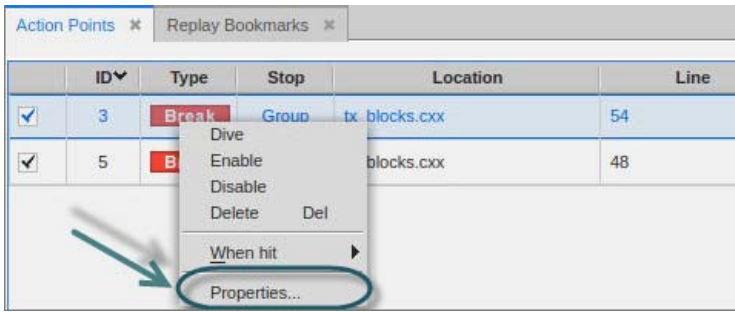


The above image shows execution stopped at a breakpoint. Similar information is displayed for the other action points, which currently are set through the command line interface as described in the section [More on Action Points Using the CLI](#).

Modifying a Breakpoint

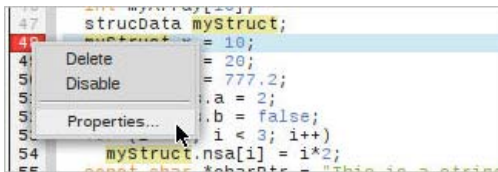
Modify a breakpoint by either:

- **In the Action Points view**, right-clicking on the breakpoint to bring up the context menu and selecting **Properties**.

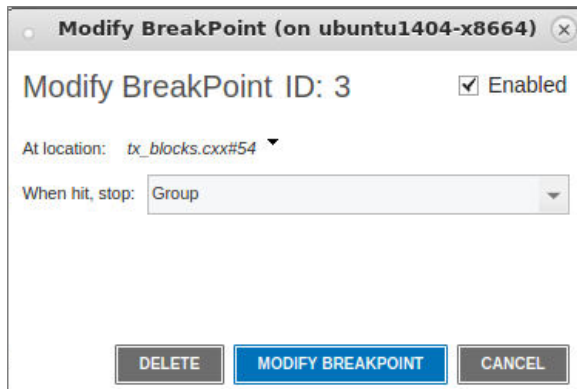


or

- In the Source view, right clicking on the breakpoint's line number to bring up the context menu and selecting **Properties**.

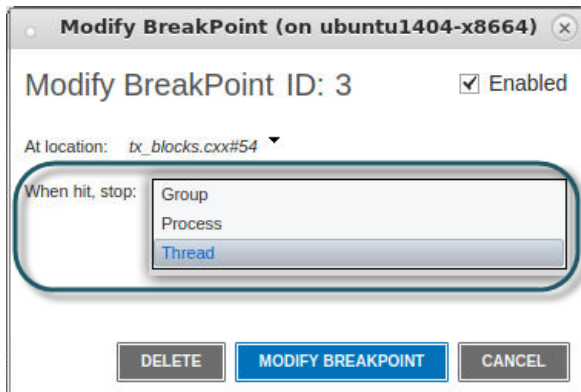


This launches the **Modify BreakPoint** dialog.



In this dialog, you can enable, disable or delete a breakpoint, view the breakpoint's location using the **At location** drop-down, or adjust the breakpoint's width under the **When hit, stop** drop-down.

For example:



The three width selections control how a breakpoint behaves in a multithreaded or multi-process program. Here's a summary:

- **Group:** Stops all running threads in all processes in the group.
- **Process:** Stops all the running threads in the process containing the thread that hit the breakpoint.
- **Thread:** Stops only the thread that first executes to this breakpoint.

RELATED TOPICS

About an action point's width

[Controlling an Action Point's Width](#)

Saving action points

[Saving and Loading Action Points](#)

About the **TV::stop_all** variable which indicates the default behavior for a breakpoint's width

The **TV::stop_all** variable

Setting Breakpoints When Using the fork()/execve() Functions

You must link with the dbfork library before debugging programs that call the **fork()** and **execve()** functions.

Debugging Processes That Call the fork() Function

By default, TotalView places breakpoints in all processes in a share group. When any process in the share group reaches a breakpoint, TotalView stops all processes in the control group. This means that TotalView stops the control group that contains the share group. This control can contain more than one share group.

To override these defaults, modify the breakpoint's width in the action point's properties, **Modify Breakpoint** dialog box.

```
CLI: dset SHARE_ACTION_POINT false
```

RELATED TOPICS

The Modify Breakpoint dialog box	Modifying a Breakpoint
More on an action point's width	Controlling an Action Point's Width
Linking with the dbfork library	Linking with the dbfork Library
More on share groups and control	How TotalView Creates Groups

Debugging Processes that Call the `execve()` Function

NOTE: You can control how TotalView handles system calls to `execve()`. See [Exec Handling](#).

Shared breakpoints are not set in children that have different executables.

To set the breakpoints for children that call the `execve()` function:

1. Set the breakpoints and breakpoint options in the parent and the children that do not call the `execve()` function.
2. Start the multi-process program using the **Group > Go** command.

When the first child calls the `execve()` function, TotalView displays the following message:

```
Processname has exec'd name.Do you want to stop it now?
```

G

3. Answer **Yes**.
(If you answer **No**, you won't have an opportunity to set breakpoints.)
4. Set breakpoints for the process.
After you set breakpoints for the first child using this executable, TotalView won't prompt when other children call the `execve()` function. This means that if you do not want to share breakpoints in children that use the same executable, set the breakpoint options using the action point properties dialog.
5. Select the **Group > Go** command.

Example: Multi-process Breakpoint

The following program excerpt illustrates the places where you can set breakpoints in a multi-process program:

```
1 pid = fork();
2 if (pid == -1)
3 error ("fork failed");
4 else if (pid == 0)
5 children_play();
```

```
6 else
7 parents_work();
```

The following table describes what happens when you set a breakpoint at different places:

Line Number	Result
1	Stops the parent process before it forks.
2	Stops both the parent and child processes.
3	Stops the parent process if the fork() function failed.
5	Stops the child process.
7	Stops the parent process.

RELATED TOPICS

[Linking with the dbfork library](#)

[Linking with the dbfork Library](#)

[Controlling system calls to **execve\(\)**.](#)

[Exec Handling](#)

Evalpoints

TotalView can execute code fragments at specified locations with a special type of action point called an *evalpoint*. TotalView evaluates these code fragments in the context of the target program, which means that you can refer to program variables and branch to places in your program.

Use evalpoints to:

- Include instructions that stop a process and its relatives. If the code fragment can make a decision whether to stop execution, it is called a *conditional breakpoint*, see [Creating Conditional Breakpoints](#).
- Test potential fixes or patches for your program; see [Patching Programs](#).
- Include a **goto** in C or Fortran that transfers control to a line number in your program. This lets you test program patches.
- Execute a TotalView function. These functions can stop execution and create barriers and countdown breakpoints. For more information on these statements, see [Using Built-in Variables and Statements](#).
- Set the values of your program's variables.

You can set an evalpoint at any source line that generates executable code. Valid source lines have a **bold** line number. When TotalView encounters an evalpoint, it executes the code in the evalpoint *before* the code on that line.

NOTE: If you call a function from an evalpoint and a breakpoint is within that function, TotalView stops execution at that breakpoint. Similarly, if an evalpoint is in the function, TotalView also evaluates that evalpoint.

Evalpoints modify only the processes being debugged—they do not modify your source program or create a permanent patch in the executable. If you save a program's action points, however, TotalView reapplies the evalpoint whenever you start a debugging session for that program.

RELATED TOPICS

Some examples of conditional breakpoints

[Creating Conditional Breakpoints](#)

Saving Action Points

[Saving Action Points to a File Using the CLI](#)

RELATED TOPICS

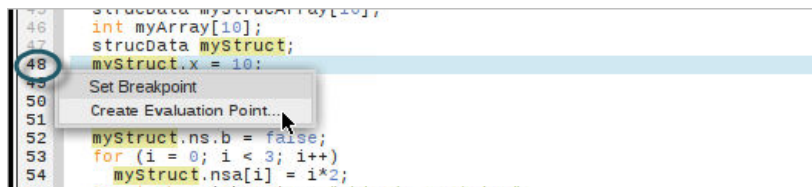
Using built-in TotalView statements to control execution [Using Built-In Statements](#)

Writing code for an expression [Using Programming Language Elements](#)

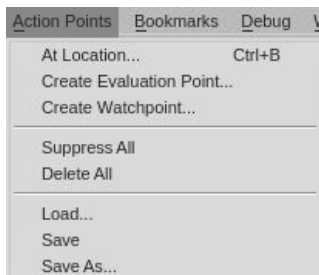
Setting an Evalpoint

You can set an evalpoint at any source line that generates executable code. Valid source lines have a **bold** line number. Create an evalpoint in these ways:

- **Through the CLI** (see [Evalpoints](#)).
- **From the Source view** by right-clicking on a line number to launch the context menu, and selecting **Create Evaluation Point**.

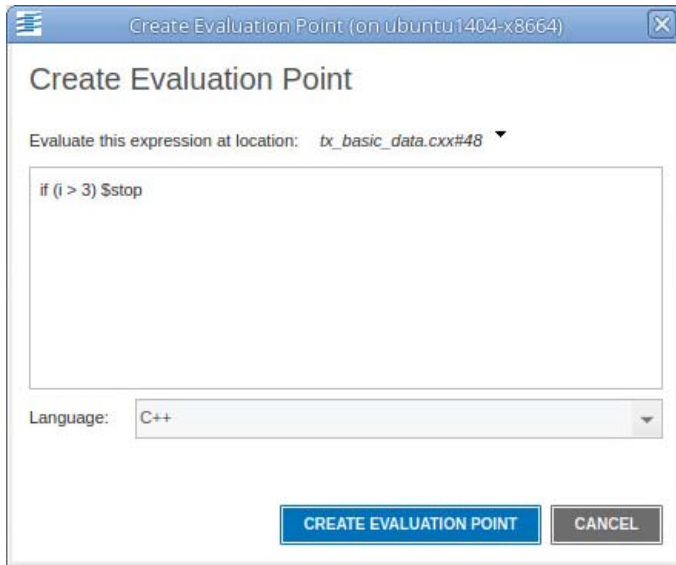


- **From the Action Points menu** by selecting **Create Evaluation Point**.

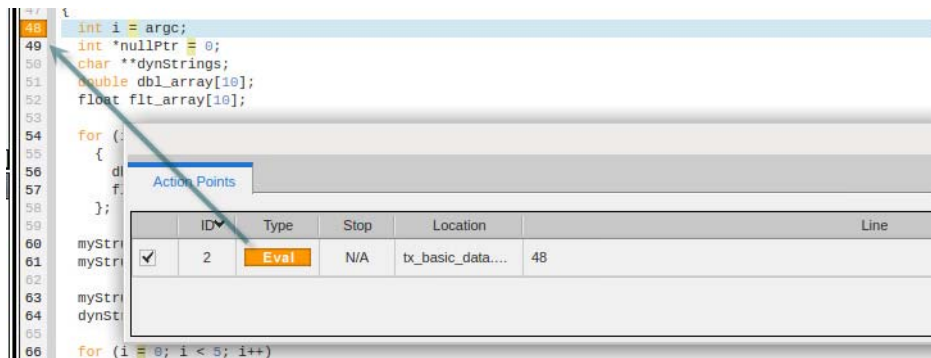


This launches the **Create Evaluation Point** dialog. Enter the code fragment and select the language (by default the language is chosen based on what TotalView detects as the language for the application). You can see the full path to the source file using the drop down arrow next to the file's name.

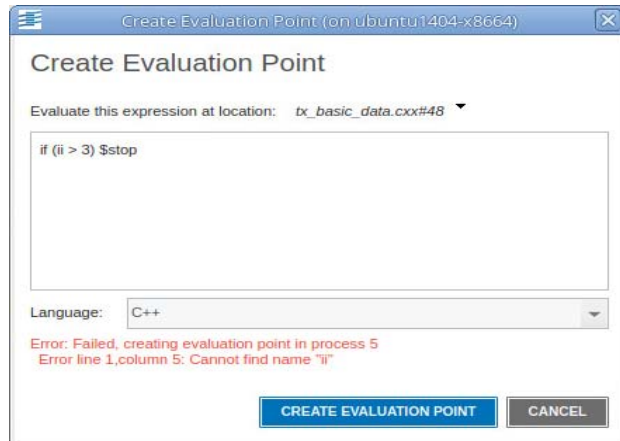
When you are satisfied with the code fragment, select **Create Evaluation Point**.



If TotalView successfully creates the evalpoint, the dialog closes, the Action Points view displays the evalpoint, and the Source view highlights the line number with the evalpoint in the corresponding color.



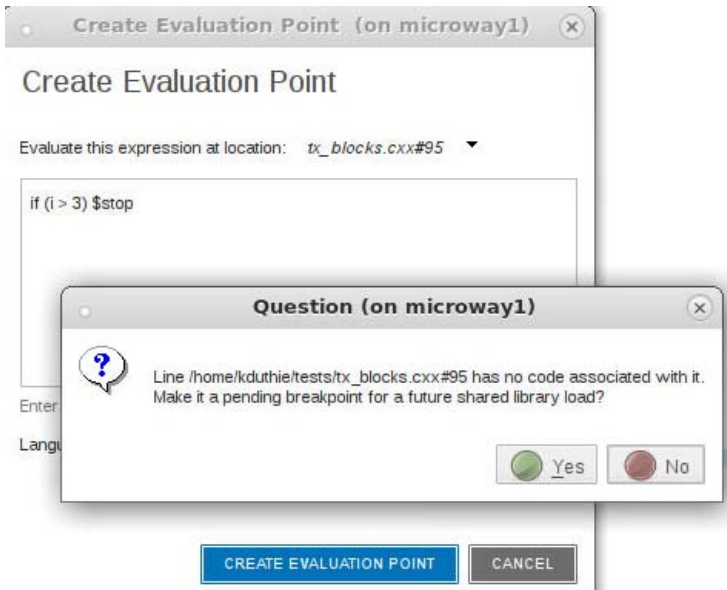
If TotalView cannot create the evalpoint, it displays an error message below the box.



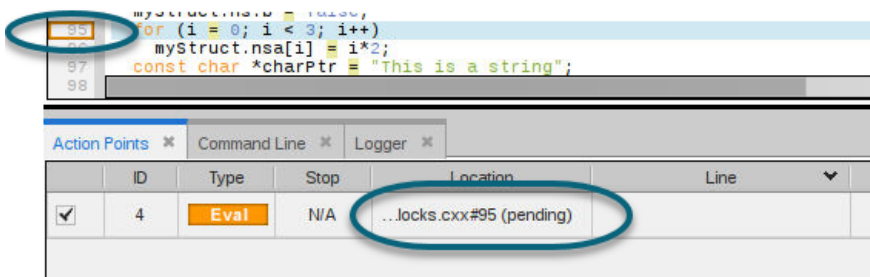
Creating a Pending Evalpoint

You can create a pending evalpoint at a location in your code that hasn't yet been loaded, for instance, when your program will dynamically load libraries at runtime. Setting a pending evalpoint is, essentially, allowing its expression to fail compilation when it is created. For example, it may reference a local variable in the code that will not be defined in the symbol table until the code is loaded and TotalView reads the debug symbols. When your program loads new code at an evalpoint location, TotalView will attempt to compile the expression. If the evalpoint expression still fails to compile, the evalpoint is handled like a breakpoint.

To create a pending evalpoint, simply create an evalpoint at the source line where you know dynamically loaded code will be. Once you click Create Evaluation Point in the Create Evaluation Point dialog and TotalView can't locate any debug symbols for that line, a pop-up prompts you to choose to create a pending evalpoint:



If you click Yes, the evalpoint is created, identified as pending with an orange boxed line in the Source view and a "pending" identifier in the Action Points tab:



A pending eval point is one in which:

- The underlying breakpoint is pending. In this case, TotalView is unlikely to be able to compile the expression (since the breakpoint is not yet instantiated), so it creates a pending evalpoint.
- A pending evalpoint has been explicitly created. Explicitly creating a pending evalpoint is useful when an evalpoint is intended to be set in dynamically loaded code (such as CUDA GPU code), and so the breakpoint slides to the host code before runtime.

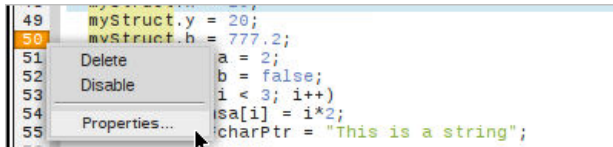
Note that the "Create a pending evalpoint" flag sticks to the evalpoint for the duration of the debug session. The flag is not saved with the eval point when TotalView saves action points; however, when restoring the action points, TotalView will set the flag if the underlying breakpoint needed to slide or was pending.

For more information on pending breakpoints, see [Pending Breakpoints](#).

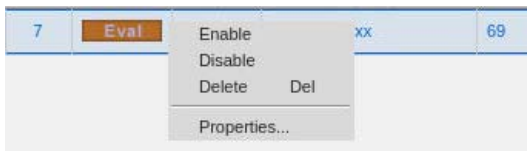
Modifying an Evalpoint

Modify an evalpoint by either:

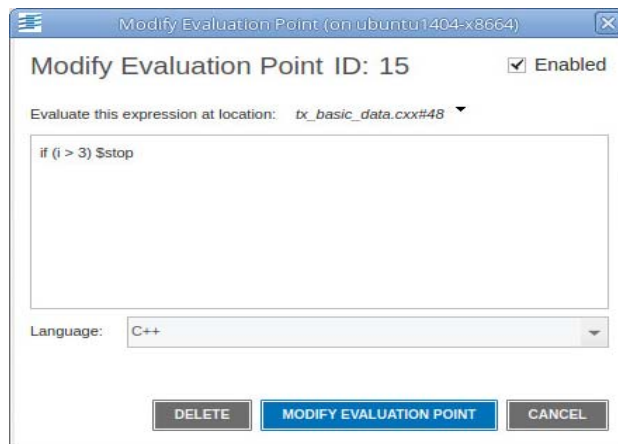
- In the **Source view**, right-clicking on the evalpoint and selecting **Properties**.



- In the **Action Points menu**, right clicking on the evalpoint to launch the context menu and selecting **Properties**.



This launches the **Modify Evaluation Point** dialog. From here you can change the code, language, or whether or not the evalpoint is enabled.



Creating Conditional Breakpoints

The following are examples for creating conditional breakpoints:

- This example defines a breakpoint that is reached whenever the **counter** variable is greater than 20, but less than 25:

```
if (counter > 20 && counter < 25) $stop;
```

- This example defines a breakpoint that stops execution every tenth time that TotalView executes the **\$count** function

```
$count 10
```

- The following example defines a breakpoint with a more complex expression:

```
$count my_var * 2
```

When the **my_var** variable equals **4**, the process stops the eighth time it executes the **\$count** function. After the process stops, TotalView reevaluates the expression. If **my_var** equals 5, the process stops again after the process executes the **\$count** function ten more times.

The TotalView internal counter is a static variable, which means that TotalView remembers its value every time it executes the evalpoint. Suppose you create an evalpoint within a loop that executes 120 times and the evalpoint contains **\$count 100**. Also assume that the loop is within a subroutine. As expected, TotalView stops execution the 100th time the evalpoint executes. When you resume execution, the remaining 20 iterations occur.

The next time the subroutine executes, TotalView stops execution after 80 iterations because it will have counted the 20 iterations from the last time the subroutine executed.

There is good reason for this behavior. Suppose you have a function that is called from several places within your program. Because TotalView remembers every time a statement executes, you could, for example, stop execution every 100 times the function is called. In other words, while **\$count** is most often used within loops, you can use it outside of them as well.

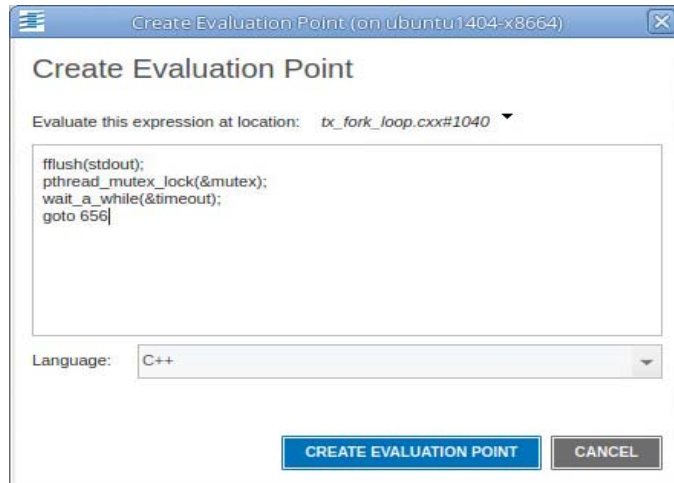
For descriptions of the **\$stop**, **\$count**, and variations on **\$count**, see [Using Built-in Variables and Statements](#).

Patching Programs

Evalpoints let you patch your programs and route around code that you want replaced, supporting branching around code that you don't want your program to execute and adding new statements. In many cases, correcting an error means that you will do both: use a **goto** to branch around incorrect lines, and then add corrections.

For example, suppose you need to change several statements. Just add these to an action point, then add a **goto** (C) or **GOTO** (Fortran) statement that jumps over the code you no longer want executed. For example, the evalpoint in [Figure 41](#) executes three statements and then skips to line 656.

Figure 41, Evalpoint expression with GOTO



Branching Around Code

The following example contains a logic error in which the program dereferences a null pointer:

```
1 int check_for_error (int *error_ptr)
2 {
3 *error_ptr = global_error;
4 global_error = 0;
5 return (global_error != 0);
6 }
```

The error occurs because the routine that calls this function assumes that the value of **error_ptr** can be 0. The **check_for_error()** function, however, assumes that **error_ptr** isn't null, which means that line 3 can dereference a null pointer.

Correct this error by setting an evalpoint on line 3 and entering:

```
if (error_ptr == 0) goto 4;
```

If the value of **error_ptr** is null, line 3 isn't executed. Note that you are not naming a label used in your program. Instead, you are naming one of the line numbers generated by TotalView.

Adding a Function Call

The example in the previous section routed around the problem. If all you wanted to do was monitor the value of the **global_error** variable, you can add a **printf()** function call that displays its value. For example, the following might be the evalpoint to add to line 4:

```
printf ("global_error is %d\n", global_error);
```


TotalView executes this code fragment before the code on line 4; that is, this line executes before **global_error** is set to 0.

Correcting Code

The following example contains a coding error: the function returns the maximum value instead of the minimum value:

```
1 int minimum (int a, int b)
2 {
3 int result; /* Return the minimum */
4 if (a < b)
5 result = b;
6 else
7 result = a;
8 return (result);
9 }
```

Correct this error by adding the following code to an evalpoint at line 4:

```
if (a < b) goto 7; else goto 5;
```

This effectively replaces the **if** statement on line 4 with the code in the evalpoint.

Using Programming Language Constructs

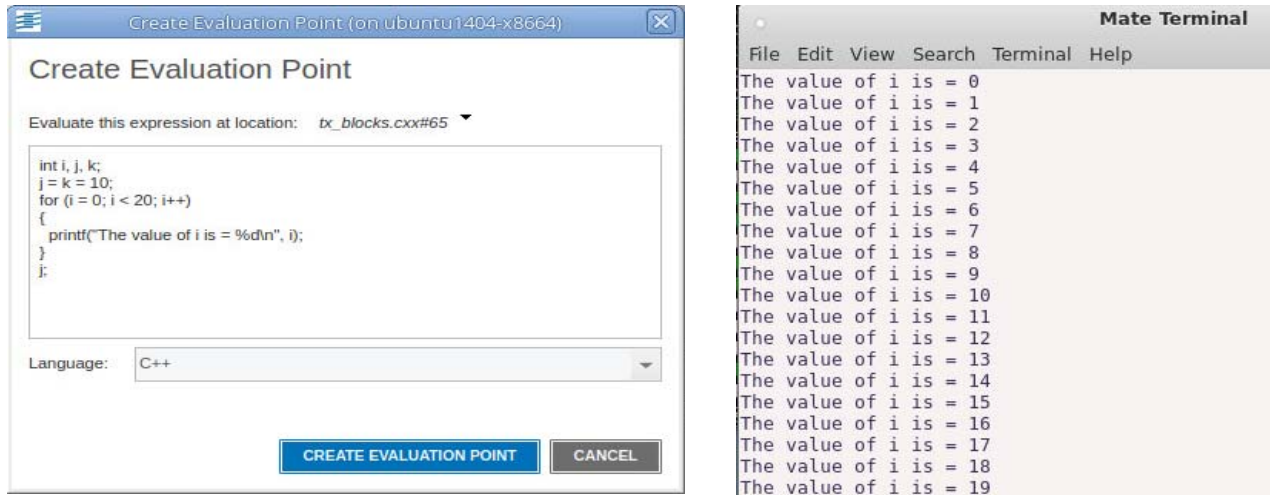
You can also use programming language constructs in an evalpoint. For example, here's a trivial example of code that can execute:

```
int i, j, k;
j = k = 10;
for (i=0; i< 20; i++)
{
j = j + access_func(i, k);
}
j;
```

This code fragment declares a couple of variables, runs them through a **for** loop, then displays the value of **j**. In all cases, the programming language constructs being interpreted or compiled within TotalView are based on code within TotalView. TotalView is not using the compiler you used to create your program or any other compiler or interpreter on your system.

Notice the last statement in the Create Evaluation Point dialog on the left in [Figure 42](#). The results are printed in the shell in which TotalView is running, displayed on the right.

Figure 42, Displaying the Value of the Last Statement



TotalView assumes that there is always a return value, even if it's evaluating a loop or the results of a subroutine returning a void. The results are, of course, not well-defined. If the value returned is not well-defined, TotalView returns a zero.

The code within an evalpoint does not run in the same address space as that in which your program runs. Because TotalView is a debugger, it knows how to reach into your program's address space. The reverse isn't true: your program can't reach into the TotalView address space. This forces some limitations upon what you can do. In particular, you cannot enter anything that directly or indirectly needs to pass an address of a variable defined within the TotalView expression into your program. Similarly, invoking a function that expects a pointer to a value and whose value is created within TotalView can't work. However, you can invoke a function whose parameter is an address and you name something within that program's address space. For example, you could say something like **adder(an_array)** if **an_array** is contained within your program.

Watchpoints

TotalView can monitor the changes that occur to memory locations with a special type of action point called a *watchpoint*. Watchpoints are most frequently used to find a statement in your program that is writing to inappropriate places. This can occur, for example, when processes share memory, and more than one process writes to the same location. It can also occur when your program writes off the end of an array or when your program has a dangling pointer.

Topics in this section are:

- [Creating Watchpoints](#)
- [Modifying Watchpoints](#)
- [Watching Memory](#)
- [Triggering Watchpoints](#)
- [Using Watchpoint Expressions](#)
- [Using Watchpoints on Different Architectures](#)

TotalView watchpoints are called *modify watchpoints* because TotalView *triggers* a watchpoint only when your program modifies a memory location. If a program writes a value into a location that is the same as that which is already stored, TotalView doesn't trigger the watchpoint because the location's value did not change.

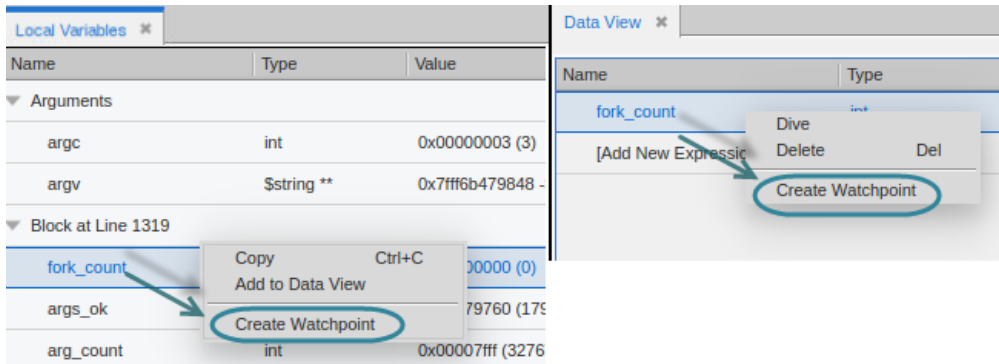
For example, if location 0x10000 has a value of 0, and your program writes a value of 0 to this location, TotalView doesn't trigger the watchpoint, even though your program wrote data to the memory location. See [Triggering Watchpoints](#) for more details on when watchpoints trigger.

NOTE: This discussion describes how to create and modify watchpoints using the UI. To set a watchpoint with the CLI see [Watchpoints](#).

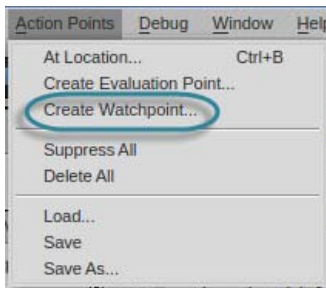
Creating Watchpoints

Create a watchpoint in these ways:

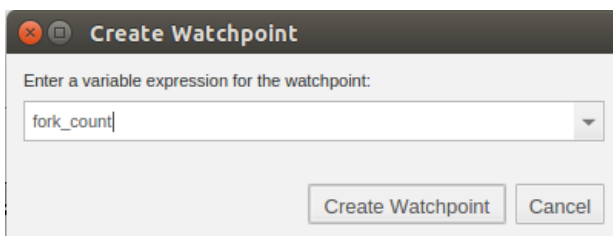
- From the Local Variable view or the Data view by selecting the variable expression, right-clicking to view the context menu, and selecting **Create Watchpoint**.



- From the Action Points menu and then selecting **Create Watchpoint**.



Because TotalView cannot determine where to set the expression when using this option, it displays a dialog box into which you type the variable's name.




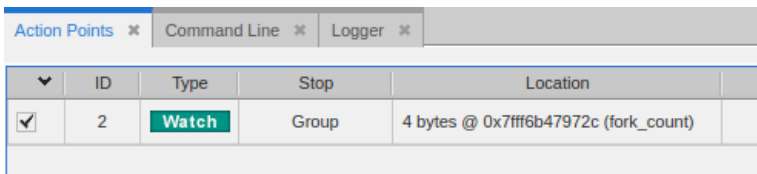
Note: If your platform doesn't support watchpoints, TotalView does not display the option.

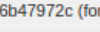
After you enter the name of the variable and click on **Create Watchpoint**, TotalView creates a watchpoint that stops all running threads in all processes in the group when the watchpoint triggers. If you wish to create a watchpoint that stops all running threads in a process or evaluates an expression when the watchpoint triggers, you must modify the watchpoint after you create it. See [Modifying Watchpoints](#) for information on modifying watchpoints.

If you set a watchpoint on a stack variable, TotalView reports that you're trying to set a watchpoint on "non-global" memory. For example, the variable is on the stack or in a block and the variable will no longer exist when the stack is popped or control leaves the block. In either of these cases, it is likely that your program will overwrite the memory, and the watchpoint will no longer be meaningful. See [Watching Memory](#) for more information.

Displaying, Deleting, or Disabling Watchpoints

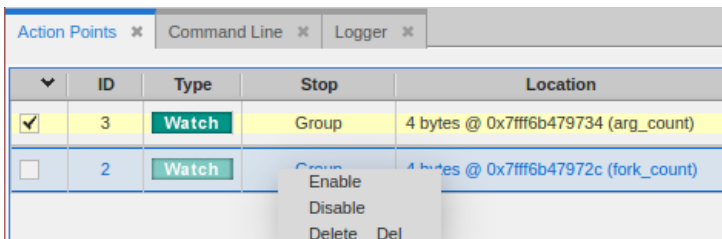
The watchpoint entry, indicated by a Watch () icon displays the action point ID, the amount of memory being watched, and the location watched.

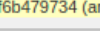
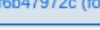


▼	ID	Type	Stop	Location
<input checked="" type="checkbox"/>	2		Group	4 bytes @ 0x7fff6b47972c (fork_count)

NOTE: A watchpoint's width is set to Group by default. See [About an Action Point's Width: Group, Process or Thread](#). See [Modifying Watchpoints](#) for information on changing the width for watchpoints.

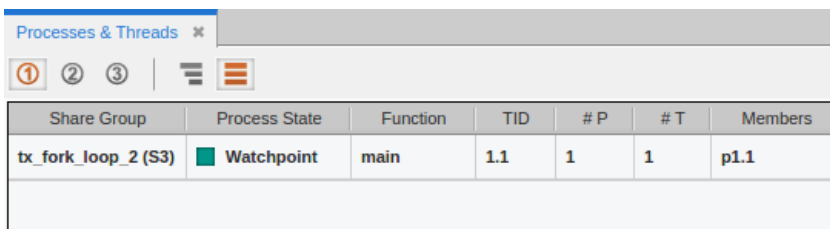
Disable or delete a watchpoint in the Action Points view by right-clicking for a context menu or pressing the **Delete** key. A disabled watchpoint appears grayed out.




▼	ID	Type	Stop	Location
<input checked="" type="checkbox"/>	3		Group	4 bytes @ 0x7fff6b479734 (arg_count)
<input type="checkbox"/>	2		Group	4 bytes @ 0x7fff6b47972c (fork_count)

- Enable
- Disable
- Delete Del

As you step through your program and the watchpoint is triggered, it displays in the Process & Threads view.



Share Group	Process State	Function	TID	# P	# T	Members
tx_fork_loop_2 (S3)	 Watchpoint	main	1.1	1	1	p1.1

Modifying Watchpoints

Modify or delete a watchpoint from the Action Points view by right clicking on the watchpoint to bring up the context menu and selecting **Properties**. This launches the **Modify Watchpoint** dialog.

The screenshot shows a dialog box titled "Modify Watchpoint (on ubuntu1404-x8664)". The main title is "Modify Watchpoint ID: 5" with a checked "Enabled" checkbox. Below this, the "Watch expression" is set to "i". The "Address" field contains "0x7fff50c270fc" and the "Length in Bytes" field contains "4". Under the "When value changes:" section, three radio buttons are present: "Stop Group" (selected), "Stop Process", and "Evaluate Expression". Below these is a text area with the placeholder text "Enter an expression, for example: if (\$newval < 0) \$stopprocess". Further down, there is a field for "Type for \$newval/\$oldval:" with the placeholder "Enter the scalar type for \$newval and \$oldval." and a "Language:" dropdown menu currently set to "C++". At the bottom of the dialog are three buttons: "DELETE", "MODIFY WATCHPOINT", and "CANCEL".

In this dialog you can set the following:

- **Enabled:** If selected, TotalView makes this watchpoint active. (If a watchpoint is inactive, TotalView ignores changes to the watched memory locations.)
- **Address:** The first (or lowest) memory address to watch. Depending on the platform, this address may need to be aligned to a multiple of the **Length in Bytes** field. If you edit the address of an existing watchpoint, TotalView alters the watchpoint so it watches this new memory location and reassigns the watchpoint's action point ID.
- **Length in Bytes:** The number of bytes that TotalView should watch. Normally, this amount is the size of the variable. However, some architectures limit the amount of memory that can be watched. In other cases, you may want TotalView to monitor a few locations in an array. For information on architectural limitations, see [Using Watchpoints on Different Architectures](#).
- **When value changes:** The three width selections control how a breakpoint behaves in a multithreaded or multi-process program. Here's a summary:

- **Stop Group:** Stop all running threads in all processes in the group. When you first create a watchpoint **Stop Group** is selected by default.
- **Stop Process:** Stop all the running threads in the process containing the thread that hit the breakpoint.
- **Evaluate Expression:** Select this to enter a code fragment in the provided field. The expression is compiled into interpreted code that is executed each time the watchpoint triggers. These points can be used to implement countdown and conditional watchpoints.

If you select **Evaluate Expression**, you must specify the following:

- **Expression field:** Enter the code fragment in the expression field.
- **Type for \$newval/\$oldval:** If you are placing the value stored at the memory location into a variable (using **\$newval** and **\$oldval**), you must define the variable's data by using a scalar type, such as int, integer, float, real, or char. You cannot use aggregate types such as arrays and structures.

If the size of the watched location matches the size of the data type entered here, TotalView interprets the **\$oldval** and **\$newval** information as the variable's type. If you are watching an entire array, the watched location can be larger than the size of this type. For more information about setting the type, see [Using Watchpoint Expressions](#).

- **Language:** Indicates the programming language in which you wrote the expression.

Watching Memory

A watchpoint tracks a memory location — it does not track a variable.

This means that a watchpoint might not perform as you would expect when watching stack or automatic variables. For example, suppose that you want to watch a variable in a subroutine. When control exits from the subroutine, the memory allocated on the stack for this subroutine is reassigned. At this time, TotalView is watching memory that is no longer associated with the original stack variable. When the stack memory is reassigned to a new stack frame, TotalView is still watching the same address location. This means that TotalView triggers the watchpoint when something changes this newly assigned memory.

Also, if your program reinvokes a subroutine, it usually executes in a different stack location. TotalView cannot monitor changes to the variable because it is at a different memory location.

All of this means that in most circumstances, you shouldn't place a watchpoint on a stack variable. If you need to watch a stack variable, you will need to create and delete the watchpoint each time your program invokes the subroutine.

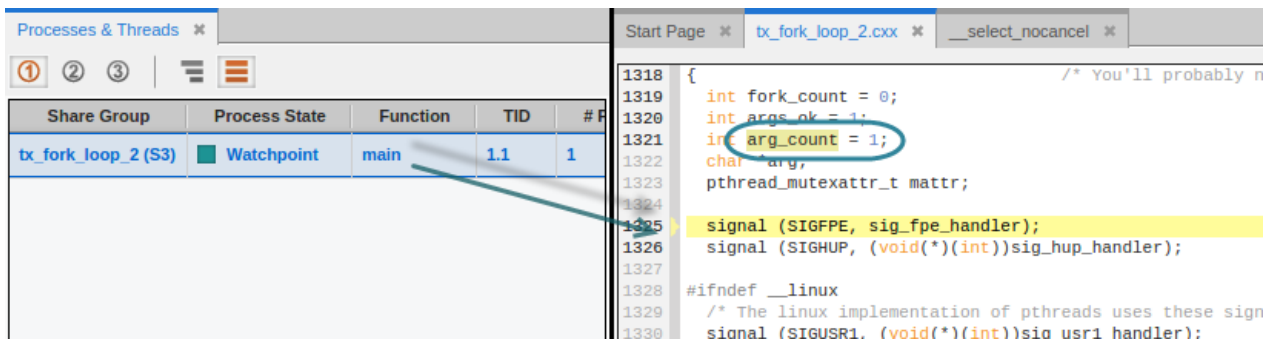
This doesn't mean you can't place a watchpoint on a stack or heap variable. It just means that what happens is undefined after this memory is released. For example, after you enter a routine, you can be assured that memory locations are always tracked accurately until the memory is released.

In some circumstances, a subroutine may be called from the same location. This means that its local variables might be in the same location. In this case, the watchpoint would behave as expected.

If you place a watchpoint on a global or static variable that is always accessed by reference (that is, the value of a variable is always accessed using a pointer to the variable), you can set a watchpoint on it because the memory locations used by the variable are not changing.

Triggering Watchpoints

When a watchpoint triggers, the thread's program counter (PC) points to the instruction *following* the instruction that caused the watchpoint to trigger. For example, this watchpoint on the variable `arg_count` triggered, placing the PC at the next instruction:



If the memory store instruction is the last instruction in a source statement, the PC points to the source line *following* the statement that triggered the watchpoint. (Breakpoints and watchpoints work differently. A breakpoint stops *before* an instruction executes. In contrast, a watchpoint stops *after* an instruction executes.)

Using Multiple Watchpoints

If a program modifies more than one byte with one program instruction or statement, which is normally the case when storing a word, TotalView triggers the watchpoint with the *lowest* memory location in the modified region. Although the program might be modifying locations monitored by other watchpoints, TotalView triggers the watchpoint only for the lowest memory location. This can occur when your watchpoints are monitoring adjacent memory locations and a single store instruction modifies these locations.

For example, suppose that you have two 1-byte watchpoints, one on location 0x10000 and the other on location 0x10001. Also suppose that your program uses a single instruction to store a 2-byte value at locations 0x10000 and 0x10001. If the 2-byte storage operation modifies both bytes, the watchpoint for location 0x10000 triggers. The watchpoint for location 0x10001 does not trigger.

Here's a second example. Suppose that you have a 4-byte integer that uses storage locations 0x10000 through 0x10003, and you set a watchpoint on this integer. If a process modifies location 0x10002, TotalView triggers the watchpoint. Now suppose that you're watching two adjacent 4-byte integers that are stored in locations 0x10000 through 0x10007. If a process writes to locations 0x10003 and 0x10004 (that is, one byte in each), TotalView triggers the watchpoint associated with location 0x10003. The watchpoint associated with location 0x10004 does not trigger.

Performance Impact of Copying Previous Data Values

TotalView keeps an internal copy of data in the watched memory locations for each process that shares the watchpoint. If you create watchpoints that cover a large area of memory or if your program has a large number of processes, you increase TotalView's virtual memory requirements. Furthermore, TotalView refetches data for each memory location whenever it continues the process or thread. This can affect performance.

Using Watchpoint Expressions

If you associate an expression with a watchpoint (by selecting the **Evaluate Expression** button in the **Modify Watchpoint** dialog box entering an expression), TotalView evaluates the expression after the watchpoint triggers. The programming statements that you can use are identical to those used when you create an eval point, except that you can't call functions from a watchpoint expression.

The variables used in watchpoint expressions must be global. This is because the watchpoint can be triggered from any procedure or scope in your program.

NOTE: Fortran does not have global variables. Consequently, you can't directly refer to your program's variables.

TotalView has two variables that are used exclusively with watchpoint expressions:

- **\$oldval:** The value of the memory locations before a change is made.
- **\$newval:** The value of the memory locations after a change is made.

The following is an expression that uses these values:

```
if (iValue != 42 && iValue != 44) {  
iNewValue = $newval; iOldValue = $oldval; $stop;}
```

When the value of the **iValue** global variable is neither 42 nor 44, TotalView stores the new and old memory values in the **iNewValue** and **iOldValue** variables. These variables are defined in the program. (Storing the old and new values is a convenient way of letting you monitor the changes made by your program.)

The following condition triggers a watchpoint when a memory location's value becomes negative:

```
if ($oldval >= 0 && $newval < 0) $stop
```

And here is a condition that triggers a watchpoint when the sign of the value in the memory location changes:

```
if ($newval * $oldval <= 0) $stop
```

Both of these examples require that you set the **Type for \$oldval/\$newval** field in the **Modify Watchpoint** dialog box.

For more information on writing expressions, see [Using Programming Language Elements](#).

If a watchpoint has the same length as the **\$oldval** or **\$newval** data type, the value of these variables is apparent. However, if the data type is shorter than the length of the watch region, TotalView searches for the first changed location in the watched region and uses that location for the **\$oldval** and **\$newval** variables. (It aligns data in the watched region based on the size of the data's type. For example, if the data type is a 4-byte integer and byte 7 in the watched region changes, TotalView uses bytes 4 through 7 of the watchpoint when it assigns values to these variables.)

For example, suppose you're watching an array of 1000 integers called **must_be_positive**, and you want to trigger a watchpoint as soon as one element becomes negative. You declare the type for **\$oldval** and **\$newval** to be **int** and use the following condition:

```
if ($newval < 0) $stop;
```

When your program writes a new value to the array, TotalView triggers the watchpoint, sets the values of **\$oldval** and **\$newval**, and evaluates the expression. When **\$newval** is negative, the **\$stop** statement halts the process.

For descriptions of **\$newval**, **\$oldval**, **\$stop**, and variations on **\$stop**, see [Using Built-in Variables and Statements](#).

This can be a very powerful technique for range-checking all the values your program writes into an array. (Because of byte length restrictions, you can only use this technique on Solaris.)

Using Watchpoints on Different Architectures

Add entry for rockM and CUDA does not support. Built on top of hardware-specific feature. Here it says "watchpoints are not available" add nVIDIA GPUs or CUDA.

The number of watchpoints, and their size and alignment restrictions, differ from platform to platform. This is because TotalView relies on the operating system and its hardware to implement watchpoints.

Watchpoint support depends on the target platform where your application is running, not on the host platform where TotalView is running.

For example, if you are running TotalView on host platform "H" (where watchpoints are not supported), and debugging a program on target platform "T" (where watchpoints are supported), you can create a watchpoint in a process running on "T", but not in a process running on "H".

NOTE: Watchpoints are not available on the Mac OS X platform

The following list describes constraints that exist on each platform:

Computer	Constraints
Linux x86-64 (AMD and Intel)	Watchpoints use the four hardware debugging registers in the x86 processor and also use the ptrace system call to manipulate those registers. You can create up to four watchpoints and each must be 1, 2, 4, or 8 bytes in length, and a memory address must be aligned for the byte length. For example, you must align a 4-byte watchpoint on a 4-byte address boundary.
Linux-PowerLE	On Linux-PowerLE platforms, TotalView uses the Linux kernel's ptrace() PowerPC hardware debug extension to plant watchpoints. The ptrace() interface implements a "hardware breakpoint" abstraction that reflects the capabilities of PowerPC BookE and server processors. If supported at all, the number of watchpoints varies by processor type. Typically, the PowerPC supports at least 1 watchpoint up to 8 bytes long. Systems with the DAWR feature support a watchpoint up to 512 bytes long. The watchpoint triggers if the referenced data address is greater than or equal to the watched address and less than the watched address plus length. Alignment constraints may apply. For example, the watched length may be required to be a power of 2, and the watched address may need to be aligned to that power of 2; that is, $-(\text{address} \% \text{length}) == 0$.
Linux ARM64	TotalView supports watchpoints for ARMv8 processors using the hardware's debug watchpoint registers. You can typically create up to four watchpoints (although some processors may have different limits, allowing from 2 to 16 watchpoints, or none at all). Each must be 1, 2, 4, or 8 bytes in length, and the watched memory address must be aligned for the byte length. Watchpoints cannot overlap.
Mac OSX	Watchpoints are not supported.

Typically, a debugging session doesn't use many watchpoints. In most cases, you are only monitoring one memory location at a time. Consequently, restrictions on the number of values you can watch seldom cause problems.

Barrier Points

A barrier breakpoint is similar to a simple breakpoint, differing only in that it holds processes and threads that reach the barrier point. Other processes and threads continue to run. TotalView holds these processes or threads until all processes or threads defined in the barrier point reach this same place. When the last one reaches a barrier point, TotalView releases all the held processes or threads, but they do not continue executing until you explicitly restart execution. In this way, barrier points let you synchronize your program's execution.

dbarrier

Topics in this section are:

- [About Barrier Breakpoint States](#)
- [Setting a Barrier Breakpoint](#)
- [Creating a Satisfaction Set](#)
- [Hitting a Barrier Point](#)
- [Releasing Processes from Barrier Points](#)
- [Using Barrier Points](#)

RELATED TOPICS

How to hold and release threads and processes

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

About Barrier Breakpoint States

Processes and threads at a barrier point are held or stopped, as follows:

Held

A held process or thread cannot execute until all the processes or threads in its group are at the barrier, or until you manually release it. The various **go** and **step** commands from the **Group**, **Process**, and **Thread** menus cannot start held processes.

Stopped

When all processes in the group reach a barrier point, TotalView automatically releases them. They remain stopped at the barrier point until you tell them to resume executing.

You can manually release held processes and threads with the **Hold** and **Release** CLI commands below. When you manually release a process, the **go** and **step** commands become available again.

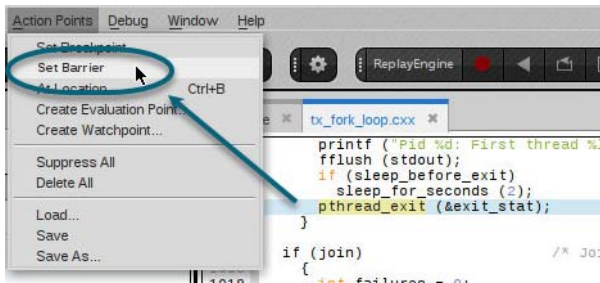
CLI: dfocus ...dhold
dfocus ...dunhold

You can reuse the **Hold** command to again toggle the hold state of the process or thread.

When a process or thread is held, TotalView displays **Stopped** next to the relevant process or thread in the Process State column in the Processes & Threads view.

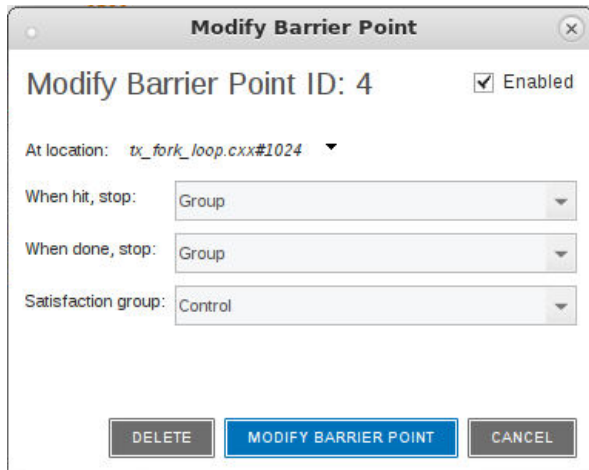
Setting a Barrier Breakpoint

Set a barrier breakpoint in the same way as a regular breakpoint: by either selecting the line in the Source pane and selecting **Action Points > Set Barrier**, or by right-clicking on the line and choosing **Set Barrier** from the context menu.



Once the barrier point is set, customize its properties by right-clicking on the barrier point in either the Source view or the Action Points tab, and choosing Properties to open the **Modify Barrier Point** dialog:

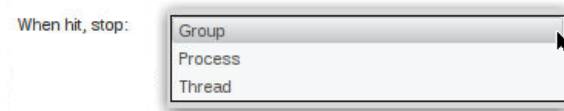
Figure 43, Modify Barrier Point Property dialog



You most often use barrier points to synchronize a set of threads. 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 reaching the barrier from responding to resume commands (for example, **step**, **next**, or **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 all of the threads being held there. *They are just released; they do not continue.* That is, they are left stopped at the barrier. If you continue the process, those threads also run.

If you stop a process and then continue it, the held threads, including those waiting at an unsatisfied barrier, do not run. Only unheld threads run.

The When hit, stop option:



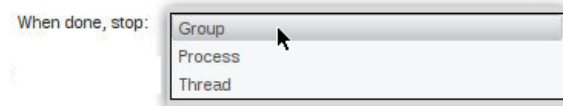
The **When hit, Stop** drop-down menu sets which other threads TotalView stops when execution reaches the barrier point, as follows:

Scope	Stopped Threads
Group	Stops all threads in the current thread's control group.
Process	Stops all threads in the current thread's process.
Thread	Stops only this thread.

CLI: `dbarrier -stop_when_hit`

After all processes or threads reach the barrier, TotalView releases all held threads. **Released** means that these threads and processes can now run.

The When done, stop option



The **When Done, Stop** drop-down menu defines what else it should stop, as follows:

Scope	Stopped Threads
Group	Stops all threads in the current thread's control group.
Process	Stops all threads in the current thread's process.
Thread	Stops only this thread.

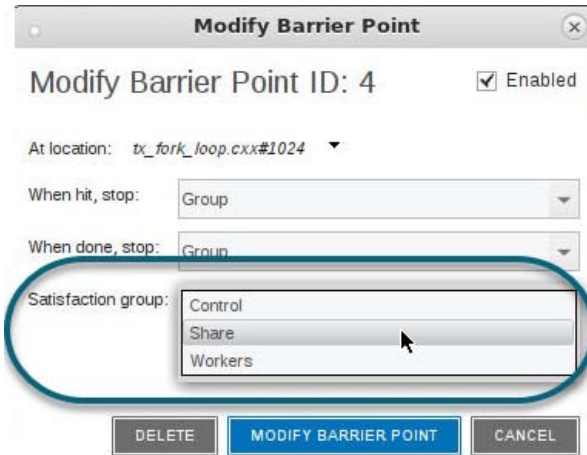
dbarrier-stop_when_done

Creating a Satisfaction Set

For more control over what processes or threads are stopped, use a *satisfaction set*. This setting defines which processes or threads must be held before TotalView can release the group. That is, the barrier is *satisfied* when TotalView has held all of the indicated processes or threads.

From the Modify Barrier Point dialog (accessed by right-clicking on the barrier point and choosing Properties), choose an option from the Satisfaction group drop-down:

Figure 44, Satisfaction Set properties



Scope	Held Processes or Threads
Control	The default. Holds all processes.
Share	Holds all the processes that share the same image as the current executable where the barrier point is set.
Workers	Holds only this thread.

Control and **Share** settings hold at the process level. For multithreaded programs, to hold the threads at the barrier point, use the **Workers** setting, which holds at the thread level.

In TotalView, the *workers group* are the threads in the **main()** routine, which is added to the Share group. For more information on worker threads and how TotalView creates groups, see the section “How TotalView Creates Groups” in the chapter “About Groups, Processes and Threads” in the *Classic TotalView User Guide*.

When you set a barrier point, TotalView places it in every process in the share group.

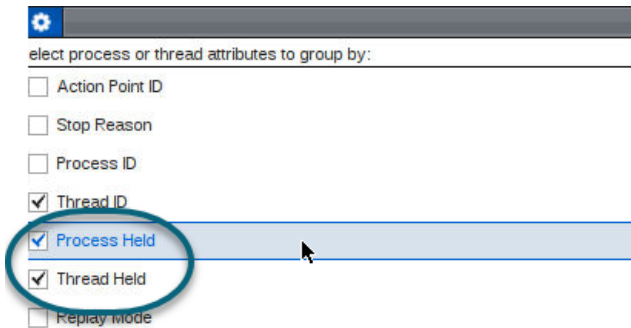
Hitting a Barrier Point

If you run one of the processes or threads in a group and it hits a barrier point, the Process and Threads View displays **Stopped** in the Process State column for that process’s or thread’s entry.

CLI: `dstatus`

If you create a barrier and all the process’s threads are already at that location, TotalView won’t hold any of them. However, if you create a barrier and all of the processes and threads are not at that location, TotalView holds any thread that is already there.

The Processes & Threads view displays which threads or processes are being held when you select “Process Held” or “Thread Held” in the “group by” dialog:



Releasing Processes from Barrier Points

TotalView automatically releases processes and threads from a barrier point when they hit that barrier point and all other processes or threads in the group are already held at it.

Changing Settings and Disabling a Barrier Point

Setting a barrier point at the current PC for a **stopped** process or thread holds the process there. If, however, all other processes or threads affected by the barrier point are at the same PC, TotalView doesn't hold them. Instead, TotalView treats the barrier point as if it were an ordinary breakpoint.

TotalView releases all processes and threads that are held and which have threads at the barrier point when you disable the barrier point (by right-clicking on it in the Action Points tab and selecting **Disable**).

```
CLI: ddisable
```

Using Barrier Points

Because threads and processes are often executing different instructions, keeping threads and processes together is difficult. The best strategy is to define places where the program can run freely and places where you need control. This is where barrier points come in.

To keep things simple, this section only discusses multi-process programs. You can do the same types of operations when debugging multithreaded programs.

Why breakpoints don't work (part 1)

If you set a breakpoint that stops all processes when it is hit and you let your processes run using the **Group > Go** command, you might get lucky and have all of your threads reach the breakpoint together. More likely, though, some processes won't have reached the breakpoint and TotalView will stop them wherever they happen to be. To get your processes synchronized, you would need to find out which ones didn't get there and then individually get them to the breakpoint using the **Process > Go** command. You can't use the **Group > Go** command since this also restarts the processes stopped at the breakpoint.

Why breakpoints don't work (part 2)

If you set the breakpoint's property so that only the process hitting the breakpoint stops, you have a better chance of getting all your processes there. However, you must be careful not to have any other breakpoints between where the program is currently at and the target breakpoint. If processes hit these other breakpoints, you are once again left to run processes individually to the breakpoint.

Why single stepping doesn't work

Single stepping is just too tedious if you have a long way to go to get to your synchronization point, and stepping just won't work if your processes don't execute exactly the same code.

Why barrier points work

If you use a barrier point, you can use the **Group > Go** command as many times as it takes to get all of your processes to the barrier, and you won't have to worry about a process running past the barrier. The Root Window shows you which processes have hit the barrier, grouping all held processes under **Breakpoint** in the first column.

Barrier Point Illustration

Creating a barrier point tells TotalView to hold a process when it reaches the barrier. Other processes that can reach the barrier but aren't yet at it continue executing. One-by-one, processes reach the barrier and, when they do, TotalView holds them.

When a process is held, it ignores commands that tell it to execute. This means, for example, that you can't tell it to go or to step. If, for some reason, you want the process to execute, you can manually release it using the **dunhold** command.

```
CLI: dfocus p dunhold -process
```

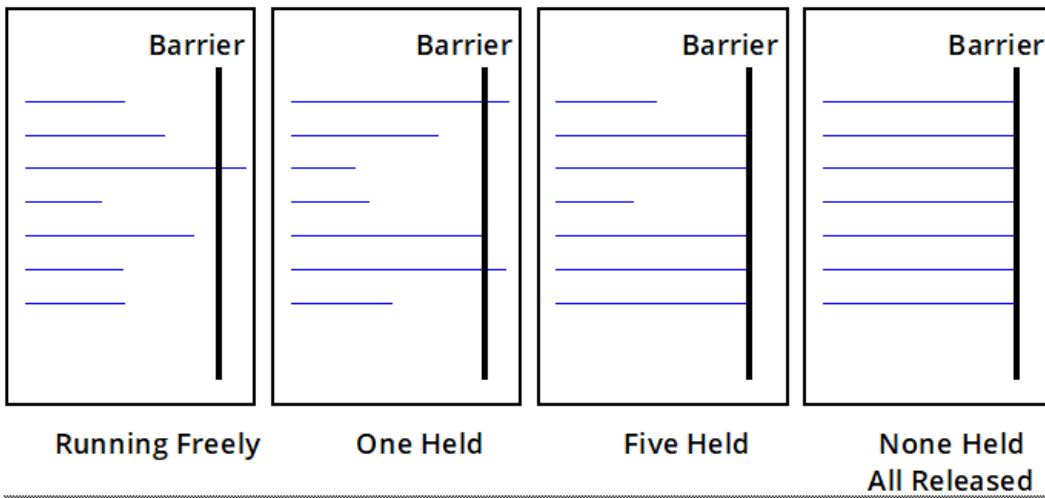
When all processes that share a barrier reach it, TotalView changes their state from held to released, which means they no longer ignore a command that tells them to begin executing.

The following figure shows seven processes that are sharing the same barrier. (Processes that aren't affected by the barrier aren't shown.)

- First block: All seven processes are running freely.

- Second block: One process hits the barrier and is held. Six processes are executing.
- Third block: Five of the processes have now hit the barrier and are being held. Two are executing.
- Fourth block: All processes have hit the barrier. Because TotalView isn't waiting for anything else to reach the barrier, it changes the processes' states to released. Although the processes are released, none are executing.

Figure 45, Running to Barriers



For more information on barriers, see [Barrier Points](#).

RELATED TOPICS

dhold

dhold in "CLI Commands" in the *TotalView Reference Guide*

dunhold

dunhold in "CLI Commands" in the *TotalView Reference Guide*

Controlling an Action Point's Width

You can control an action point's scope, or *width*, i.e. whether it stops a group of processes, a single process (which includes all its threads), or a single thread. For example, in a multithreaded program, you might not want to stop other threads when a thread hits a breakpoint.

About an Action Point's Width: Group, Process or Thread

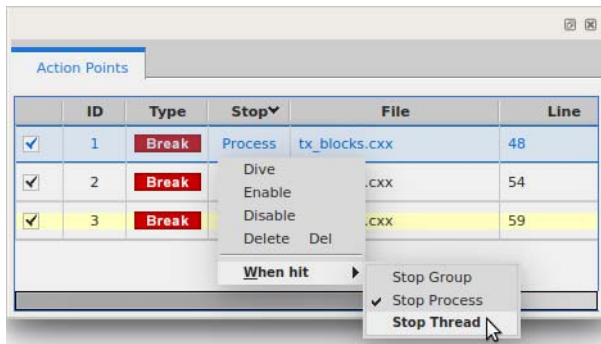
For a single-process, single-threaded program, an action point's width is irrelevant: when the thread hits the breakpoint, it stops. For a multi-process, multithreaded program, it is useful to finely control what to stop. You can stop all the threads in a group, all the threads in a process, or just that single thread.

- **Group:** All the processes a program creates are placed into a *control group*.
When an action point is set to **Stop Group**, and a thread reaches the breakpoint, all running threads in all processes in the group stop.
- **Process:** A process can contain any number of threads.
The default setting for action points is **Stop Process**, which stops all the running threads in the process containing the thread that hit the breakpoint. This is useful in a multi-process program in which you might want to let the other processes continue running.
- **Thread:** A thread is a single unit of execution created by your program.
When an action point is set to **Stop Thread**, the thread that first executes to this breakpoint stops. The other threads retain their current states, running or stopped.

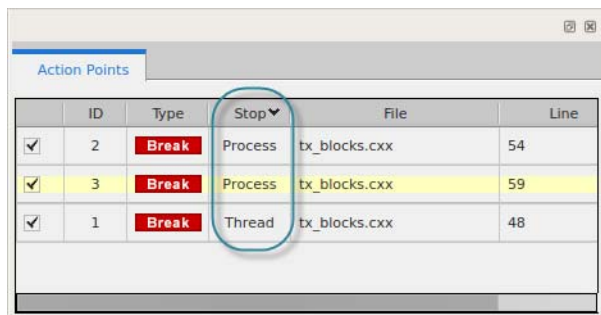
Setting the Action Point's Width

Select **When hit** from the context menu, and choose **Stop Group**, **Stop Process**, or **Stop Thread**.

Figure 46, Action Points width context menu




The **Stop column** displays the selected width:



Note: The “When hit” context menu is only available for breakpoints.

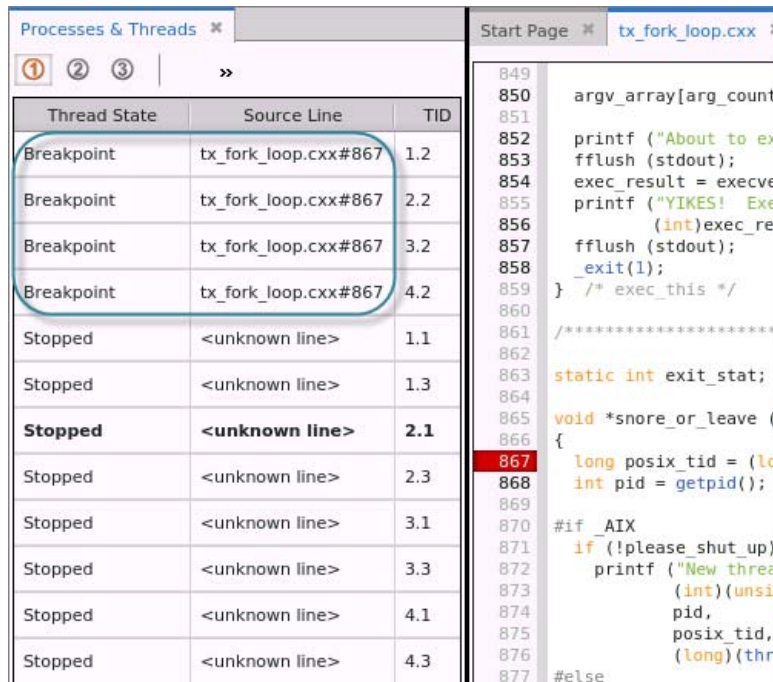
Action Point Width and Process/Thread State

To see how this works, let’s add some breakpoints of varying widths. The program here has four processes, each with three threads.

In the Processes & Threads view, we’ll group by Thread State and Source Line and display the List View ().

- **Using the default Process Width:**
 - Set a breakpoint with the default width of Process.

- Select **Go** from the debug toolbar.

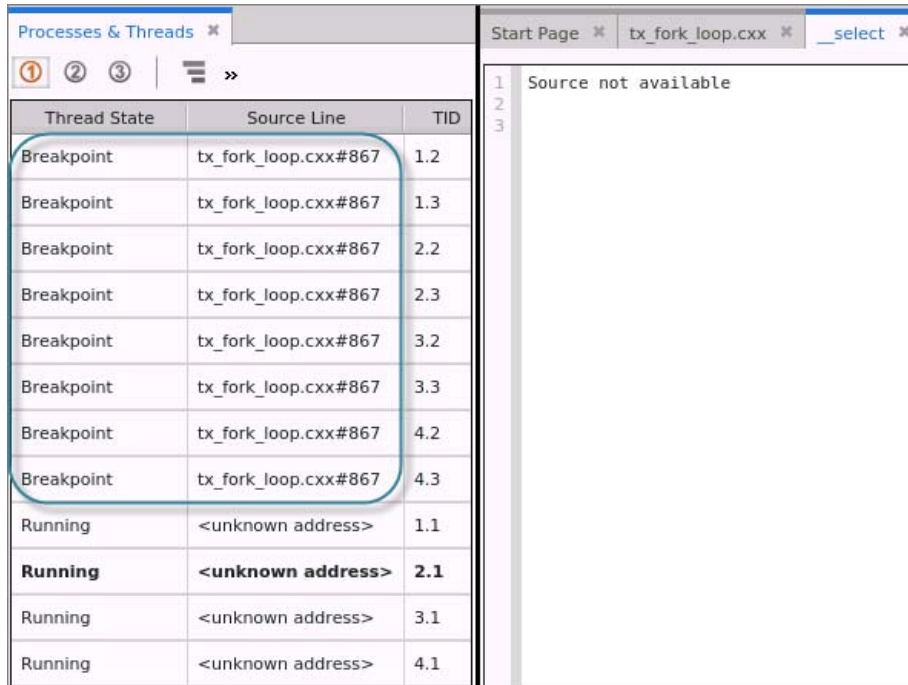


Each of the four processes had a thread (#2) that hit the breakpoint (although it's also possible that another thread in a process would have had time to hit the breakpoint before it was stopped).

The first thread that hit the breakpoint stopped all other threads at whatever point they had reached.

■ **Using Thread Width:**

We'll change the breakpoint's width to Thread, and run the program again.



Each of the eight threads that could encounter the breakpoint did, and are stopped. The other four continue to run, i.e. the threads hitting the breakpoint do not stop other threads.

■ Using Group Width:

We'll change the breakpoint's width to Group and run the program again. When a thread hits the breakpoint, all threads in all processes in the group stop.

Thread State	Function	TID
Breakpoint	snore_or_leave	4.3
Stopped	__select	1.1
Stopped	__select	1.2
Stopped	__select	2.1
Stopped	__select	2.2
Stopped	__select	3.1
Stopped	__select	3.2
Stopped	__select	4.1
Stopped	__select	4.2
Stopped	snore_or_leave	1.3
Stopped	snore_or_leave	2.3
Stopped	snore_or_leave	3.3

Selecting **Go** again resumes all the threads until one hits the breakpoint, at which point, all other threads are again stopped. We can continue to run the program until all eight threads have passed the breakpoint. With no more threads to hit the breakpoint, all the threads are left running.

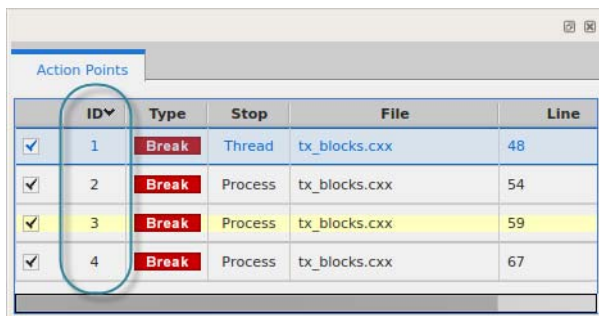
Managing and Diving on Action Points

The **Action Points view** contains the following columns:

- **Enable toggle:** Checkbox that enables or disables the breakpoint.
- **ID:** The Action Point ID. Use this ID when you need to refer to the action point. Like process and thread identifiers, action point identifiers are assigned numbers as they are created. The ID of the first action point created is 1; the second ID is 2, and so on. These numbers are never reused during a debugging session.
- **Type:** The type of Action Point, in this case breakpoint.
- **Stop:** The Action Point's width, i.e. its scope or what it should stop: all threads in the process, all threads in all processes in the group, or just the first thread to reach the breakpoint. The default is Process.
- **File:** The source file in which the breakpoint is set.
- **Line:** The line at which the breakpoint is set in the source file.

Sorting

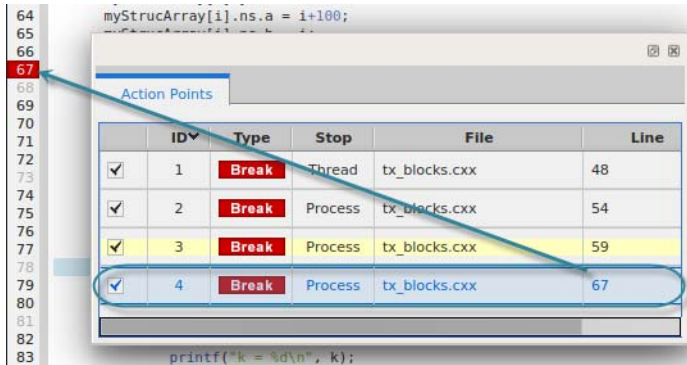
You can sort any column by clicking the column header. A sorted column displays a down or up arrow to indicate its sorting order:



	ID▼	Type	Stop	File	Line
<input checked="" type="checkbox"/>	1	Break	Thread	tx_blocks.cxx	48
<input checked="" type="checkbox"/>	2	Break	Process	tx_blocks.cxx	54
<input checked="" type="checkbox"/>	3	Break	Process	tx_blocks.cxx	59
<input checked="" type="checkbox"/>	4	Break	Process	tx_blocks.cxx	67

Diving

To *dive* on an Action Point, double-click it in the Action Points view. This displays its location in the source file in the Source view.



NOTE: You cannot dive on a watchpoint since it does not point to a location in source but rather is directly mapped to a memory location.

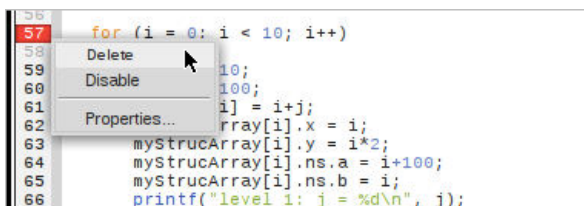
Deleting, Disabling, and Suppressing

You can either completely delete or just disable an Action Point if you think you may use it later.

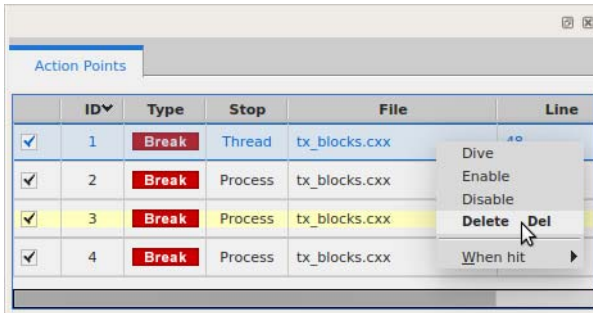
■ Deleting an Action Point

Delete one or more Action Points in several ways:

- **In the Source view**, launch the context menu by right-clicking on the Action Point's line number and selecting **Delete**.



- In the **Action Points** menu, launch the context menu by right-clicking on the Action Point and selecting **Delete**.



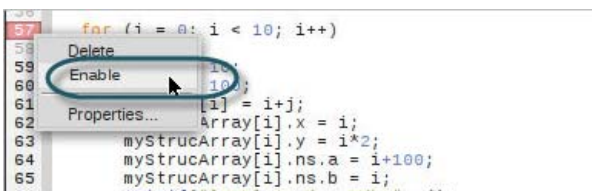
- Clicking on the Action Point in the Source view. This toggles an Action Point on or off.
- Using the **Delete key** on your keyboard.

NOTE: You can select multiple Action Points and use any of these methods to delete several at once. The **Action Points > Delete All** main menu item deletes all breakpoints.

■ Disabling and Enabling an Action Point

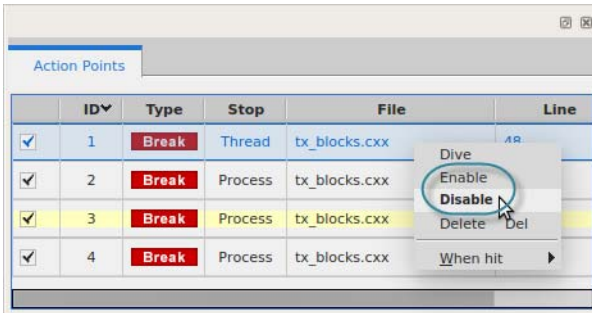
A disabled Action Point displays as a dimmed icon.

- In the **Source view**, access the context menu by right-clicking on the Action Point and selecting **Disable** or **Enable**:

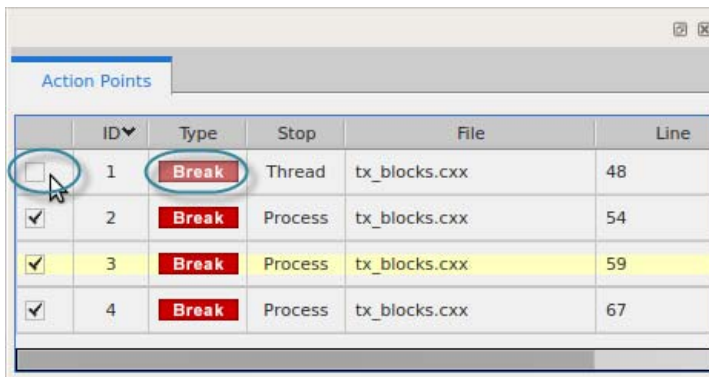


- In the **Action Points** menu:

Access the context menu by right-clicking on the Action Point and selecting **Disable** or **Enable**:

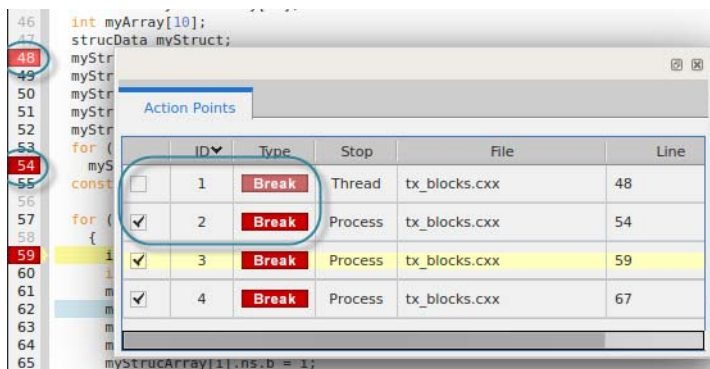


Use the Enable/Disable checkbox toggle:



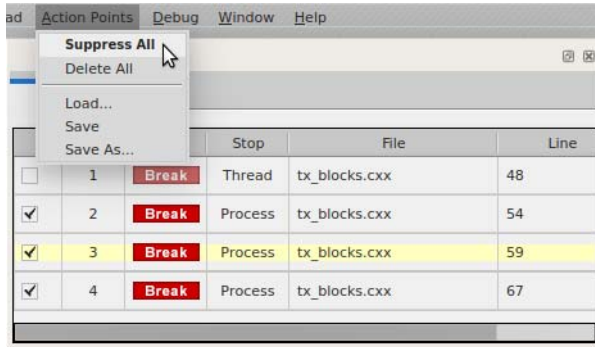
NOTE: Tip: You can use disabled breakpoints to flag places in your code for quick navigation. Create a breakpoint, disable it, and then dive on it to display that line of code in the Source view.

Both the Source view and the Action Points view display the state of an Action Point:



■ Suppressing Action Points

The main **Action Points** menu has a toggle menu item **Suppress All**.

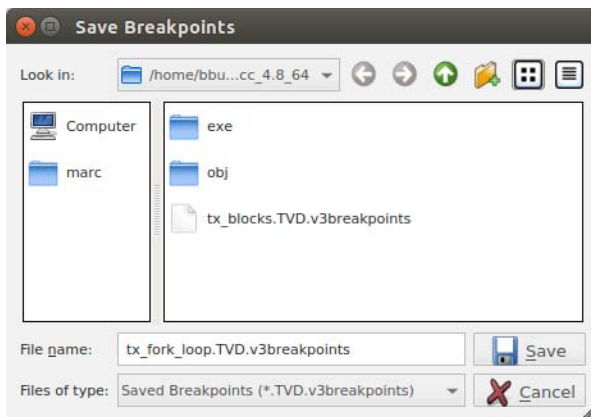


Toggling this item on, as shown, effectively disables all existing action points. If the code is run, threads will not stop at any action points. Although you can create new action points (and delete existing ones), the new action points too will be effectively disabled. Toggling this item off restores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

Saving and Loading Action Points

Both the main **Action Points** menu, and the action points context menu, have **Save**, **Save As**, and **Load** menu items.

- **Save** saves the action points and their state to a file with the default name *program_name.TVD.v4breakpoints*, where *program_name* is the name of your program. The file is created if need be, and an existing file of the same name is overwritten. Use **Save As** to save under a different name.
- **Save As** and **Load** open a dialog:



This dialog lets you choose the directory and file from which to load or save the action points.

You can save and load action point files with **Suppress All** turned on. Actions points are saved with the state they would have when unsuppressed. Loaded action points are suppressed, but regain their saved state when **Suppress All** is turned off.




More on Action Points Using the CLI


The CLI provides access to all action point features using the command line view.

Creating Barrier Points, Evalpoints, and Watchpoints






- A *breakpoint* stops execution of processes and threads that reach it. See [Breakpoints](#).
- An *evalpoint* executes a code fragment when it is reached. See [Evalpoints](#).
- A *watchpoint* monitors a location in memory and stops execution when a provided condition is met. See [Watchpoints](#).
- A *barrier point* synchronizes a set of threads or processes at a location. See [Barrier Points](#).

Identifying Icons for All Types of Action Points

In the Action Points view, evalpoints, barrier points and watchpoints created in the CLI display as , , and  icons.

In the Source view, evalpoints and barrier points show up as line numbers with the same colors, for example,  identifies an evalpoint. Watchpoints do not appear in the Source view because they represent memory locations, not lines in the code.

In the Action Points view, these action points look like this:

	ID	Type	Stop	Location	
<input checked="" type="checkbox"/>	9		Group	tx_fork_loop_2.cxx	244
<input checked="" type="checkbox"/>	1		Group	4 bytes @ 0x7fffd5c5dbc (fork_count)	
<input checked="" type="checkbox"/>	3		Process	tx_fork_loop_2.cxx	641
<input checked="" type="checkbox"/>	8		Process	tx_fork_loop_2.cxx	237
<input checked="" type="checkbox"/>	2		Process	4 bytes @ 0x7fffd5c5dc4 (arg_count)	

Breakpoints

The CLI provides additional parameters for breakpoints, some of which are not yet supported in the UI. For example, you can provide a specific function or address on which to set a breakpoint:

```
CLI: dbreak breakpoint-expr
```

```
CLI: dbreak -address addr
```

- **Set a breakpoint directly on a function:**

```
dbreak my_function
```

Sets a breakpoint on the function `my_function`. If multiple files include this function, this call sets a breakpoint in all the files.

Note: For breakpoints on functions of the same name that span multiple files, the UI currently displays each breakpoint with the same ID. Deleting one deletes all breakpoints that match that ID.

- **Set a breakpoint directly on an address:**

```
dbreak -address 0x2b0b7aad1470
```

Sets a breakpoint at a specific address, useful to observe a specific location in memory.

Evalpoints

A breakpoint with associated code is an evalpoint. When your program reaches an evalpoint, TotalView executes the code. Evalpoints can contain print statements, commonly used in debugging, but, unlike print statements, don't require that you recompile your program. They also let you patch your programs and route around code that you want replaced—for example, to branch around code that you don't want your program to execute or to add new statements.

```
CLI: dbreak line-number -e expr
```

For example:

- **Print the value of result:**

```
dbreak 597 -e { printf("The value of result is %d\n", result) };
```

Note that this breakpoint does not stop execution. Evalpoints do exactly what you tell them to do; you don't have to stop program execution just to observe print statement output.

- **Skip some code:**

```
dbreak 50 -e {goto 63};
```

Any thread that reaches this breakpoint transfers to line 63.

- **Stop a loop after a certain number of iterations:**

```
dbreak 597 -e { if ( i % 100) == 0)
{
printf("The value of i is %d\n", i);
```

```
$stop;  
}  
;
```

Uses programming language statements and a built-in debugger function to stop a loop every 100 iterations. It also prints the value of **i**.

```
dbreak 597 -e { $count 100 };
```

In contrast, this evalpoint just stops the program every 100 times a statement is executed.

Watchpoints

To create a watchpoint expression with the CLI, use the **-e** argument on the **dwatch** command, and enter an expression. The expression is compiled into interpreted code that is executed each time the watchpoint triggers.

```
CLI: dwatch -e expr
```

TotalView has two variables used exclusively with watchpoint expressions:

- **\$oldval**: The value of the memory locations before a change is made.
- **\$newval**: The value of the memory locations after a change is made.

NOTE: Watchpoints are not available on the Mac OS X platform.

For example:

```
{if ($newval > 2) $stop}
```

This watchpoint triggers when the updated value of the variable **arg_count** equals more than two.

At this point, you can see in the Local Variables view that *arg_count* equals 1:

The screenshot shows two panels from a debugger. The top panel, 'Local Variables', displays a table of variables:

Name	Type	Value
Arguments		
argc	int	0x00000003 (3)
argv	\$string **	0x7ffdd5c5ed8 -> 0x7ffdd5c81a2 -> "/home/bburns/sandl
Block at Line 1319		
fork_count	int	0x00000000 (0)
args_ok	int	0x00000001 (1)
arg_count	int	0x00000001 (1)
arg	\$string *	0x00000000

The bottom panel, 'Command Line', shows the following command:

```
Linux x86_64 TotalView 2017X.0.8
Thread 1:1 has exited
d1.<: dwatch arg_count -e {if ($newval > 2) $stop}
```

The watchpoint is now reflected in the Action Points view with the watchpoint **Watch** icon:

The screenshot shows the 'Action Points' view with the following table:

ID	Type	Stop	Location
4	Watch	Group	4 bytes @ 0x7ff294fcde4 (arg_count)
6	Watch	Process	4 bytes @ 0x7ff294fcdec (fork_count)

When you advance the program by choosing **Go**, **Next** or **Step**, for instance, TotalView stops at this watchpoint when the value of `arg_count` has been incremented past 2.

The screenshot displays the TotalView IDE interface with several panels:

- Processes & Threads:** Shows a process named `tx_fork_loop_2 (S3)` with a green 'Watchpoint' icon next to it.
- Source:** Shows C++ code with line 1388 highlighted in yellow: `if (args_ok && argc > arg_count)`.
- Local Variables:** A table showing the current state of variables:

Name	Type	Value
fork_count	int	0x00000002 (2)
args_ok	int	0x00000001 (1)
arg_count	int	0x00000003 (3)
arg	\$string *	0x7ffffd5c822c -> "2"
mtx	nthread_mutex	(nthread_mutexattr_t)
- Action Points:** A table listing watchpoints:

ID	Type	Stop	Location
1	Watch	Group	4 bytes @ 0x7ffffd5c5dbc (fork_count)
2	Watch	Process	4 bytes @ 0x7ffffd5c5dc4 (arg_count)
- Command Line:** Shows the debugger command `d1.<> dwatch arg_count -e {if ($newval > 2) $stop}` and the output `Thread 1.1 hit watchpoint 2 (4 bytes @ 0x7ffffd5c5dc4)`.

- The **Processes & Threads view** reports the Process State stopped at this watchpoint.
- The **Source view** identifies the location of the PC.
- The **Local Variables view** reports that `arg_count`'s value is at 3.
- The **Command Line view** reports that watchpoint 2 has been hit.
- The **Action Point view** displays a pale yellow background to identify the stopped watchpoint.

RELATED TOPICS

Breakpoints	dbreak
Evalpoints	dbreak
Barrier points	dbarrier
Watchpoints	dwatch

Barrier Points

In a multithreaded, multi-process program, threads and processes are often executing different instructions, so cannot be easily synchronized to all stop at the same breakpoint. Use a **barrier breakpoint** to hold processes and threads that reach the barrier point until all have reached it. With a barrier point, you can use the **Group > Go** command as many times as it takes to get all of your processes to the barrier, and you won't have to worry about a process running past it.

The Processes and Threads pane displays which processes have hit the barrier, grouping all held processes under Breakpoint in the first column.

```
CLI: dbarrier line-number
```

You can fine-tune how a barrier works to define additional elements to stop when a barrier point is satisfied or a thread encounters a barrier point. You can also incorporate an expression into a barrier point, similar to an evalpoint.

To insert a default barrier point:

```
dbarrier 123
```

This barrier stops each thread in all processes in the control group when it arrives at line 123. After all processes arrive, the barrier is satisfied, and TotalView releases all processes.

Saving Action Points to a File Using the CLI

You can save a program's action points to a file. TotalView then uses this information to reset these points when you restart the program. When you save action points, TotalView creates a file named *program_name.TVD.v4breakpoints*, where *program_name* is the name of your program.

```
CLI: dactions -save filename
```

Start TotalView with the **-sb** option (see "TotalView Command Syntax") to automatically save your breakpoints.

```
CLI: dsetTV::auto_save_breakpoints
```

At any time, you can restore saved action points.

```
CLI: dactions -load filename
```

RELATED TOPICS

The **TV::auto_save_breakpoints** variable **TV::auto_save_breakpoints** in "TotalView Variables"

The **TV::auto_load_breakpoints** variable **TV::auto_load_breakpoints** in "TotalView Variables"

Suppressing and Unsuppressing Action Points

Action points can be suppressed and unsuppressed as described in [Deleting, Disabling, and Suppressing](#). The CLI commands for this are:

```
CLI: dactions -suppress
```

```
CLI: dactions -unsuppress
```

Examining and Editing Data

- Viewing Data in TotalView
- About Expressions
- The Call Stack, Local Variables, and Registers Views
- The Data View
- The Array View
- Using the CLI to Examine Data

Viewing Data in TotalView

TotalView is rich with features to analyze your program's data.

The Call Stack, Local Variables View, Data View, and Register View

The Call Stack, the Local Variables, Data View, and Registers view all work together to provide views of your data at different points of your running program.

The Local Variables view, Registers view, and the Data View work in concert to display your program's data in detail.

The Local Variables view displays blocks of variables local to the selected call stack frame. When you move through the backtrace, change the thread of focus or the PC changes, the local variables in the Local Variables view update. The same is true of the Register view which displays the value of general purpose registers and floating point registers local to a call stack frame.

The Data View enables you to create expressions in order to analyze your data. Add new variables to the Data View by either entering the variable name in the **Add New Expression** field in the Data View, right clicking on the variable and selecting **Add to Data View** or **Add to New Data View** from the context menu, or by simply dragging the variable name from the Local Variables view to the Data View. Both views update variable values as your program runs.

The Data View automatically transforms and aggregates your data so that it displays in a way that makes it easy to examine. If you are using Standard Template Library (STL) types, this is especially useful and is analogous to the customized STLView in Classic TotalView.

Edit a wide range of data while debugging your programs, such as variable type and value. If a variable is complex, dive on it to get more detailed information. See [Diving on Variables](#).

About Expressions

Either directly or indirectly, accessing and manipulating data requires an evaluation system. When your program (and TotalView, of course) accesses data, it must determine where this data resides. The simplest data lookups involve two operations: looking up an address in your program's symbol table and interpreting the information located at this address based on a variable's datatype. For simple variables such as an integer or a floating-point number, this is straightforward.

Looking up array data is slightly more complicated. For example, if the program wants **my_var[9]**, it looks up the array's starting address, then applies an offset to locate the array's 10th element. In this case, if each array element uses 32 bits, **my_var[9]** is located 9 times 32 bits away.

In a similar fashion, your program obtains information about variables stored in structures and arrays of structures.

Structures complicate matters slightly. For example **ptr->my_var** requires three operations: extract the data contained within address of the **my_var** variable, use this information to access the data at the address being pointed to, then display the data according to the variable's datatype.

Accessing an array element such as **my_var[9]** where the array index is an integer constant is rare in most programs. In most cases, your program uses variables or expressions as array indices; for example, **my_var[ctr]** or **my_var[ctr+3]**. In the latter case, TotalView must determine the value of **ctr+3** before it can access an array element.

Here is an illustration showing TotalView accessing the **my_var array** in the three ways discussed in this section:

Figure 47, Data View: Accessing Array Elements

Name	Type	Value
myArray	int...	(int[10])
[0]	int	0x0000000a (10)
[1]	int	0x0000000b (11)
[2]	int	0x0000000c (12)
[3]	int	0x0000000d (13)
[4]	int	0x0000000e (14)
[5]	int	0x00000001 (1)
[6]	int	0x0000082b (2091)
[7]	int	0x00000001 (1)
[8]	int	0x00000000 (0)
[9]	int	0x00000000 (0)
myArray[3]	int	0x0000000d (13)
i	int	0x00000004 (4)
myArray[i-1]	int	0x0000000d (13)
[Add New Expression]		

Using C++

The TotalView expression system is able to interpret the way you define your classes and their inheritance hierarchy. For example, assume that you have the following declarations:

```
class Cylinder : public Shape { public:
...
};
```

Figure 48 shows the second expression of cylinder cast to the type `struct Shape`, and TotalView properly evaluating the expression as the new type to show `struct Shape`'s data members.

Figure 48, Class Casting

Name	Type	Value
▼ cylinder	struct Cylinder	(struct Cylinder)
▶ Circle	struct Circle	(struct Circle)
m_height	double	50.5
▼ cylinder	struct Shape	(struct Shape)
▶ \$vtable	int(...)**	0x00401730 -> 0x00401464 : Shape::area(void) const
m_area	double	32046.64552
m_volume	double	809177.79938
[Add New Expression]		

The Call Stack, Local Variables, and Registers Views

The **Call Stack**, **Local Variables** and **Registers** views work together to display the stack and all the arguments, local variables, and registers associated with the selected frame.

Figure 49, Call Stack, Local Variables, and Registers Views

The screenshot displays three debugger views stacked vertically:

- Local Variables:** A table with columns 'Name', 'Type', and 'Value'.

Name	Type	Value
Arguments		
argc	int	0x00000001 (1)
argv	\$string **	0x7fff95c69068 -> 0x7fff95c6a0b3 -> "/etc/...
strucArray	struct junk[20]	(struct junk[20])
- Call Stack:** A list of function frames. The top frame is 'main' (C++), which is bolded. Below it are '_libc_start_main' and '_start'.
- Registers:** A table with columns 'Name' and 'Value'.

Name	Value
%fs	0x00000000 (0)
%gs	0x00000000 (0)
%eflags	0x00000297 (IOPL=0,CF+PF+AF+SF+IF)
%rip	0x004005ac (main+0x40)


The Call Stack View

The **Call Stack View** shows the backtrace of the thread that is currently in focus and stopped, i.e., the Thread of Interest or **TOI**, bolded in the Process and Thread View:

To view the backtrace from a different thread, make a new thread selection in the Process and Thread View.

Note that the Call Stack displays the language and name of the program or function in

focus: 

The **Information tab**  on the Call Stack displays additional detail about the location of the stopped thread and the selected frame.

The info panel displays which function the selected stack frame is in, the source file containing that function, the line number where the PC is, and the Frame Pointer (FP) for the selected frame.

The Local Variables View

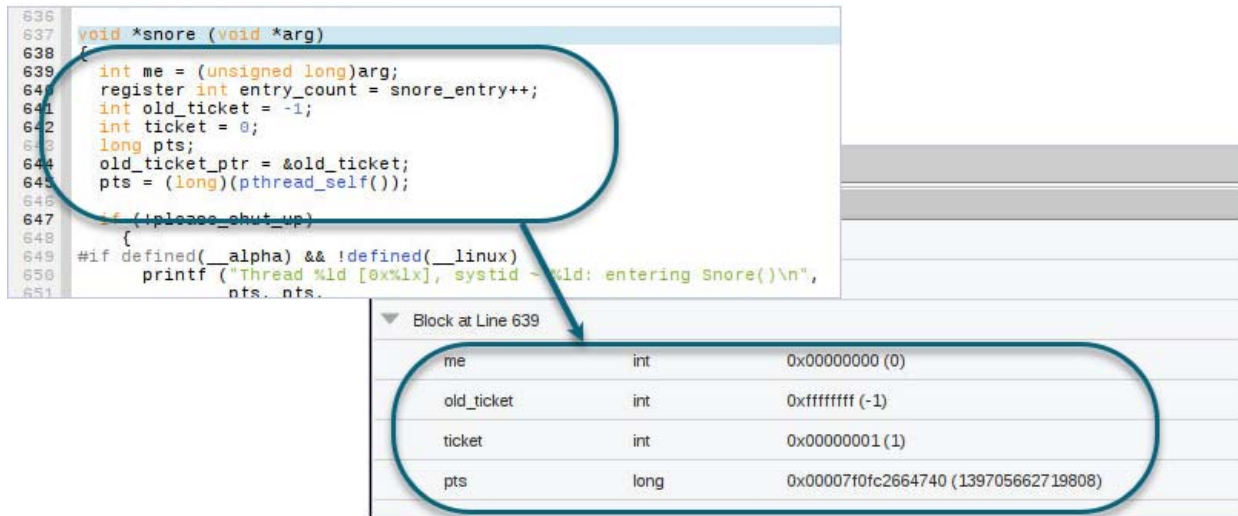
The **Local Variables** view displays all the arguments and local variables associated with the selected frame in the Call Stack.

NOTE: Global variables are not displayed in the Local Variables view. To view them, add them to the Data View. See [Entering a New Expression into the Data View](#).

The **Local Variable's** default three columns display each variable's **Name**, **Type** and **Value** at the time that the thread stopped. (You can customize the visible columns in the view, turning on Thread ID and Address, similar to the Data View. See [Customizing the Data View](#) for more information.)

For example, in [Figure 50](#), for selected function `snore`, the local variables display under the associated scope or *program block*.

Figure 50, Local Variables display under the associated program block

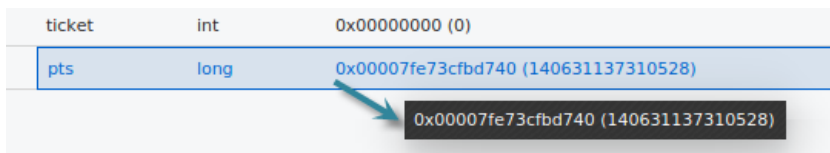


You can expand and collapse the list of variables in a block by clicking the left arrow next to it in the Local Variables pane:

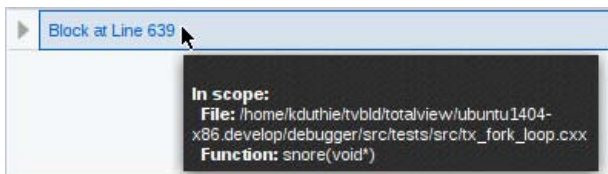


Displaying information in tooltips:

Placing your cursor over a variable displays its value in a tooltip:

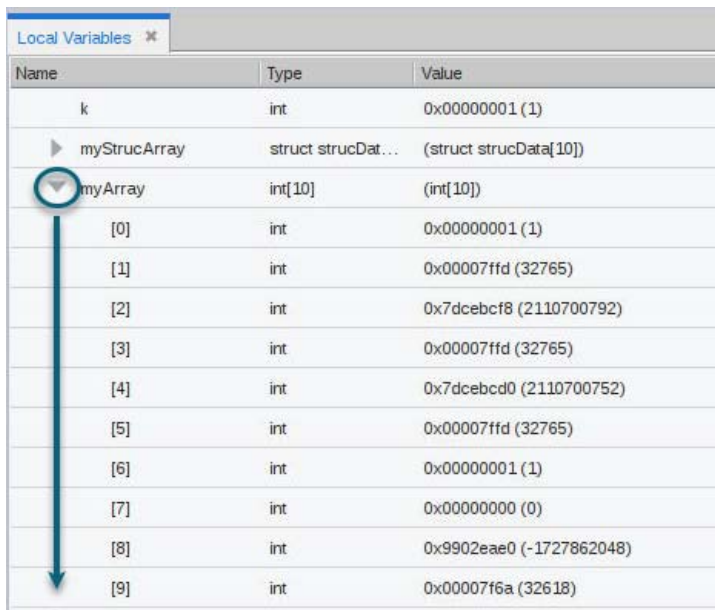


Hovering your cursor over a block displays the file and function in which it appears:



Viewing complex variable structures:

Select the **right arrow** to display the substructures in a complex variable.



Name	Type	Value
k	int	0x00000001 (1)
▶ myStrucArray	struct strucDat...	(struct strucData[10])
▼ myArray	int[10]	(int[10])
[0]	int	0x00000001 (1)
[1]	int	0x00007ffd (32765)
[2]	int	0x7dcebcf8 (2110700792)
[3]	int	0x00007ffd (32765)
[4]	int	0x7dcebcd0 (2110700752)
[5]	int	0x00007ffd (32765)
[6]	int	0x00000001 (1)
[7]	int	0x00000000 (0)
[8]	int	0x9902eae0 (-1727862048)
[9]	int	0x00007f6a (32618)

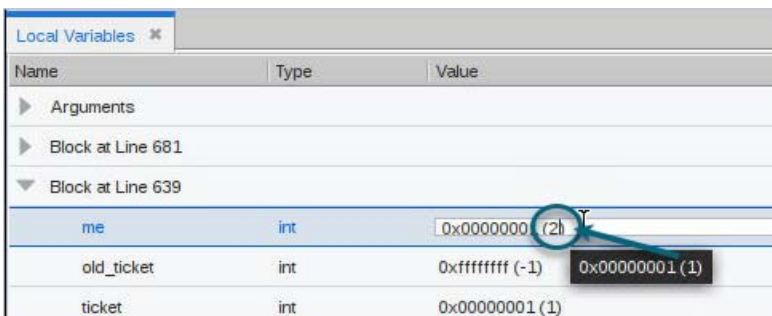
Note that you can also dive on a complex variable that you have copied to the Data View. See [Entering a New Expression into the Data View](#).

Editing the value of a variable:

If your data's value is not what you expect, you can change a variable's value to test a fix. The new value changes the source code for that session only.

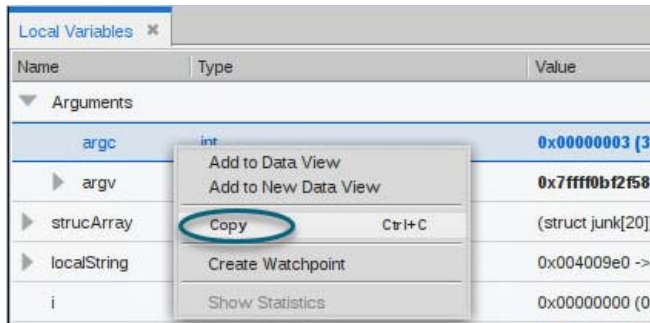
Double-click on the value in the Value column and enter a new value.

For example, this edit changes the value of `me` from 1 to 2:



Name	Type	Value
▶ Arguments		
▶ Block at Line 681		
▼ Block at Line 639		
me	int	0x00000002 (2)
old_ticket	int	0xffffffff (-1)
ticket	int	0x00000001 (1)

Copy a variable definition from the TotalView UI to another document by right-clicking on the variable and selecting **Copy**.



TotalView copies the variable as a tab-separated string so that pasting it into another program results in columns of data delineated by a tab.

The Registers View

The Registers view is not open by default. To open it, from the **Window** menu, choose **Views > Registers**.

The Registers view displays the contents of CPU registers for the selected frame in the Call Stack. Viewing a different stack frame in the Call Stack reloads the values to those relevant to that frame.

The view includes two collapsible sections, **General Purpose Registers** and **Floating Point Registers**.



NOTE: Register abbreviations and meanings are architecture-specific. See [Architectures](#) in the *TotalView Reference Guide*. Note that if your platform doesn't support dedicated floating point registers, all registers will appear in the General Purpose Register section.

The Registers view displays each register's **Name** and **Value** at the time that the thread stopped. If the value has changed, it appears in **bold**, similar to [The Data View](#).

For example, here the general purpose **rax** and **rdx** registers' values have changed:



Name	Value
General Purpose Registers	
%rax	0x00000003 (3)
%rdx	0x00000004 (4)
%rcx	0x00000b48 (2888)
%rbx	0x7fda1e2e5700 (140574785951488)

The value has two parts: the hexadecimal and the annotation in parentheses.

Running threads display no registers, so the view will be empty if all threads are running.

Edit or Cast a Register

To edit the value of a register or cast it to a different type, add it to the Data View. Similar to the Local Variables view, add a register to the Data View by dragging and dropping it into the view or right-clicking on the register entry and selecting **Add to Data View**.

Once in the Data View, you can edit the data by double-clicking on the value and entering a new one, or cast the data by double-clicking the existing type and entering a new one.

RELATED TOPICS

Using the **dwhere** command to view registers [dwhere](#) in the *TotalView Reference Guide*

The Data View [The Data View](#)

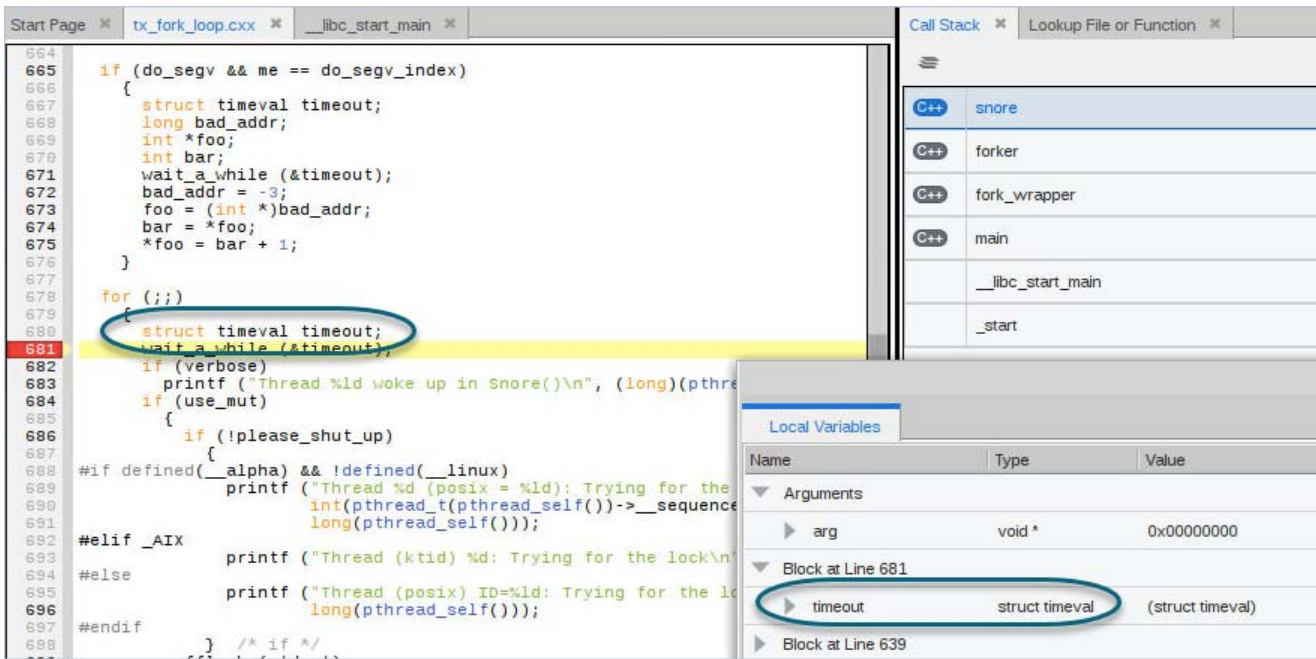
Casting to another type in the Data View [Casting to Another Type](#)

Viewing Call Stack Data

Let's consider the **Source View** alongside the Call Stack. The Source View displays your program's source code and any breakpoints you have set. The highlighted yellow line and arrow shows where execution has stopped.

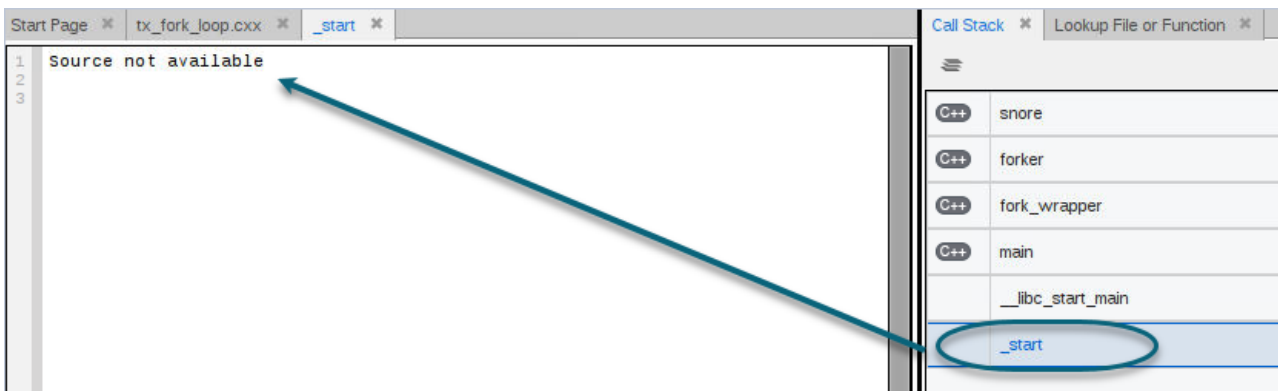
In this example, the program is within the block that starts at line 680, with the **Program Counter**, or PC, stopped at line 681. At that scope is a local struct variable, `timeout`, displayed in the Local Variables view.

Figure 51, The Local Variables view displays local variables for the TOI



The Call Stack view uses icons to identify several functions as C++. Other possible icons include Fortran or C. If a language is displayed, then there is debug information for that frame, so, for instance, `__libc_start_main` and `_start` have no debug information, and the source is not available. If selected, “Source not available” displays.

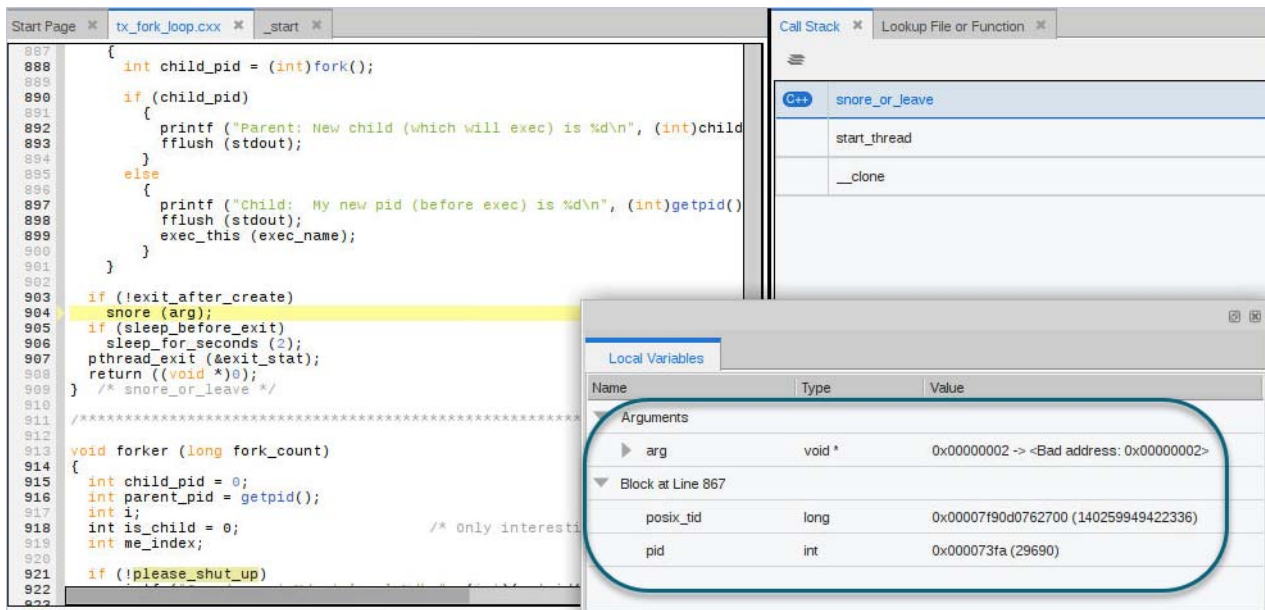
Figure 52, Source not available



Refocusing the Source View updates the Local Variables View

Move up and down in the stack trace in the Call Stack View and select a new frame to refocus the Source View to the selected source for that routine. Figure 53 illustrates that moving down the stack to `snore_or_leave` updates the Local Variables view with new local values and arguments.

Figure 53, Refocusing the Frame



Viewing Data in Fortran

This section demonstrates the display of data when debugging a Fortran program. The behavior of the Call Stack and Local Variables view is essentially the same, but the display differs to reflect Fortran-defined data.

Viewing Modules and Their Data

Fortran 90 lets you place functions, subroutines, and variables inside modules. You can then include these modules elsewhere with a **USE** command. This command makes the names in the module available in the *using* compilation unit, unless you either exclude them with a **USE ONLY** statement or rename them. This means that you don't need to explicitly qualify the name of a module function or variable from the Fortran source code.

When debugging this kind of information, you need to know the location of the function being called, so TotalView uses the following syntax when it displays a function contained in a module:

```
modulename`functionname
```

Variable names are handled similarly:

modulename`variablename

NOTE: This assumes that TotalView is able to determine the module in which the function or variable resides. Sometimes the information the compiler makes available is insufficient and sometimes, although a function uses a module, TotalView is unable to determine that a module is the source of the function, and so is unable to properly qualify the names. In this case names are displayed unqualified as a local function with local variables.

The above qualified syntax can be used with the **Lookup File or Function** view, and with the **dprint** command in the CLI:

dprint *modulename`variablename*

Figure 54 illustrates some of the points made above.

Figure 54, Display of Fortran Module Data

The screenshot displays a Fortran source code editor with the following code:

```

1  MODULE mmm
2  integer*1 www
3  integer xxx
4  integer yyy
5  real zzz
6  contains
7
8  subroutine init
9      xxx = 1
10     yyy = 2
11     zzz = 0.4
12     www = 5
13 end subroutine init
14 END MODULE mmm
15
16 subroutine sub1
17 USE mmm
18
19 www = 1
20 xxx = yyy
21
22 !STOP: check that the
23 call sub2
24
25 end subroutine sub1
26
27 subroutine sub2
28
29 USE mmm, ONLY : yyy=
30 real www
31
32 zzz = 111
33 yyy = 999
34
35 www = 222
36 !STOP: check that the yyy, zzz and www have the correct values
37 end subroutine sub2
38
39 program main
40 use mmm
41 call init
42 call sub1
43
44

```

The Local Variables window shows the following data:

Name	Type	Value
mmm	(Module)	
mmm`www	INTEGER*1	0 (0x00)
mmm`xxx	INTEGER*4	0 (0x00000000)
mmm`yyy	INTEGER*4	0 (0x00000000)
mmm`zzz	REAL*4	0

The qualified subroutine name appears in the Call Stack, and the qualified variable names appear in the Local Variables view. Note also that the **init()** routine can be called from **main** without qualification because of the **USE** statement.

Common Blocks

For each common block defined in the scope of a subroutine or function, TotalView creates an entry in that function's common block list. The names of common block members have function scope, not global scope. If you select the function in the Call Stack, the common blocks and their variables appear in the Local Variables view. From there, of course, you can move those variables to the Data View to create expressions or cast a type, for example.

Figure 55 illustrates the handling of common blocks.

Figure 55, Fortran Common Blocks

The screenshot displays the TotalView interface with the following components:

- Code Editor:** Shows Fortran code for a program named `common_test`. It includes variable declarations for `ix`, `iy`, `iz`, `fo`, `fo_o`, `ix_x`, `iy_y`, and `iz_z`. A common block is defined with members `xp`, `ix`, `iy`, `iz`, `fo`, and `fo_o`. The code also shows the allocation and assignment of `xarray` and `xp`.
- Call Stack:** Lists the current function `common_test` and its callers: `main`, `__libc_start_main`, and `_start`.
- Local Variables:** A table showing the values of local variables:

Name	Type	Value
foo	INTEGER*4	2012 (0x000007dc)
xarray	REAL*8,allocatable::(42,11)	(REAL*8,allocatable::(42,11))
foo	(Common)	
fo_o	(Common)	
xp	REAL*8,pointer::(42,11)	(REAL*8,pointer::(42,11))
ix	INTEGER*4	1 (0x00000001)
iy	REAL*4	2
iz	REAL*8	3
fo	INTEGER*4	42 (0x0000002a)
- Data View:** A table showing the memory layout of the variable `xp`:

Name	Type	Value
xp	REAL*8,poi...	(REAL*8,pointer::(42,11))
(1,1)	REAL*8	9
(2,1)	REAL*8	9

Fortran 90 User-Defined Types

A Fortran 90 user-defined type is similar to a C structure. TotalView displays a user-defined type as **type(name)**, the syntax used in Fortran 90 to create a user-defined type.

For example, the following code fragment defines two user types, **foo** and **bar**:

```
type foo
integer ifoo
end type foo
```

```

type bar
integer mdarray(2,3,4,5)
end type bar

```

And this code creates variables of these types, one a simple type, one an 20-dimensional array, and one a pointer:

```

type (bar), target :: just_a_bar
type (foo), dimension(20), target :: foo_array
type (foo), pointer, dimension(:) :: foo_p

```

TotalView displays these in the Local Variables view, which you can then add to the Data View if you wish to create expressions or cast a type:

Figure 56, Fortran User-Defined Types

The screenshot shows the TotalView interface. On the left, the Fortran source code for a module is displayed. A blue circle highlights the definition of the 'bar' type: `type bar`, `integer mdarray(2,3,4,5)`, and `end type bar`. A blue arrow points from this circle to the 'Local Variables' view on the right. In the 'Local Variables' view, a table lists variables: 'chars' (character(len=50)), 'just_a_bar_p' (type(bar),pointer), 'just_a_bar' (type(bar)), 'just_a_foo_p' (type(foo),pointer), 'just_a_foo' (type(foo)), 'foo_array' (type(foo)(20)), and 'total' (INTEGER*4). A blue circle highlights 'just_a_bar' in this table, with another blue arrow pointing to the 'Data View' window. The 'Data View' window shows a table with columns 'Name', 'Type', and 'Value'. It lists 'foo_p', 'just_a_bar', 'mdarray', 'foo_array', and '(1)'. A blue circle highlights 'just_a_bar' in this table, with a blue arrow pointing to its value '(type(bar))'.

Fortran 90 Deferred Shape Array Types

Fortran 90 lets you define deferred shape arrays and pointers. The actual bounds of a deferred shape array are not determined until the array is allocated, the pointer is assigned, or, in the case of an assumed shape argument to a subroutine, the subroutine is called.

The following example shows the type of a deferred shape array of real data with no defined lower or upper bounds:

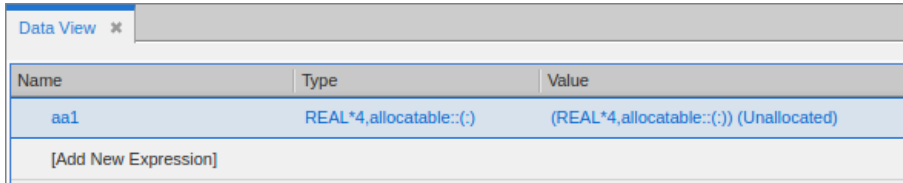
```

real, allocatable, dimension(:) :: aal

```

Here is the unallocated array displayed in the Data View:

Figure 57, Fortran Deferred Shape Array, unallocated



As you run the program, the array is allocated at line 303 below. Note that the type has been modified in the Data View.

Figure 58, Fortran Deferred Shape Array

```

301  pa1 => pa2
302  pa5 => arrays(:,:,1,1,1,1)
303  allocate(aa1(20:50))
304  do i = 20, 50
305     aa1(i) = i*i
306  end do
307  !STOP!
308
309  char
310
311  call
312
313  print
314
315  call
316  call
317  call
318
319  call
320  call
321  call

```

Name	Type	Value
aa1	REAL*4,allocatable::(20:50)	(REAL*4,allocatab...
(20)	REAL*4	400
(21)	REAL*4	441
(22)	REAL*4	484
(23)	REAL*4	529
(24)	REAL*4	0

Fortran 90 Pointer Types

A Fortran 90 pointer type points to scalar or array types.

TotalView implicitly handles slicing operations that set up a pointer or assumed shape subroutine argument so that the indices and values it displays in the Local Variables view and Data View are the same as in the code. For example, this code sets up and assigns an array and a pointer to that array:

```
integer, target, dimension(5,2:10) :: ia21,ia22
```

```
integer, pointer, dimension(:, :) :: ip21, ip22
ip21 => ia21(:, 2, : : 2)
```

Figure 59 displays the original array `ia21` and its pointer `ip21` in the Data View.

Figure 59, Original Fortran Array

Name	Type	Value
▶ ia21	INTEGER*4(5,2:10)	(INTEGER*4(5,2:10))
ip21	INTEGER*4,pointer::(:, :)	(INTEGER*4,pointer::(:, :)) (Unassociated)
[Add New Expression]		

Figure 60 illustrates the pointer `ip21` representing a slice of the `ia21` array after the assignment of the pointer.

Figure 60, Fortran Pointer Representing an Array Slice

Name	Type	Value
▶ ia21	INTEGER*4(5,2:10)	(INTEGER*4(5,2:10))
▶ ip21	INTEGER*4,pointer::(3,3)	(INTEGER*4,pointer::(3,3))
[Add New Expression]		

Fortran Parameters

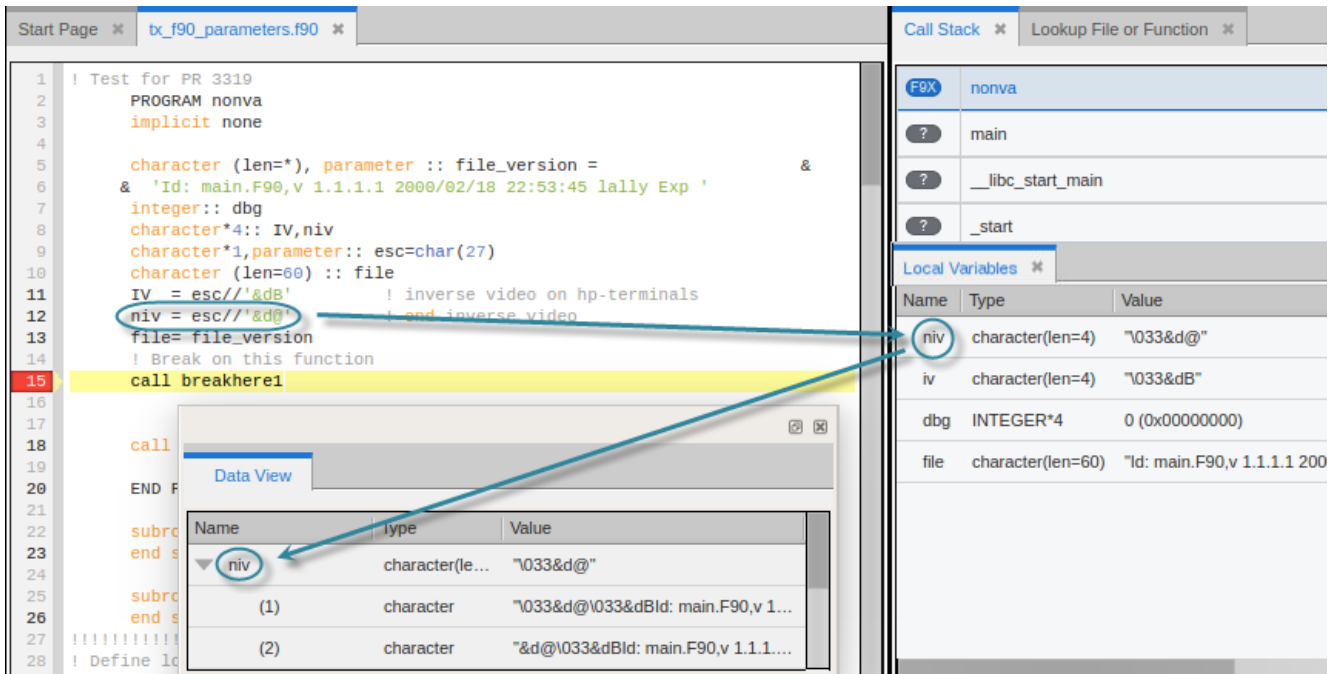
A Fortran **PARAMETER** defines a named constant. If your compiler generates debug information for parameters, they are displayed in the same way as any other variable. However, some compilers do not generate information that TotalView can use to determine the value of a **PARAMETER**. This means that you must make a few changes to your program if you want to see this type of information. For Fortran 90, you can define variables in a module that you initialize to the value of these **PARAMETER** constants; for example:

```
INCLUDE 'PARAMS.INC'
MODULE CONSTS
SAVE
INTEGER PI_C = PI
...
END MODULE CONSTS
```


The **PARAMS.INC** file contains your parameter definitions. You then use these parameters to initialize variables in a module. After you compile and link this module into your program, the values of these parameter variables are visible. For Fortran 77, you can achieve the same results if you make the assignments in a common block and then include the block in `main()`. You can also use a block data subroutine to access this information.

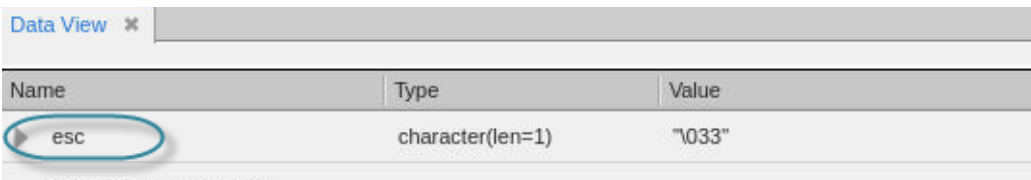
Figure 61 assigns a parameter to a local variable for display in the Data View:

Figure 61, Fortran Parameters



If the compiler provides enough information to look at parameters directly, then you can add the parameter directly to the Data View, like so:

Figure 62, Fortran Parameters, added to Data View



The Data View

The Data View enables you to create expressions in order to analyze your data.

Once a variable is displayed in the Data View, you can manipulate it in multiple ways in order to clearly see what your data is doing.

- [Adding Variables to the Data View](#)
- [Diving on Variables](#)
- [Working with Complex Variables in the Data View](#)
- [Editing an Expression](#)
- [Displaying Arrays](#)
- [Viewing Individual Elements in an Array of Structures](#)
- [Customizing the Data View](#)

Adding Variables to the Data View

Once you have started your program and it has stopped at a breakpoint or by using a stepping command, the Local Variables view populates with local data from whatever stack frame is selected in the Call Stack. The Source view is also refocused on the source file associated with the selected frame.

Creating a new expression allows you to manipulate your data in multiple ways in order to clearly see your data's structure, type, and value at any given point in your program's execution. To add expressions, i.e. variables, to the Data View, use one of these methods:

- [Add to the Data View from the Local Variables View](#) using either the context menu or by dragging and dropping.
- [Create a New Expression from within the Data View.](#)
- [Move a Variable from the Source View to the Data View](#) using the context menu.
- [Entering a New Expression into the Data View.](#)

NOTE: Because global variables are not displayed in the Local Variables view, add them to the Data View by typing them in directly. See [Entering a New Expression into the Data View](#).

Add to the Data View from the Local Variables View

Use the Local Variables view to add variables to the Data View, either by using the context menu or by dragging and dropping. Choose **Add to Data View** to add to the existing view, or **Add to New Data View** to create a new Data View window.

Figure 63, Add to the Data View using the context menu

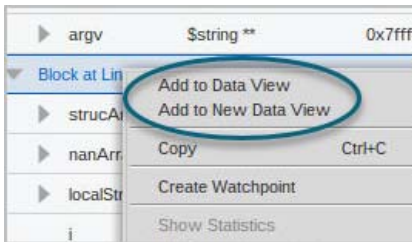
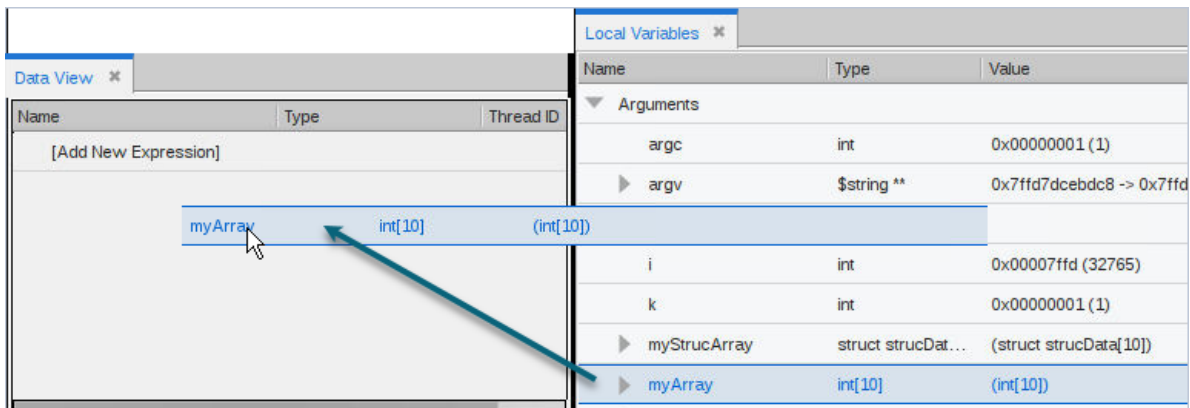
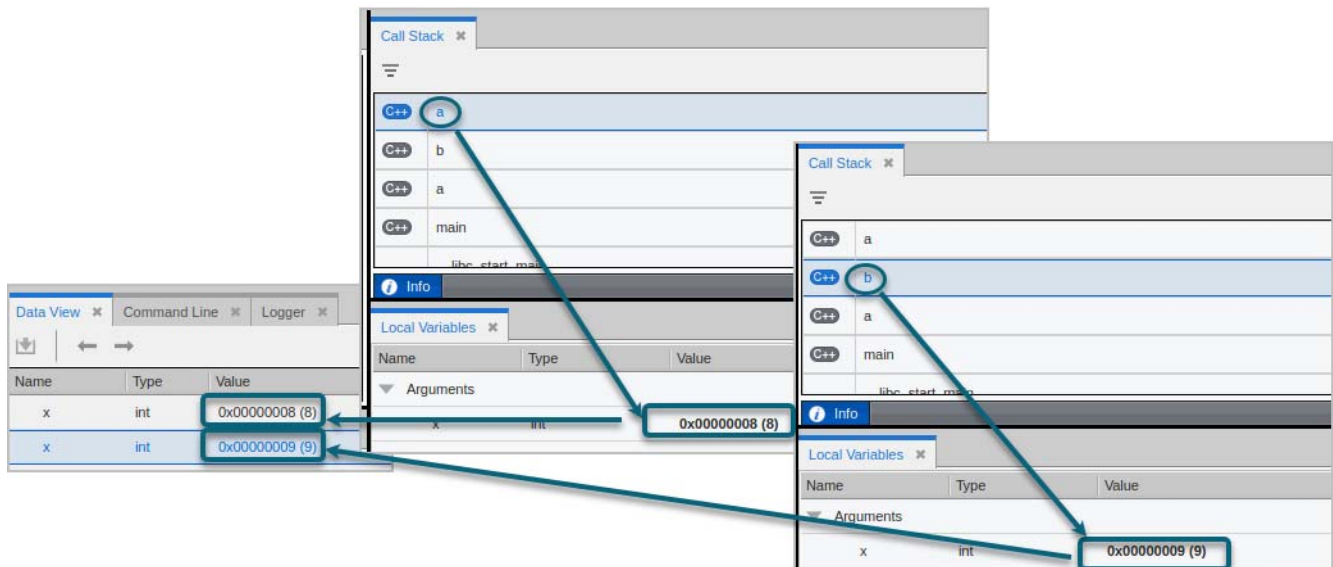


Figure 64, Add to the Data View by dragging from Local Variables



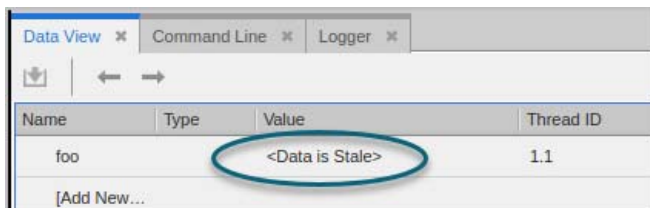
As you step through your program and the focus changes, the variables are evaluated using the scope in which they were added rather than the current scope.

Figure 65, Evaluated expressions use the original scope



In Figure 65, the variable **x** was added to the Data View at two different scopes from two different, selected stack frames, so reflects a different value.

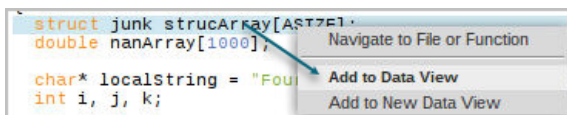
If a variable goes out of scope, an error is displayed:



Move a Variable from the Source View to the Data View

Use the context menu in the Source view to add a variable to the Data View.

Right-click on the variable and select **Add to Data View**.



You can also select an entire valid expression:

```

61     myArray[i] = i+j;
62     myStrucArray[i].x = i;
63     myStrucArray[i].y = i*2;
64     myStrucArray[i]
65     myStrucArray[i]
66     printf("level 1
67     if ( i < 5)

```

The expressions are added to the Data View:

Name	Type	Value
myArray	int[10]	(int[10])
myStrucArray[i]	struct strucData	(struct strucData)
[Add New Expression]		

Create a New Expression from within the Data View

You can add a new expression to the Data View in multiple ways:

- **Dragging and dropping** an existing expression to the row **[Add a new Expression]**.

Here, the `int i` is added, resulting in two copies in the Data View.

Name	Type	Value
stringArray	\$string *[20]	(\$string *[20])
i	int	0x00000014 (20)
[Add New Expression]		
i	int	0x00000014 (20)

- Using the **Dive in New Data View** or **Duplicate** features, available from the context menu. Right-click on an expression in the Data View and select **Dive in New Data View** or **Duplicate**. See [Dive in New Data View](#) or [Duplicating an Expression](#).
- Entering a new expression manually. See [Entering a New Expression into the Data View](#).

Entering a New Expression into the Data View

Because variables are actually expressions— in fact, lvalues that evaluate to a memory location— you can enter an expression into the Name field to troubleshoot data problems.

To add a new expression, double-click on **Add New Expression** in the Data View and begin typing.

Some examples:

- **View just a sub-element of a structure**, by entering it as a new expression:

Name	Type	Value
myStruct	struct struc...	(struct strucData)
x	int	0x0000000a (10)
y	int	0x00000014 (20)
b	float	777.2
ns	struct neste...	(struct nestedStruct)
nsa	int[3]	(int[3])
myStruct.x		

A new expression is added. This is the same as diving on the sub-element.

myStruct.x	int	0x0000000a (10)
------------	-----	-----------------

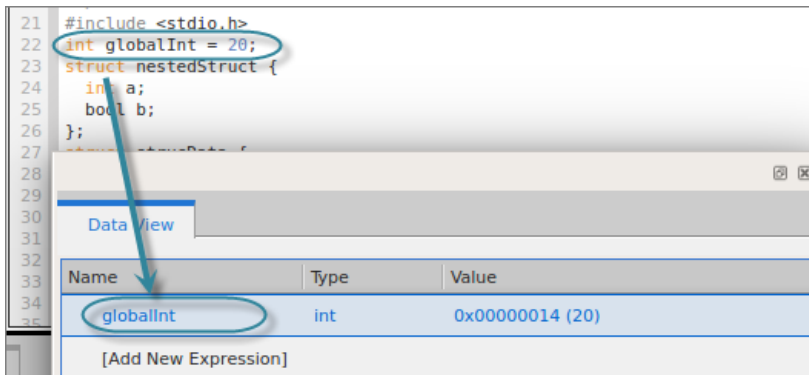
- **Increment a variable:**

myStruct.x + 5	int	0x0000000f (15)
----------------	-----	-----------------

- **Add or subtract variables**, assuming some relationship:

Name	Type	Value
j	int	0x00000008 (8)
f	float	10.2
f+j	float	18.2

- **Add a global variable.** Type the variable name into the Name column and TotalView automatically enters the type and value.



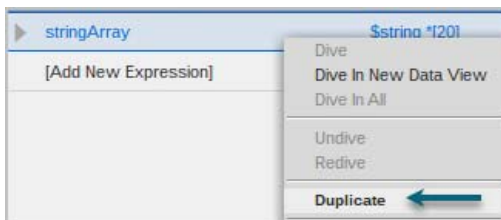
The value updates in the Data View as you run through your program.

Operating on Multiple Expressions

You can select multiple expressions to perform an operation on, such as to duplicate, dive, or delete. Hold down the **Ctrl** key and select the expressions, then right-click for the context menu.

Duplicating an Expression

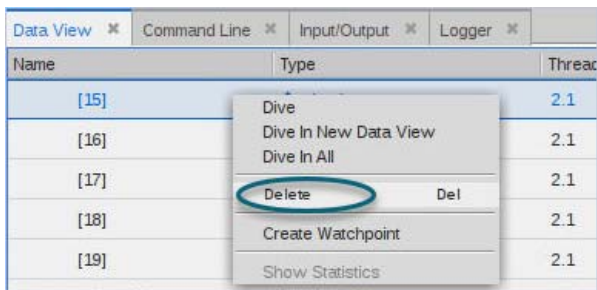
Choosing **Duplicate** from the context menu makes a copy of the variable and places it at the bottom of the existing Data View. You can duplicate any variable, including a single scalar variable, an entire complex variable, or a subelement of a complex variable.



If the duplicated expression had been “dived on,” the dive stack is also duplicated. See [Diving on Variables](#).

Deleting an Expression

Delete an expression using the context menu and selecting **Delete** or by clicking **Delete** on your keyboard.



NOTE: For complex variables, deleting a single element deletes the top-level variable from the Data View. You cannot delete a single element of a structure or an array.

Diving on Variables

If your data is complex, you can dive on it to drill down for detail on its subelements. For example, dive on an expression or variable in the Data View to view a nested data item at the top level. The nested item could be a structure member, an element in an array, or the target of a pointer.

Dive on a variable via a context menu or use the **Dive** button. From the context menu, choose from several options related to diving:

- **Dive.** Available only for complex variables when a subelement is selected. This option *dives* into the subelement, replacing its parent element in the view. See [Diving on Complex Variables](#).
- **Dive in New Data View.** Select this option to launch a new Data View window containing the variable. This may be useful if you want multiple windows in which to view different elements of a variable. See [Dive in New Data View](#).
- **Dive In All.** This option is relevant only for arrays. See [The Dive In All Command](#).
- **Redive or Undive.** These options are enabled only when a dive has already been performed. See [Redive and Undive](#).

Working with Complex Variables in the Data View

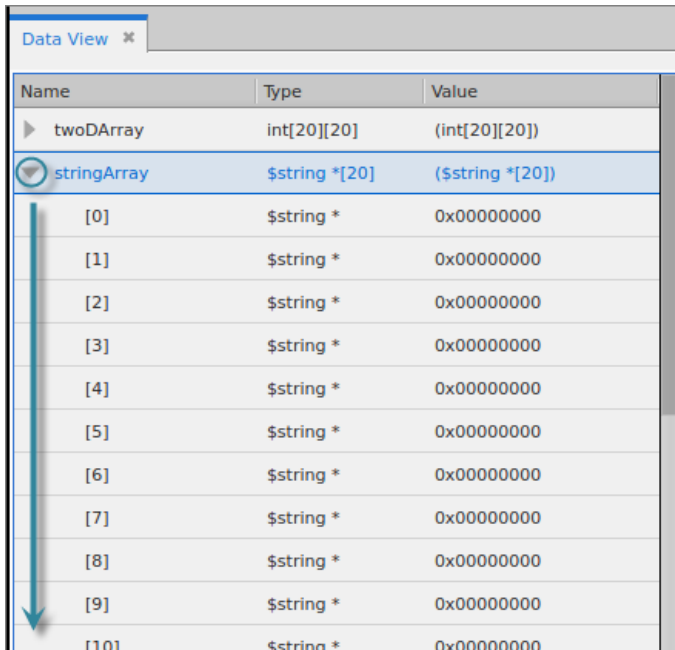
View, edit, and analyze complex variables using the Data View.

- [Viewing Elements of Complex Variables](#)

- Diving on Complex Variables

Viewing Elements of Complex Variables

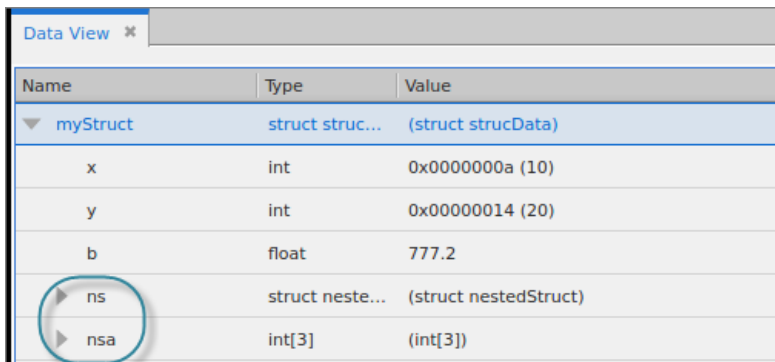
Select the **right arrow** to display the substructures in a complex variable.



The screenshot shows a 'Data View' window with a table of variables. The 'stringArray' variable is selected, and a blue right-pointing arrow is positioned over its first element, [0].

Name	Type	Value
▶ twoDArray	int[20][20]	(int[20][20])
▼ stringArray	\$string *[20]	(\$string *[20])
[0]	\$string *	0x00000000
[1]	\$string *	0x00000000
[2]	\$string *	0x00000000
[3]	\$string *	0x00000000
[4]	\$string *	0x00000000
[5]	\$string *	0x00000000
[6]	\$string *	0x00000000
[7]	\$string *	0x00000000
[8]	\$string *	0x00000000
[9]	\$string *	0x00000000
[10]	\$string *	0x00000000

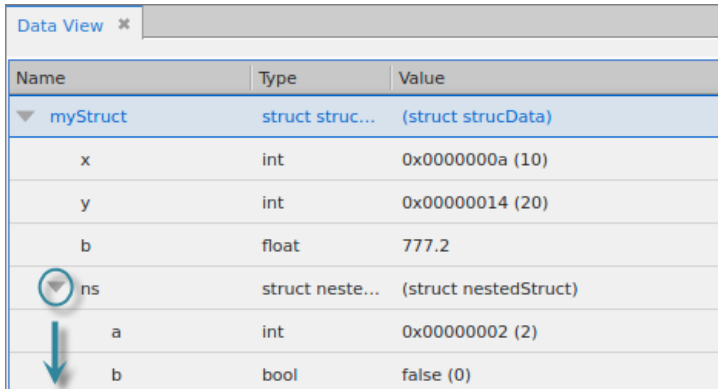
If your complex variable has nested structures, these display in the Data View:



The screenshot shows a 'Data View' window with a table of variables. The 'myStruct' variable is expanded, and a blue right-pointing arrow is positioned over the 'ns' field.

Name	Type	Value
▼ myStruct	struct struc...	(struct strucData)
x	int	0x0000000a (10)
y	int	0x00000014 (20)
b	float	777.2
▶ ns	struct neste...	(struct nestedStruct)
▶ nsa	int[3]	(int[3])


View the nested structures by selecting the down arrow:

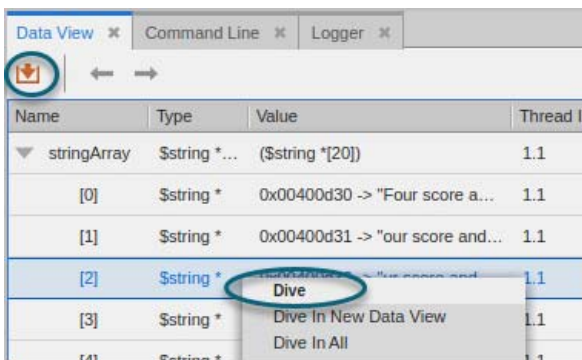


Name	Type	Value
myStruct	struct struc...	(struct strucData)
x	int	0x0000000a (10)
y	int	0x00000014 (20)
b	float	777.2
ns	struct neste...	(struct nestedStruct)
a	int	0x00000002 (2)
b	bool	false (0)

Diving on Complex Variables

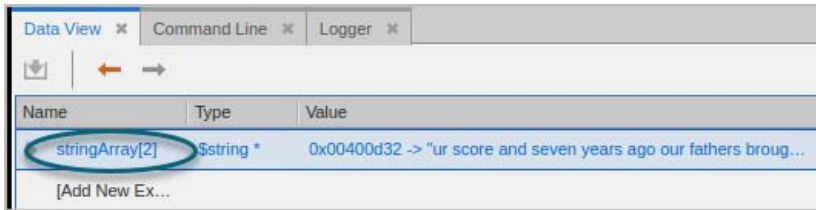
The subelements of complex structures can be analyzed by diving on them. Diving on a subelement of a compound structure replaces the top level structure with the subelement, on which you can further dive (if, for instance, you have an array of structures), so that you can edit and manipulate it as needed.

Right-click on a subelement in the Data View and select **Dive**, or click the **Dive** arrow ().



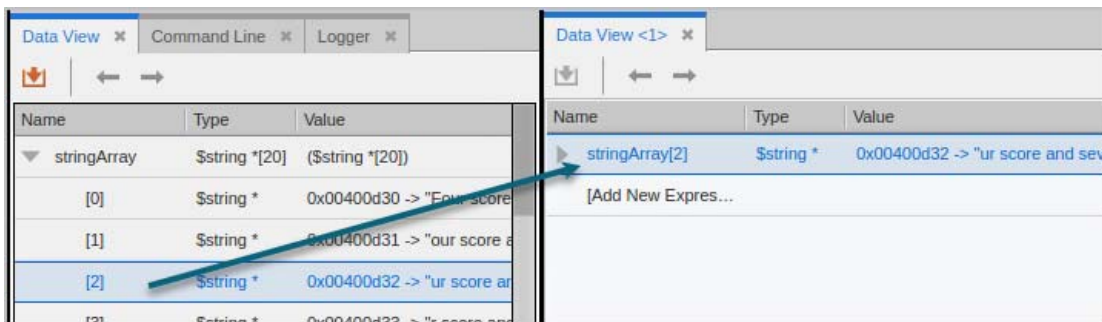
Name	Type	Value	Thread I
stringArray	\$string *...	(\$string *[20])	1.1
[0]	\$string *	0x00400d30 -> "Four score a...	1.1
[1]	\$string *	0x00400d31 -> "our score and...	1.1
[2]	\$string *	0x00400d32 -> "the score and...	1.1
[3]	\$string *	0x00400d33 -> "the score and...	1.1
[4]	\$string *	0x00400d34 -> "the score and...	1.1

The new expression *replaces* the selected expression.



Dive in New Data View

You may wish to retain the parent variable in the Data View rather than replacing it in the view. Choosing **Dive in New Data View** from the context menu instead of Dive creates a *new* Data View containing the selected element. The new window is numbered, starting with "<1>." If you undock or redock one of the Data View windows, you can view both the parent element and the subelement on which you dived side-by-side.



You can also duplicate a variable, which makes a copy and places it at the bottom of the existing Data View. See [Duplicating an Expression](#).

Redive and Undive

When diving on compound structures, use the **Redive** and **Undive** arrows (← →) to move up and down the dive stack.

- After diving on an element in a compound variable, the **Undive** arrow (←) becomes active, and **Undive** is available in the context menu. Choosing Undive moves one level back up the dive stack.
- After choosing Undive, the **Redive** arrow (→) becomes active. Rediving repeats the previous dive.

When diving on arrays or other variables with multiple, nested structures, you can use Dive In All to display a single field of a structure in an array of structures as a *new array* across all the structures. See [The Dive In All Command](#).

Editing an Expression

Edit an expression in the Data View by double-clicking inside any field to make the text editable. You can enter any expression into the Name field, changing a variable's name, type, or value.

Dereferencing a Pointer

When you dive on a variable, it is not dereferenced automatically. To dereference it so you can see its target, edit the expression. For example, for this pointer to a string:

Name	Type	Value
▶ localString	\$string *	0x004009c8 -> "HelloHelloHelloHelloHel..."

Double-click in the **Name** column to make the text editable, and then dereference the pointer:

Name	Type
*localString	\$string

The Data View displays the variable's value:

Name	Type	Value
*localString	\$string	"HelloHelloHelloHelloHelloHello"

For **argv**, i.e. a pointer to a pointer, dereference it twice.

Name	Type	Value
*localString	\$string	"HelloHelloHelloHelloHelloHello"
**argv	\$string	"/home/bburns/sandbox/build/linux-x86-..."

Changing the Value of Data

If your data's value is not what you expect, you can change a variable's value to test a fix. The new value changes the source code for that session only. If you kill the program and restart it, the previous value is reinstated.

Double-click on the value in the Value column and enter a new value.

For example, this example changes the value of `x` from 10 to 100:

Name	Type	Value
myStruct	struct struc...	(struct strucData)
x	int	0x0000000a (100)
y	int	0x00000014 (20) 0x0000000a (10)

Casting to Another Type

You may need to cast your data to a type that is more meaningful. Enter the cast code in the Type field. Here are some examples.

Casting to an Array in the Data View

Cast a variable into an array by adding an array specifier to the Type declaration. For example, adding `[3]` to a variable declared as an `int` changes it into an array of three `ints`.



Press **Enter** to cast the variable:

j	int[3]	(int[3])
[0]	int	0x0000000a (10)
[1]	int	0x00000064 (100)
[2]	int	0x00000010 (16)

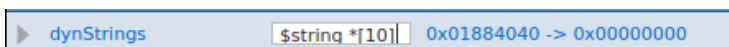
Depending on the array declaration, TotalView displays arrays differently. See [Displaying Arrays](#).

Displaying an Allocated Array

Display an allocated array. Using `malloc()` (in C and C++) creates a pointer to allocated memory. For example:

```
dynStrings = (char**) malloc (10 * sizeof(char*));
```

Because the debugger doesn't know that this is a pointer to an array of `ints`, to display the array, change its type to `$string *[10]`.



Then click the down arrow to display your array of 10 strings.

Name	Type	Value
▼ dynStrings	\$string *[10]	(\$string *[10])
[0]	\$string *	0x7fffb7b95a80 -> "\377\377\377\377"
[1]	\$string *	0xf63d4e2e -> <Bad address: 0xf63d4e2e>
[2]	\$string *	0x0040032a -> "_libc_start_main"
[3]	\$string *	0xffffffff -> <Bad address: 0xffffffff>
[4]	\$string *	0x7fffb7b95be8 -> "\022b\271\267\377\177"
[5]	\$string *	0x7f761afdd1a8 -> "2("
[6]	\$string *	0x7f761b5b34c0 -> ""
[7]	\$string *	0x7f761b5b61c8 -> ""
[8]	\$string *	0x00000000
[9]	\$string *	0x00000001 -> <Bad address: 0x00000001>

Built-In Types

TotalView provides a number of predefined types. These types are preceded by a **\$**. You can use these built-in types anywhere you can use those defined in your programming language. These types are also useful in debugging executables with no debugging symbol table information. The following table describes the built-in types:

Table 3: TotalView Built-in Types

Type String	Language	Size	Description
\$address	C	void*	Void pointer (address).
\$char	C	char	Character.
\$character	Fortran	character	Character.
\$code	C	architecture-dependent	Machine instructions. The size used is the number of bytes required to hold the shortest instruction for your computer.
\$complex	Fortran	complex	Single-precision floating-point complex number. The complex types contain a real part and an imaginary part, which are both of type real .
\$complex_8	Fortran	complex*8	A real*4 -precision floating-point complex number. The complex*8 types contain a real part and an imaginary part, which are both of type real*4 .
\$complex_16	Fortran	complex*16	A real*8 -precision floating-point complex number. The complex*16 types contain a real part and an imaginary part, which are both of type real*8 .

Table 3: TotalView Built-in Types

Type String	Language	Size	Description
\$double	C	double	Double-precision floating-point number.
\$double_precision	Fortran	double precision	Double-precision floating-point number.
\$extended	C	architecture-dependent; often long double	Extended-precision floating-point number. Extended-precision numbers must be supported by the target architecture. In addition, the format of extended floating point numbers varies depending on where it's stored. For example, the x86 register has a special 10-byte format, which is different than the in-memory format. Consult your vendor's architecture documentation for more information.
\$float	C	float	Single-precision floating-point number.
\$int	C	int	Integer.
\$integer	Fortran	integer	Integer.
\$integer_1	Fortran	integer*1	One-byte integer.
\$integer_2	Fortran	integer*2	Two-byte integer.
\$integer_4	Fortran	integer*4	Four-byte integer.
\$integer_8	Fortran	integer*8	Eight-byte integer.
\$logical	Fortran	logical	Logical.
\$logical_1	Fortran	logical*1	One-byte logical.
\$logical_2	Fortran	logical*2	Two-byte logical.
\$logical_4	Fortran	logical*4	Four-byte logical.
\$logical_8	Fortran	logical*8	Eight-byte logical.
\$long	C	long	Long integer.
\$long_long	C	long long	Long long integer.
\$real	Fortran	real	Single-precision floating-point number. When using a value such as real , be careful that the actual data type used by your computer is not real*4 or real*8 , since different results can occur.
\$real_4	Fortran	real*4	Four-byte floating-point number.
\$real_8	Fortran	real*8	Eight-byte floating-point number.
\$real_16	Fortran	real*16	Sixteen-byte floating-point number.

Table 3: TotalView Built-in Types

Type String	Language	Size	Description
\$short	C	short	Short integer.
\$string	C	char	Array of characters.
\$void	C	long	Area of memory.
\$wchar	C	<i>platform-specific</i>	Platform-specific wide character used by wchar_t data types
\$wchar_s16	C	16 bits	Wide character whose storage is signed 16 bits (not currently used by any platform)
\$wchar_u16	C	16 bits	Wide character whose storage is unsigned 16 bits
\$wchar_s32	C	32 bits	Wide character whose storage is signed 32 bits
\$wchar_u32	C	32 bits	Wide character whose storage is unsigned 32 bits
\$wstring	C	<i>platform-specific</i>	Platform-specific string composed of \$wchar characters
\$wstring_s16	C	16 bits	String composed of \$wchar_s16 characters (not currently used by any platform)
\$wstring_u16	C	16 bits	String composed of \$wchar_u16 characters
\$wstring_s32	C	32 bits	String composed of \$wchar_s32 characters
\$wstring_u32	C	32 bits	String composed of \$wchar_u32 characters

Displaying Arrays

The declaration of an array can include a lower and upper bound separated by a colon (:).

For C or C++, the default lower bound is **0**; for Fortran, it is **1**. Further, C and C++ use brackets to define an array, while Fortran uses parentheses. In the following example, an array of ten integers is declared in C and then in Fortran:

```
int a[10];
integer a(10)
```

The elements of the array range from **a[0]** to **a[9]** in C, while the elements of the equivalent Fortran array range from **a(1)** to **a(10)**.

When an array's lower bound is the default, the UI displays only the extent (that is, the number of elements in the dimension). Consider the following Fortran array declaration:

```
integer a(1:7,1:8)
```


Since both dimensions of this array use Fortran's default **1** lower bound, TotalView displays the data type using only the extent of each dimension, as follows:

```
integer(7,8)
```

If an array declaration doesn't use the default lower bound, TotalView displays both the lower and upper bound for each dimension. For example, this Fortran array is displayed in TotalView the same way it is declared:

```
integer a(-1:5,2:10)
```

RELATED TOPICS

Cast a variable to an array

[Casting to an Array in the Data View](#)

Viewing an array in the Array View

[The Array View](#)

Viewing Individual Elements in an Array of Structures

If you have an array of structures data object, you may wish to analyze a particular field of the structure in all the elements of the array; this would require expanding all the elements and then scrolling through the expanded array and visually searching through for that single field.

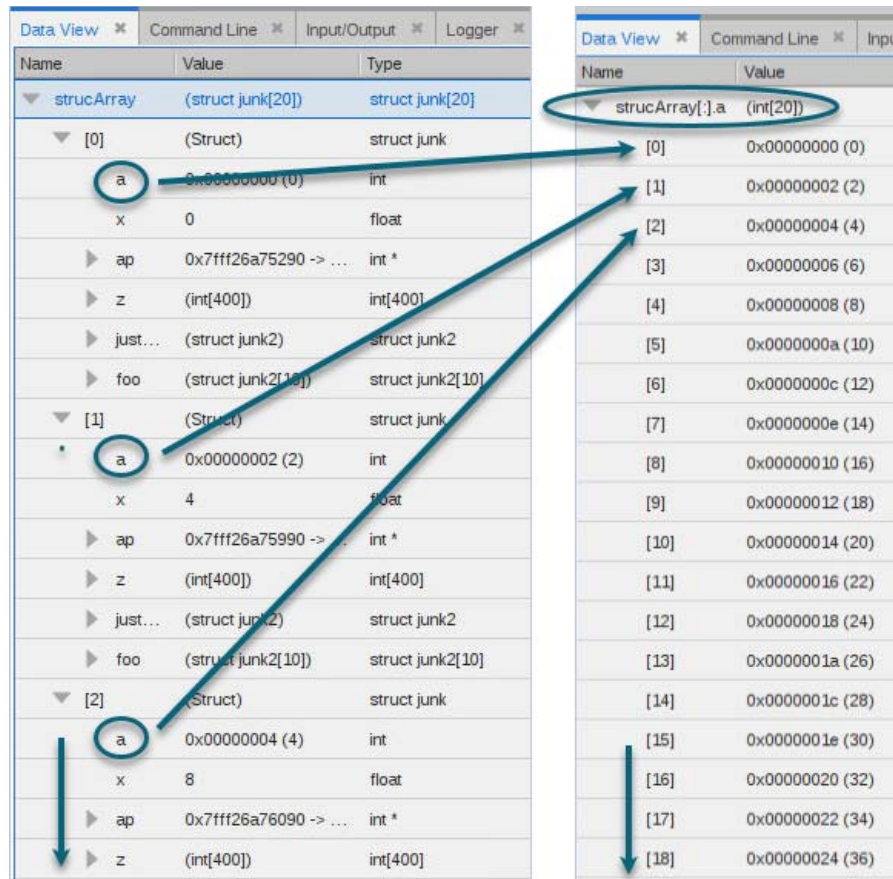
To avoid this painstaking process, use the **Dive-In-All** command.

The Dive In All Command

The **Dive-In-All** command can display a single field of a structure in an array of structures as a *new array* across all the structures. This makes it much easier to look at the values of only that field.

In [Figure 66](#), the array of structures `strucArray` in the left pane contains 20 structures, far more than the Data View can actually display. After you select the **Dive in All** command with element `a` selected, TotalView creates a new array in the Data View that contains all these `a` elements across all structures, shown in the right pane.

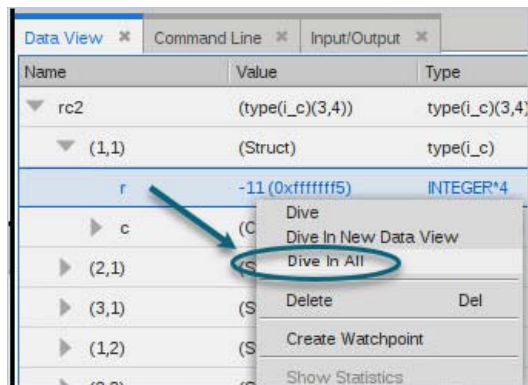
Figure 66, Dive In All creates a new array to visualize a single element across structures



Accessing Dive In All

To Dive-In-All on a variable, right-click a field in a single element of the array in the Data View, and choose **Dive In All** (short for “dive in all elements”) from the context menu.

Figure 67, Dive In All context menu



NOTE: If the array created after a Dive-In-All command is a scalar type, you can display more detail by right-clicking it and selecting **Show Statistics**. See [Viewing Array Statistics](#).

Displaying a Fortran Structure

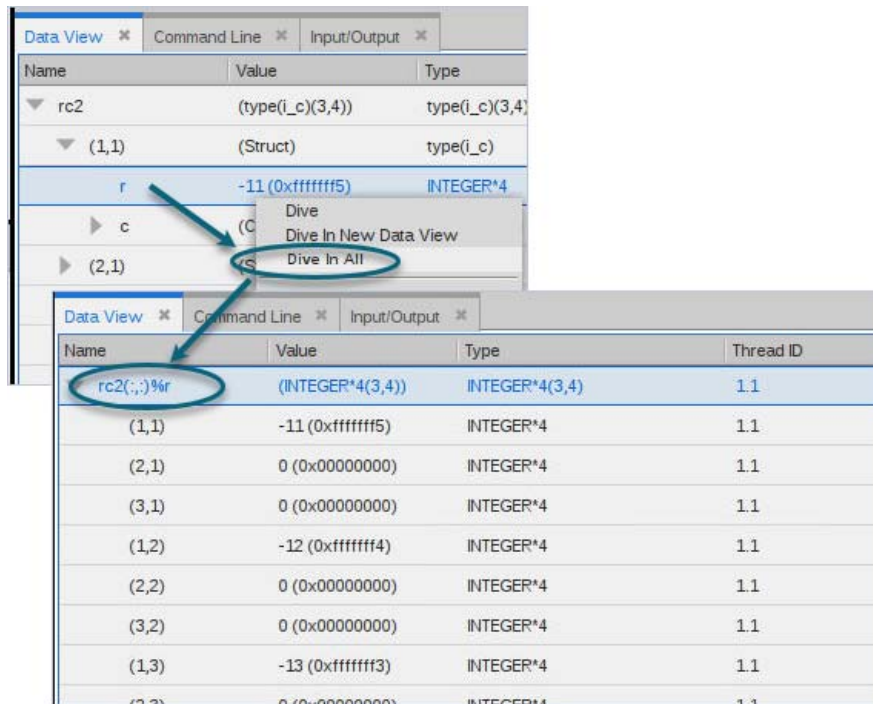
Consider the following Fortran definition of an array of structures:

```
type i_c
  integer r
  complex c
end type i_c

type(i_c), target :: rc2(3,4)
```

With the array in the Data View, select an **r** element, right click, then select **Dive In All**. TotalView displays all of the **r** elements of the **rc2** array as if they were a single array.

Figure 68, Displaying a Fortran Structure



Displaying a C++ Structure

The **Dive in All** command can also display the elements of a C array of structures as arrays. Figure 69 shows a unified array of structures and a multidimensional nested array in a structure.

Figure 69, Displaying C Structures and Arrays

The figure consists of three overlapping screenshots of a debugger's Data View window, illustrating the navigation through nested C structures and arrays. Red circles and arrows highlight the path from the top-level array to the nested structure array and then to its individual elements.

Top Screenshot: Shows the root of the data view. The variable `strucArray` is of type `struct junk[20]`. Its element `[0]` is expanded to show fields `a` (int, 0x00000000) and `x` (float). The field `foo` is highlighted with a red circle.

Middle Screenshot: Shows the expanded view of `strucArray[0].foo`, which is of type `struct junk2[10]`. It contains an array of four `struct junk2` elements, indexed `[0][0]` through `[0][3]`. The element `[0][0]` is highlighted with a red circle.

Bottom Screenshot: Shows the expanded view of `strucArray[0].foo[0].b`, which is of type `int[20]`. It contains an array of five `int` elements, indexed `[0][0]` through `[0][4]`. The element `[0][0]` is highlighted with a red circle.

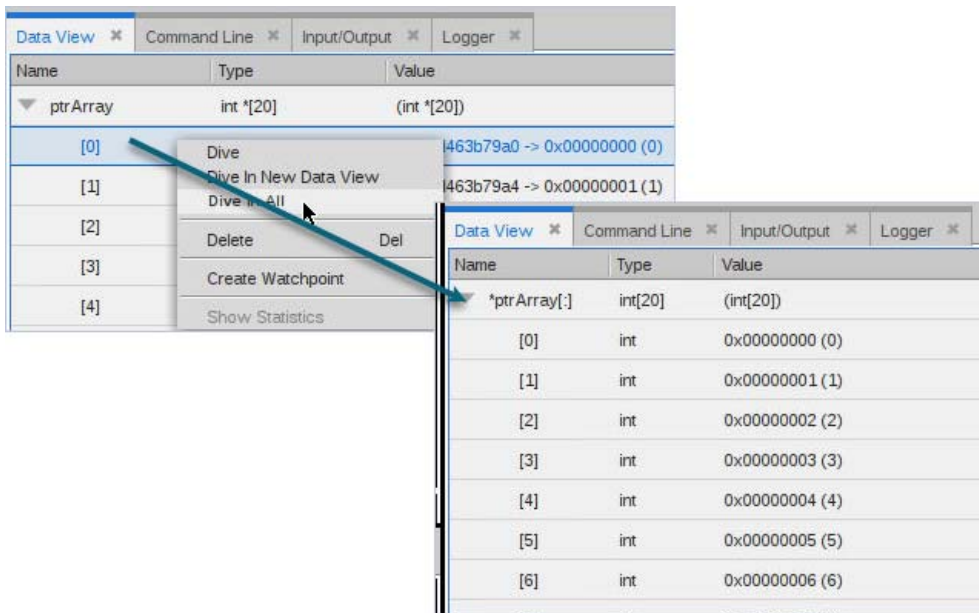
Here, the array of structures **foo** contains a nested array of structures **b**, visually represented like so:

- The array `strucArray[:].foo` visually represents an array composed of every **foo** element in each of the 20 `strucArray` objects.
- Drilling further, the array `strucArray[:].foo[:].b` visually represents an array composed of every **b** element in the array of structures `strucArray[:].foo`.

Using Dive In All to Dereference a Pointer in an Array of Pointers

If you have an array of pointers that you first want to dereference to see the values, use **Dive In All** on a pointer to create a new array of the dereferenced value. In [Figure 70](#), `ptrArray` is an array of pointers of integers. Using **Dive In All** on a single pointer creates a new array of just integers in which each element of the original array is dereferenced.

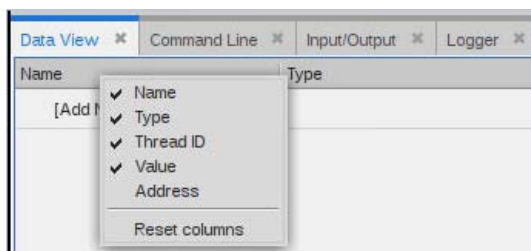
Figure 70, Dive In All used to dereference a pointer in an array of pointers



Customizing the Data View

Customizing the Displayed Columns

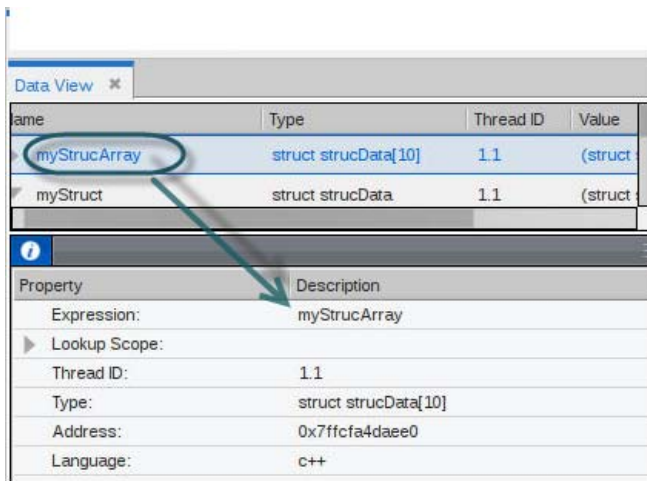
Customize the columns to display by right-clicking anywhere on the column header and selecting the column names.



You can also create a new Data View for more options in viewing data. See [Dive in New Data View](#).

The Data View Drawer

The Data View drawer, available by double-clicking on its bottom banner (see [Drawers](#) for more information on working with drawers) displays detailed information about the selected expression. If more than one expression is selected, the first is used to propagate information in the drawer.



In addition to some of the detail available in the view itself, the drawer displays the language and the *lookup scope*. The lookup scope is the scope of the program where the lookup was initiated.



The lookup scope consists of the executable image, the file in the image, and function name in the file. The Block Line field is typically the first line number in the program's lexical block in which the lookup occurred. The Block ID is merely a synthetic name that TotalView assigns to every lexical block in a program. Hovering your cursor over Block ID displays more detailed description in a tool tip.

The Array View

The Array View provides a way to visualize array data, examine array statistics, and slice arrays for easier analysis.

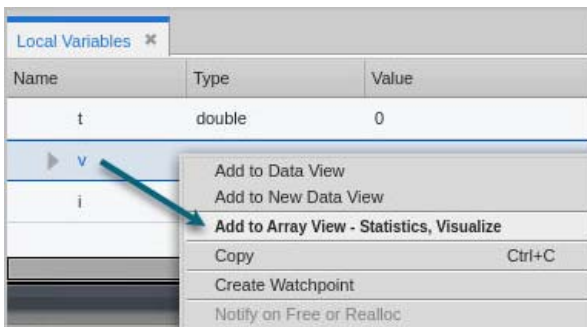
The Array View may be open when you launch TotalView; if not, open it from **Windows > Views > Array View**. You can also open it by adding an array to it.

- Adding Arrays to the Array View
- Viewing Array Statistics
- Visualizing Array Data
- Configuring Arrays

Adding Arrays to the Array View

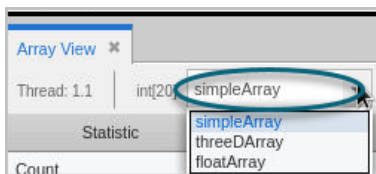
Depending on your settings, the Array View may open when you launch TotalView. To open the view, right-click on an array in either the Local Variables view or the Data View, and select **Add to Array View - Statistics, Visualize**.

Figure 71, Adding a variable to the Array View from the Local Variables view



Use the menu to add any number of arrays to the Array View.

If you add multiple arrays to the Array View, they appear in the Variables dropdown so you can select them at any point:



NOTE: Not all arrays are valid for statistics or visualizing, in which case, the menu item **Add to Array View - Statistics, Visualize** is not enabled. For example, pointers to allocated memory must be cast to an array before adding to the Array View.

The Array View Toolbar

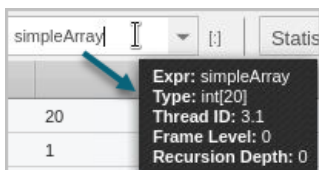
The Array View toolbar provides access to array statistics, visualization, and the ability to slice arrays. Its display defaults to a view of an array's statistics. See [Viewing Array Statistics](#).




The toolbar displays:

- The thread of focus
- Whatever type the array is set to. The type could reflect the original type of the array when it was added to the Array View or the edited type from the configuration dialog box.

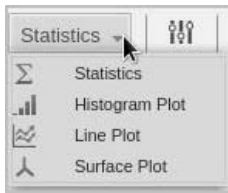
Hover over the array name in the toolbar array dropdown to get a tooltip with more information on the array:



- The array slice and stride if there is one.
By default, TotalView displays all elements in the array, for instance, `[:]` for a single-dimension array, `[:][:]` for a 2D array, and so on. If an array has been sliced, its slice will display in the toolbar.
- A dropdown to select either Statistics or a visualization tool.
- The Array Configuration Options toolbar button (), which opens the Array Configuration dialog where you can slice arrays. See [Configuring Arrays](#).

Array Statistics and Visualization

The **Statistics** dropdown of the Array View provides access to statistical information generated from the array values.



See

- [Viewing Array Statistics](#)
- [Visualizing Array Data](#)

Viewing Array Statistics

Figure 72, The Array Statistics view

Statistic	Value
Count	20000
Zero Count	0
Sum	-359978000.448795
Minimum	-37997.8984375
Maximum	2000.09997558594
Median	-17998.900390625
Mean	-17998.9000224398
Standard Deviation	11547.2938133416
First Quartile	-27998.900390625
Third Quartile	-7998.89990234375
Lower Adjacent Value	-37997.8984375
Upper Adjacent Value	2000.09997558594
Nan Count	0

The above is a one-dimensional floating point array composed of 20,000 elements, identified under the Count statistic. See [Array Statistics Detail](#) for information on all displayed statistics.

Array statistics are also available through the CLI, as switches to the **dprint** command.

RELATED TOPICS

[Cast a variable to an array](#)

[Casting to an Array in the Data View](#)

RELATED TOPICS

Slicing array data

[Slicing Arrays](#)

Displaying arrays

[Displaying Arrays](#)***Array Statistics Detail***

If you have added a slice (see [Slicing Arrays](#)), these statistics describe only the information currently being displayed; they do not describe the entire array. For example, if an array includes positive values, but a slice omits array values that are more than 0, the median value is negative even though the entire array's real median value is more than 0.

■ Count

The total number of displayed array values. If you're displaying a floating-point array, this number doesn't include NaN or Infinity values.

■ Zero Count

The number of elements whose value is 0.

■ Sum

The sum of all the displayed array's values.

■ Minimum

The smallest array value.

■ Maximum

The largest array value.

■ Median

The middle value. Half of the array's values are less than the median, and half are greater than the median.

■ Mean

The average value of array elements.

■ Standard Deviation

The standard deviation for the array's values.

■ Quartiles, First and Third

Either the 25th or 75th percentile values. The first quartile value means that 25% of the array's values are less than this value and 75% are greater than this value. In contrast, the third quartile value means that 75% of the array's values are less than this value and 25% are greater.

■ Lower Adjacent Value

This value provides an estimate of the lower limit of the distribution. Values below this limit are called *outliers*. The lower adjacent value is the first quartile value minus the value of 1.5 times the difference between the first and third quartiles.

- **Upper Adjacent Value**

This value provides an estimate of the upper limit of the distribution. Values above this limit are called *outliers*. The upper adjacent value is the third quartile value plus the value of 1.5 times the difference between the first and third quartiles.

- **Denormalized Count**

A count of the number of denormalized values found in a floating-point array. This includes both negative and positive denormalized values as defined in the IEEE floating-point standard. Unlike other floating-point statistics, these elements participate in the statistical calculations.

- **Infinity Count**

A count of the number of infinity values found in a floating-point array. This includes both negative and positive infinity as defined in the IEEE floating-point standard. These elements do not participate in statistical calculations.

- **NaN Count**

A count of the number of NaN (not a number) values found in a floating-point array. This includes both signaling and quiet NaNs as defined in the IEEE floating-point standard. These elements do not participate in statistical calculations.

- **Checksum**

A checksum value for the array elements.

Visualizing Array Data

To visualize your data, choose either Histogram Plot, Line Plot, or Surface Plot from the Statistics dropdown.

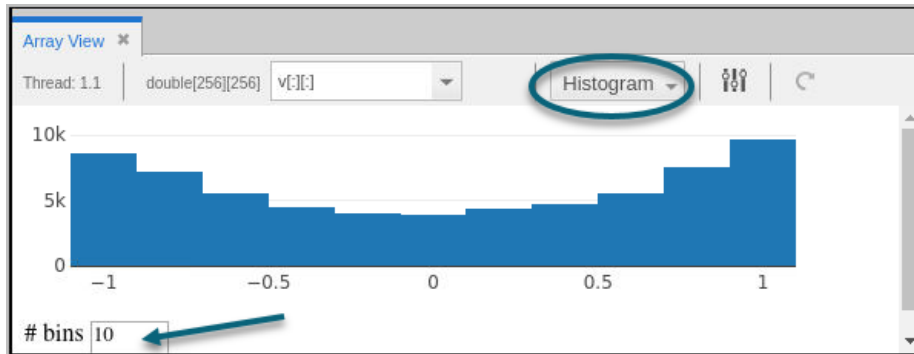
Different datasets can require different views to display their data. For example, you could use a histogram to see the distribution of a dataset, or lines and surface plots to view trends or slope.

The examples here display all the data for an array. To display a subset, you can slice the data. See [Slicing Arrays](#).

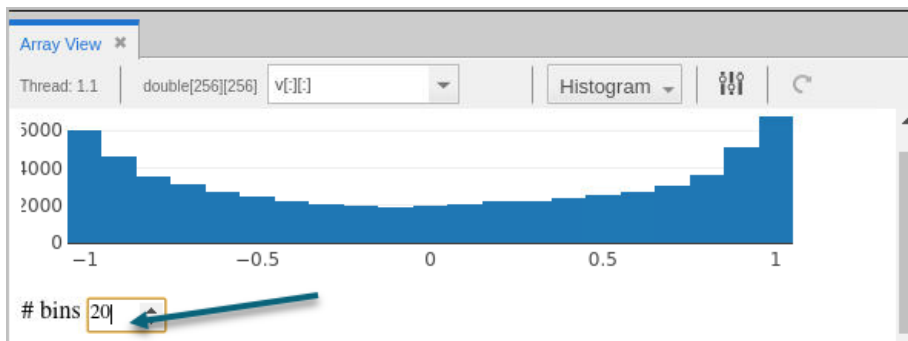
Histogram View

By default, the view displays 10 bins, or buckets:

Figure 73, Array View > Histogram



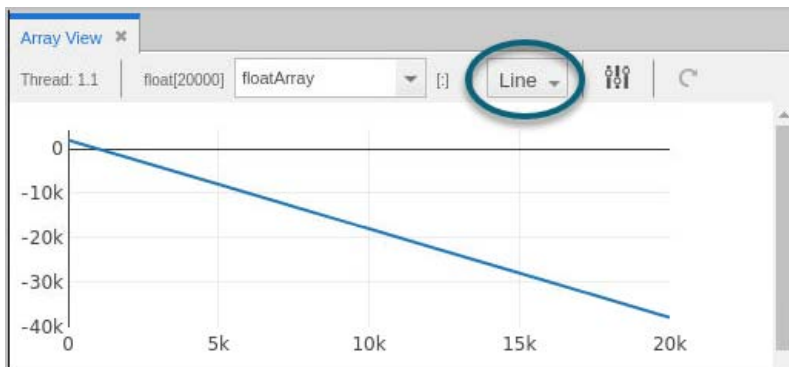
Change the number of bins to evaluate a different dataset distribution:



Line Plot View

To display your data as x and y coordinate pairs, use the **Line Plot** view. This view is useful to plot trend lines in your one-dimensional datasets. For example:

Figure 74, Array Data > Line Plot



Note that for higher-dimensional datasets in the Array View, the Line Plot displays a flattened, one-dimensional dataset. For example, given an array like this:

```
[[31 12 43][42 1 16
[0 42 0]]
```

The Line Plot displays a flattened array, like so:

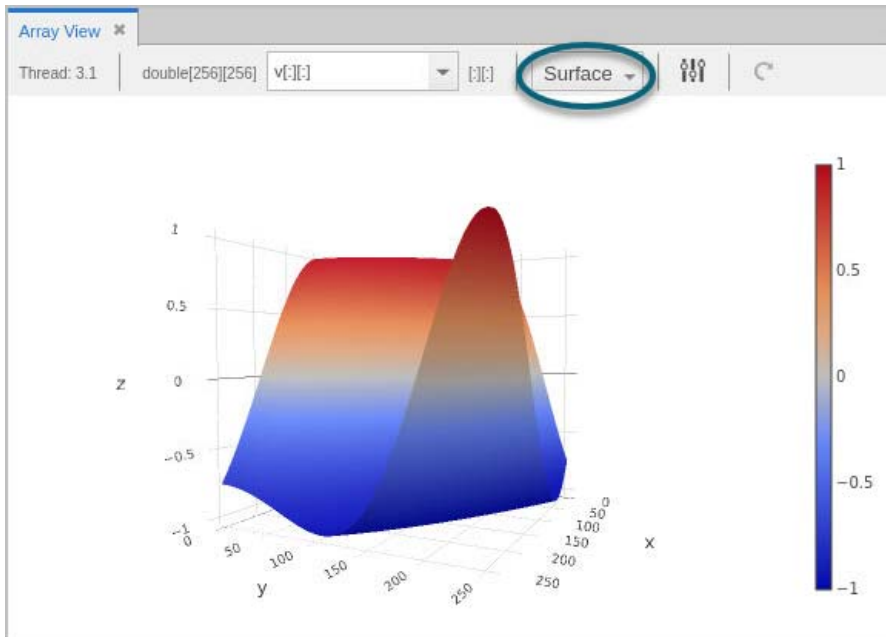
```
[31 12 43 42 1 16 0 42 0]
```

Surface Plot View

The Surface plot displays two-dimensional datasets as a surface in two or three dimensions. The dataset's array indices map to the first two dimensions (X and Y axes) of the display, and the values map to the height (Z axis). This can be useful to show a relationship across three variables and to observe trends in two-dimensional datasets.

NOTE: Surface plot display is supported only on Linux-x86-64, Linux ARM64, and macOS, and requires OpenGL version 2.1 or greater.


Figure 75, Array View: Surface Plot View



Use the plot controls to rotate or zoom the display.



Updating the View

After advancing your program, the view does not update automatically. To refresh the display, click the **Update** button ().


Changing the Thread of Focus

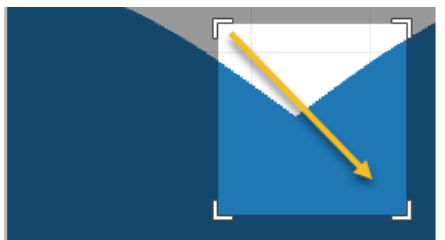
if you change the program's thread of focus, it's not reflected in the array displayed in the Array View, which displays the original thread of focus when the array was added to the view. You can, however, maintain multiple arrays in the Array View that are tied to different threads of focus.

Zooming Into Data

To view some data in detail, use the zoom toolbar, which displays when you place your cursor inside the graph:

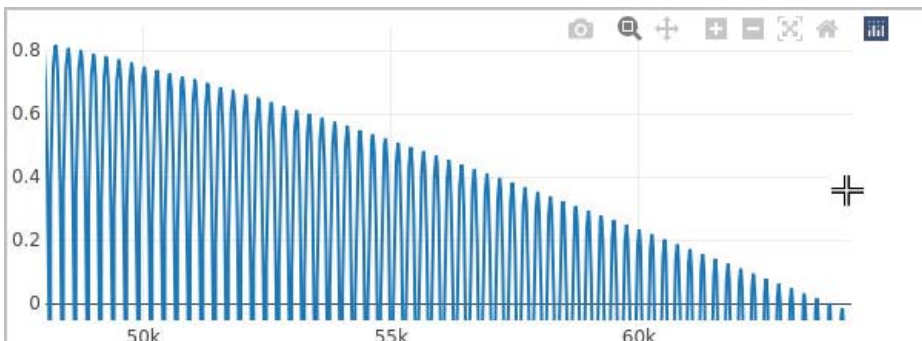




- **To zoom in**, either:
 - Use the Plus button ().
 - Drag an area to view:



TotalView zooms in on your data:

Figure 76, Array View > zooming in on data



To undo the zoom, either double-click on the graph or select the Reset Axes home button () or the Zoom out button ().

If you know the indices you want to examine, you can also slice the array to view a subsection; see [Slicing Arrays](#).

RELATED TOPICS

Slicing arrays to visualize or see statistics for a subsection of an array [Slicing Arrays](#)

Viewing arrays in the Data View [Working with Complex Variables in the Data View](#)

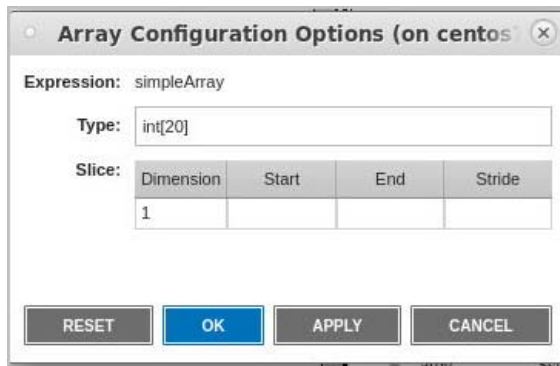
Displaying arrays [Displaying Arrays](#)

Configuring Arrays

Use the Array Configuration Options dialog to isolate and view a smaller portion of data by either slicing the array or adding strides as well as editing the array's type.

To open this dialog, select the Array Configuration button () from the Array View.

Figure 77, Array Configuration Options dialog

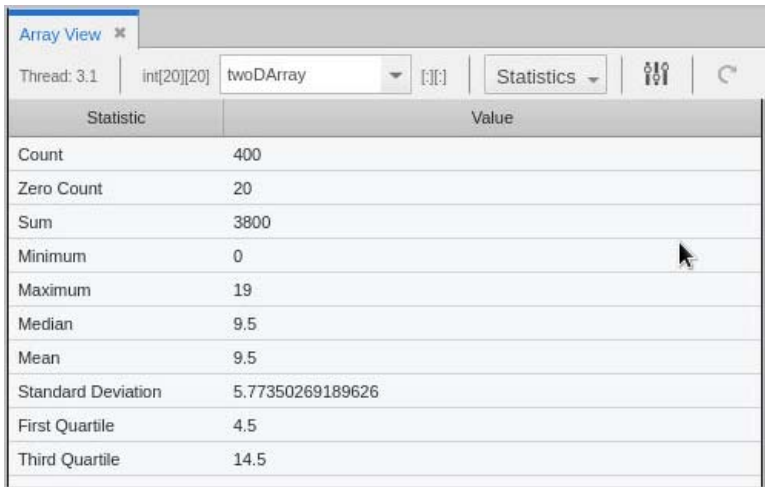


Here, enter a slice and/or stride to view a portion of an array.

Selecting different arrays in the Array View automatically updates the display in the Array Configuration Options dialog. The table in the dialog also updates to reflect the number of dimensions in the array.

Slicing Arrays

Consider a two-dimensional array containing 400 elements.



The screenshot shows the 'Array View' window with the following statistics:

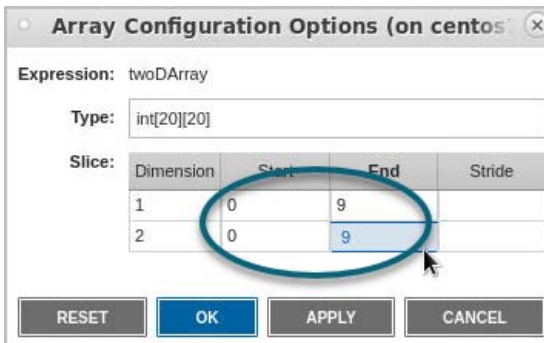
Statistic	Value
Count	400
Zero Count	20
Sum	3800
Minimum	0
Maximum	19
Median	9.5
Mean	9.5
Standard Deviation	5.77350269189626
First Quartile	4.5
Third Quartile	14.5

Slice the array to visualize or generate statistics on a sub-portion. To slice, open the Array Configuration Options dialog.

The general form for a slice definition is:

lower_bound:upper_bound[:stride]

For example, for a 2D array, **[0:9][0:9]** would use only elements 0 through 9 of each dimension, cutting the number of elements used to calculate the statistics to 100. Enter these values into the **Start** and **End** fields:

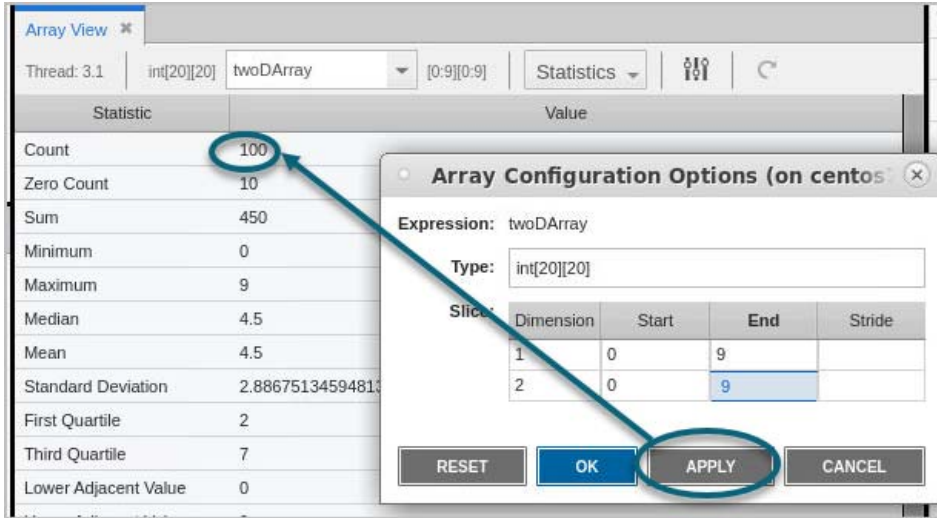


The screenshot shows the 'Array Configuration Options' dialog box. The 'Expression' is 'twoDArray' and the 'Type' is 'int[20][20]'. The 'Slice' section contains a table with the following values:

Dimension	Start	End	Stride
1	0	9	
2	0	9	

The 'Start' and 'End' fields for both dimensions are circled in blue. The dialog also includes 'RESET', 'OK', 'APPLY', and 'CANCEL' buttons.

Click **Apply** to immediately view your changes in the Array View.



NOTE: Clicking **Apply** applies your changes, but keeps the dialog box open so you can continue to test different slices. Click **OK** to apply your changes and close the dialog box. Select **Reset** to cancel your changes.

Adding a Stride

The default value for stride is 1; that is, show *all elements* in the range. If a stride of 2 were added to one of the dimensions, the statistical calculations would use every other element, cutting it to 50.

Slicing or adding a stride to an array is reflected in both the Statistics view and the plot views:

Figure 78, Sliced arrays display in all views

Statistics view

Statistic	Value
Count	50
Zero Count	10
Sum	200
Minimum	0
Maximum	8
Median	4

Plot Line view

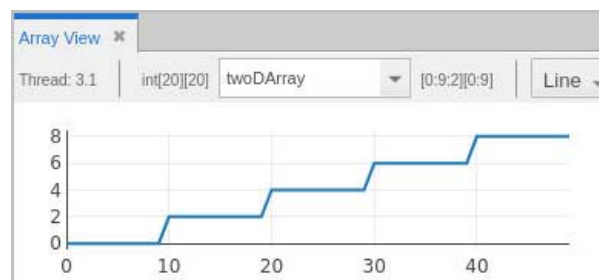
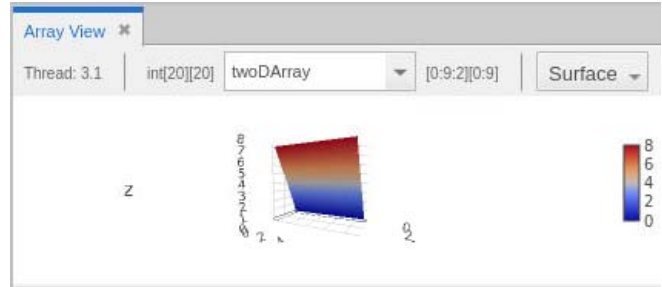


Figure 78, Sliced arrays display in all views

Histogram view



Surface Plot view



To view or analyze the stats of just the second dimension of the array, you could hold the first dimension steady, for example, `[0:0][:]`.

Statistic	Value
Count	20
Zero Count	20
Sum	0
Minimum	0
Maximum	0

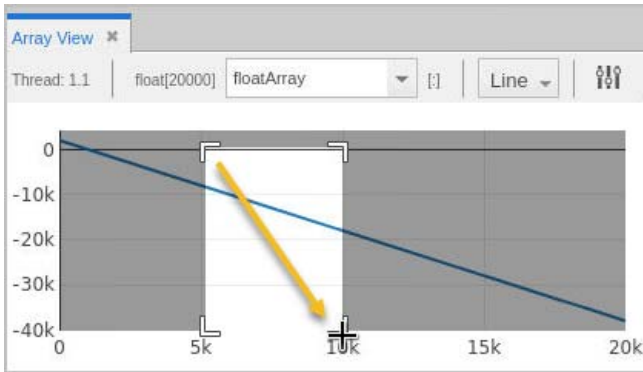
In this different 2D array, the slice includes all elements from the first dimension, but only the 10th element from the second dimension.

Statistic	Value
Count	62976
Zero Count	1
Sum	17205.7422474616
Minimum	-0.99999999723243
Maximum	0.999999997855555
Median	0.431621043897503
Mean	0.273211110022011
Standard Deviation	0.636059535641972
First Quartile	-0.216064885747958
Third Quartile	0.847091896573672
Lower Adjacent Value	-0.99999999723243

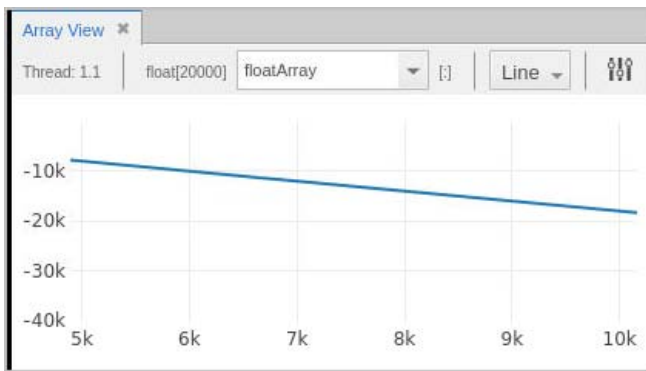
Slicing Using the Zoom Tool

You can also use the zoom tool to select an x-axis or y-axis range. Here is a 20,000-element, single-dimensional array in which we want to see just the values from 5,000 to 10,000.

First, select the slice by dragging:

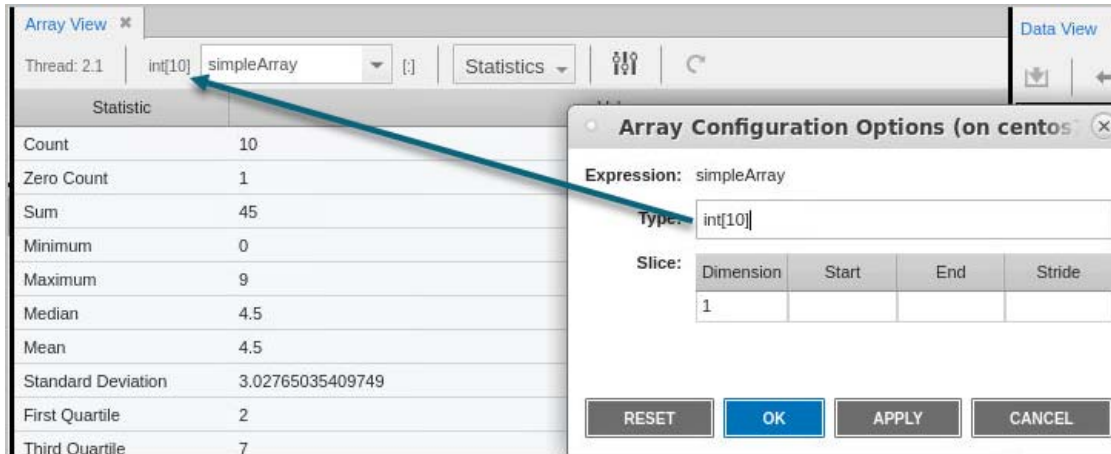


This slices the display, like so:

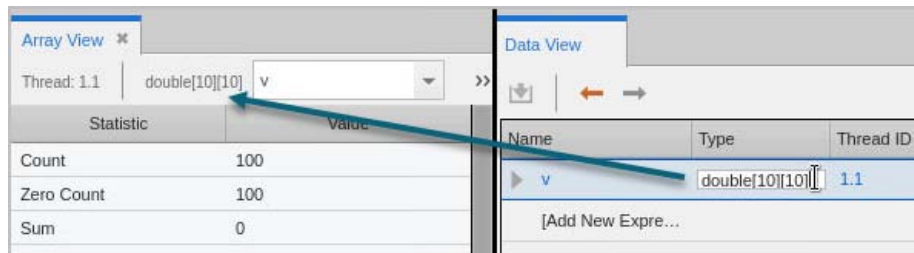


Casting to Another Type in the Array View

Use the Array Configuration Options dialog to cast your array to another type. Enter the new type in the Type field. The new type is immediately reflected in the Array View.



If you have cast the type to some other shape array in the Local Variables view or Data View, the edited data is displayed in the Array View. For example, editing this array in the Data View is then reflected when the array is added to the Array View:



C++ STL Type Transformations

While the C++ STL (Standard Template Library) offers standard template classes and simplifies data access, useful analysis of variables when debugging can be next to impossible, because most debuggers rely on the definitions of the data used by your compiler. In particular for STL, which uses abstractions such as structures, classes, and typedefs, much of the compiler-generated information provides little value for analysis.

To solve this problem, TotalView by default transforms STL types into a logical aggregated list, array, or structure view that makes it easier to examine the contents of STL objects. Transformed STL types include strings, arrays, vectors, dequeues, lists, ordered and unordered maps, multimaps, sets, and multisets, along with other objects.

STL type transformation is part of the TotalView [type transformation facility \(TTF\)](#) that provides tools for customizing how to view data.

NOTE: The TTF supports not only STL object transformation, but also the ability to create custom transformations. See [Creating Type Transformations](#) in the *Total View Reference Guide*.

STL type transformation supports specific compilers, compiler versions, and STL objects. See [Supported C++ STL Type Transformations](#) for specifics.

For example, [Figure 79](#) shows an untransformed `std::map<int, int>` compiled using the GNU C++ compiler (gcc), while [Figure 80](#) shows the same object after being transformed, in which the elements of the map are clearly visible.

NOTE: Transformed and untransformed objects display similarly in both the Local Variables view and the Data View, so you don't need to add the object to the Data View to see its transformed type.

Figure 79, An untransformed `std::map<int,int>` object

Name	Type	Value	Address
▼ m1	class std::map<int,int,std::less<int>,std::alloca...	(class std::map<int,int,std::less<int>,std::allo...	0x7ffd...
▼ _M_t	std::map<int,int,std::less<int>,std::allocator<st...	(std::map<int,int,std::less<int>,std::allocator<...	0x7ffd...
▼ _M_impl	struct std::_Rb_tree<int,std::pair<const int,int...	(struct std::_Rb_tree<int,std::pair<const int,i...	0x7ffd...
▼ all...	class std::allocator<std::_Rb_tree_node<std::...	(class std::allocator<std::_Rb_tree_node<st...	0x7ffd...
	class std::_new_allocator<std::_Rb_tree_no...	(class std::_new_allocator<std::_Rb_tree_n...	0x7ffd...
▼ _...	struct std::_Rb_tree_key_compare<std::less<i...	(struct std::_Rb_tree_key_compare<std::les...	0x7ffd...
▼	struct std::less<int>	(struct std::less<int>)	0x7ffd...
	struct std::binary_function<int,int,bool>	(struct std::binary_function<int,int,bool>)	0x7ffd...
▼ _...	struct std::_Rb_tree_header	(struct std::_Rb_tree_header)	0x7ffd...
▼	struct std::_Rb_tree_node_base	(struct std::_Rb_tree_node_base)	0x7ffd...
	enum std::_Rb_tree_color	_S_red (0)	0x7ffd...
▶	std::_Rb_tree_node_base::Base_ptr	0x01809180 -> (struct std::_Rb_tree_node ...	0x7ffd...

Figure 80, A transformed `std::map<int,int>` object

Name	Type	Value	Address
▼ m1	class std::map<int,...	(class std::map<int,int,std::less<int>,std::alloc...	0x7ffc60f58650
▼ 0	map_element	(Map_element)	0x0066beb0
Key	int	0x00000001 (1)	0x0066bed0
Value	int	0x00000001 (1)	0x0066bed4
▼ 1	map_element	(Map_element)	0x0066bee0
Key	int	0x00000002 (2)	0x0066bf00
Value	int	0x00000004 (4)	0x0066bf04
▶ 2	map_element	(Map_element)	0x0066bf10
▶ 3	map_element	(Map_element)	0x0066bf40
▶ 4	map_element	(Map_element)	0x0066bf70
▶ 5	map_element	(Map_element)	0x0066bfa0

Supported C++ STL Type Transformations

Table 4 lists the supported TotalView built-in C++ STL type transformations for the C++ library classes and iterators for the container classes.

Table 4: Supported C++ STL Type Transformations

Container Type	Class	Iterator Type	Transformed Type
Sequence	array (C++11)	Random	Array (Dense)
	vector	Random	Array (Dense)
	deque	Random	Array (Dense)
	forward_list (C++11)	Forward	List
	list	Bidirectional	List
Associative	set	Bidirectional	Tree
	multiset	Bidirectional	Tree
	map	Bidirectional	Tree
	multimap	Bidirectional	Tree
Unordered associative	unordered_set	Forward, Local	Hashtable
	unordered_multiset	Forward, Local	Hashtable
	unordered_map	Forward, Local	Hashtable
	unordered_multimap	Forward, Local	Hashtable
Adaptors	stack (deque,list,vector)	n/a	Struct
	queue (deque,list)	n/a	Struct
	priority_queue (deque,vector)	n/a	Struct
General utilities	pair	n/a	Struct
	tuple (C++11)	n/a	Struct
Memory management	unique_ptr (C++11)	n/a	Struct
	shared_ptr (C++11)	n/a	Struct
	weak_ptr (C++11)	n/a	Struct
Numeric	complex	n/a	Struct
Strings	string	n/a	Struct

Struct TTF “\$elide_” Members

Some STL classes are transformed to structures containing a member name with the prefix `$elide_`. By convention, STL TTFs use `$elide_str` for pointers to character strings and `$elide_ptr` for pointers to other types.

This prefix allows the debugger to display *only that member* in certain contexts.

In Figure 81, `lit_int` is a transformed local iterator for an integer.

Figure 81, Transformed iterator displaying its value

Name	Type	Value	Address
int_unordered_map	int_unordered_map_t	(int_unordered_map_t)	0x7ffc8e62f760
pod_unordered_map	pod_unordered_map_t	(pod_unordered_map_t)	0x7ffc8e62f720
it_int	std::unordered_map<int,int, std::hash<int>...>	0x0231dec8 -> (struct std::pair<const int,int>)	0x7ffc8e62f718
cit_int	std::unordered_map<int,int, std::hash<int>...>	0x0231dee8 -> (struct std::pair<const int,int> const)	0x7ffc8e62f710
lit_int	std::unordered_map<int,int, std::hash<int>...>	0x0231dec8 -> (struct std::pair<const int,int>)	0x7ffc8e62f6f0
clit_int	std::unordered_map<int,int, std::hash<int>...>	0x0231dee8 -> (struct std::p... 0x0231dec8 -> (struct std::pair<const int,int>)	
it_pod	std::unordered_map<int,pod_t, hash_pod...>	0x02323418 -> (struct std::pair<const int,pod_t>)	0x7ffc8e62f6c8

Note that the `$elide_ptr` member value is reported both in the top-level view and in a tooltip. There is no need to drill down into the structure’s contents because the value is already displayed.

To see the actual `$elide_ptr` member and other, perhaps less important members of a transformed structure, expand `lit_int` in Figure 82, to view the `bucket` and `bucket_count` members as well.

Figure 82, Transformed iterator with an \$elide_ptr prefix

Name	Type	Value	Address
lit_int	std::unordered_multiset<int, std::hash<int>, std::...>	0x023c7ec8 -> 0x00000066 (102)	0x7ffc0899...
\$elide_ptr	int const *	0x023c7ec8 -> 0x00000066 (102)	(None)
*(\$elide_ptr)	const int	0x00000066 (102)	0x023c7ec8
bucket	unsigned long	0x000000000000000b (11)	0x7ffc0899...
bucket_count	unsigned long	0x000000000000000d (13)	0x7ffc0899...

Iterator Type Transformations

Iterator transformation depends on the container class, i.e., according to the Standard, some containers support only forward iterators while some support random or bidirectional iterators. Adaptor classes are built on top of other containers, so they have no iterators.

- Random and bidirectional iterators include type transformations for the `iterator`, `const_iterator`, `reverse_iterator`, and `const_reverse_iterator` types for the container class.
- Forward iterators include type transformations for the `iterator` and `const_iterator` types for the container class.
- Local iterators apply to the unordered containers only and include type transformations for the `local_iterator` and `const_local_iterator` types for the container class.
- All iterators transform to a structure containing a member named `$elide_ptr` that is a pointer to the value type, and points to an element in the container. (See [Struct TTF "\\$elide_" Members](#).)
- Local iterator transformed types also contain `bucket` and `bucket_count` members, corresponding to element's bucket number and the number of buckets in the hashtable.
- The adaptor classes act as a wrapper to an underlying container class (deque, list, or vector), which is transformed according to the normal TTF rules.

TTFs and TotalView Expressions

The TotalView expression system *always* operates on the original (untransformed) type, never on the transformed type. Trying to evaluate an expression that uses the members of a transformed type directly will fail.

NOTE: Expressions operate only on the original, untransformed type, never the transformed type.

For example, this illustrates the difference between evaluating an expression that *dereferences* a deque reverse iterator (`*rit_int`) by calling `operator()*` for the object, and one that *displays* `rit_int` using the STL TTF rules.

```
d1.<> p *rit_int
*rit_int = 0x00000066 (102)

d1.<> p rit_int
rit_int = {
  $elide_ptr = 0x024440d8 -> 0x00000066 (102)
}
```

Here, however, the expression `rit_int.$elide_ptr` causes an expression evaluation error because the expression system knows nothing about the transformed type, and thus there is no member named `$elide_ptr` in the original, untransformed type.

```
d1.<> p {rit_int.$elide_ptr}
dprint: Error evaluating expression 'rit_int.$elide_ptr':
  Error line 1,column 9: Cannot find name "$elide_ptr"
```

Type Transformations CLI Commands

STL type transformations are enabled by default. You can turn them off in the CLI and can also query TotalView to see what types of transformations are supported.

Controlling Type Transformations

Toggle STL type transformations on and off using either the CLI Boolean state variable **TV::ttf** or the command line options **-ttf** | **-nottf**.

- To control TTF for this session and future sessions, use the CLI's **dset** command to set the **TV::ttf** variable in a TotalView start up file, for example, `~/ .tvdrc`:

```
dset TV::ttf false | true
```

NOTE: This variable does not persist between TotalView sessions unless placed in a TotalView startup file.

- To set TTF for just a single session, use the command line option when starting TotalView:

```
totalview -nottf | -ttf
```

When enabled, the UI and certain CLI commands (such as **dprint**) display the transformed type. When set to **false**, the original, untransformed type is used.

For example, consider a reverse iterator `rit_int`. Here is the untransformed variable:

```
d1.<> dwhat rit_int
In thread 1.1:
Name: rit_int; Type: std::deque<int, std::allocator<int> >::reverse_iterator; Size:
32 bytes; Addr: 0x7ffefad6ed30
  Scope: ##/etnus/scratch/home/jdelsign/tvbld/linux-x86-64/fedora36-x8664/
totalview.develop/debugger/src/tests/bld/gcc_12.2.1_64/tx_c++11_iterators#.../.../
src/tx_c++11_iterators.cxx#test_deque_iterators(void) (Scope class: Any)
  Address class: auto_var (Local variable)

d1.<> dset TV::ttf false
false
d1.<> dprint rit_int
rit_int = {
  iterator = {
```

```

    }
    current = {
        _M_cur = 0x015730dc -> 0x00000000 (0)
        _M_first = 0x015730d0 -> 0x00000064 (100)
        _M_last = 0x015732d0 -> 0x00000000 (0)
        _M_node = 0x01573098 -> 0x015730d0 -> 0x00000064 (100)
    }
}

```

And here is the transformed version after setting the state variable **TV::ttf** to true:

```

d1.<> dset TV::ttf true
true
d1.<> dprint rit_int
rit_int = {
    $elide_ptr = 0x015730d8 -> 0x00000066 (102)
}
d1.<>

```

Querying Type Transformations

Use the **TV::type_transformation** command to return a list of supported type transformations.

For example, the following command finds the type transformations registered with a description matching **GNU std::** and returns the name and **validate_callback** property for **GNU std::deque** type transformation:

```

d1.<> exec <<[join [TV::type_transformation get -all id type_transformation_descrip-
tion] "\n"] grep "GNU std::"
1 | 26 {GNU std::deque<>}
1 | 38 {GNU std::tuple}
1 | 39 {GNU std::array}
1 | 40 {GNU std::initializer_list}
1 | 41 {GNU std::reverse_iterator}
1 | 42 {GNU std::vector<>::iterator}
1 | 43 {GNU std::deque<>::iterator}
1 | 44 {GNU std::list<>::iterator}
1 | 45 {GNU std::{multi,}{map,set}<>::iterator}
1 | 46 {GNU std::unordered_{multi,}{map,set}<>::iterator}
1 | 47 {GNU std::unordered_{multi,}{map,set}<>::local_iterator}
d1.<>

```

STL TTF Troubleshooting

If STL objects are not properly being transformed, it is usually due to either the way a program was compiled or the version of the STL you are using.

The Compiler Discarded Type Information

The compiler may discard the required debug information, which it will do if the type information is not used by the program itself or the compiler is designed to assume the type information is defined in another module. If this happens, compiling the program with one or more of the following flags may help:

- **-fno-eliminate-unused-debug-types:** Use this flag with both GNU g++ and Clang++ compilers to ensure unused types are not discarded.
- **-fstandalone-debug:** Use this flag with Clang++ compilers to emit full debug information for all types used by the program.

Consult the compiler documentation for more information and additional flags.

The STL Implementation Changed

The STL implementation may have changed and the STL validation procedure does not handle the new class layout. It is common for STL library implementations to change the layout of the C++ STL template classes, so older versions of TotalView may not work with newer versions of an STL library.

Contact TotalView Support to request support for newer versions of an STL library.

Using the CLI to Examine Data

To access all the functionality of Classic TotalView, you can use the CLI.

NOTE: For this release of TotalView, using functionality in the CLI that is not present in the UI does not update the UI.

Changing the Display of Data

Viewing STL Datatypes

By default, TotalView transforms STL types. See [C++ STL Type Transformations](#).

Changing Size and Precision

You can change the format that TotalView uses to display a variable's value using one of a series of `TV::data_format` variables that control the precision for simple data types.

For example, you can set how many character positions a value uses when TotalView displays it and how many numbers to display to the right of the decimal place. You can also customize how to align the value and if numbers should be padded with zeros or spaces.

CLI: To obtain a list of variables that you can set, type `"dset TV::data_format*"`.

RELATED TOPICS

Data format CLI variables

A list of the TotalView data format variables in the *TotalView Reference Guide*

Displaying Variables

Displaying Program Variables

Display local and global variables using **dprint**:

CLI: `dprint variable`

This command lets you view variables and expressions without having to select or find them.

For example, **dprint j** returns the value of **j**:

```
j = 0x00000005 (5)
```

CLI: `dwhere`, `dup`, and `dprint`

Use **dwhere** to locate the stack frame, use **dup** to move to it, and then use **dprint** to display the value.

Dereferencing Variables Automatically

In most cases, you want to see what a pointer points to, rather than the value of its variable. Use the CLI to automatically dereference pointers.

Dereferencing pointers is especially useful when you want to visualize the data linked together with pointers, since it can present the data as a unified array. Because the data appears as a unified array, you can use TotalView/array manipulation commands to view the data.

CLI: `TV::auto_array_cast_bounds`

`TV::auto_deref_in_all_c`

`TV::auto_deref_in_all_fortran`

`TV::auto_deref_initial_c`

`TV::auto_deref_initial_fortran`

`TV::auto_deref_nested_c`

`TV::auto_deref_nested_fortran`

Automatic dereferencing can occur in the following situations:

- When TotalView initially displays a value.
- When you dive on a value in an aggregate or structure.

Displaying Areas of Memory

You can display areas of memory using hexadecimal, octal, or decimal values:

- **An address**
- **A pair of addresses**

All octal constants must begin with 0 (zero). Hexadecimal constants must begin with 0x.

The Processes & Threads View

The Processes & Threads view displays detail about all processes and threads in your debugging session, allowing you to organize them into aggregate groupings based on attributes. This chapter focuses on the view in the UI and how it updates and displays, depending on focus and program execution. For a deeper dive into debugging multithreaded, multi-process programs, see [Part III, Parallel Debugging](#).

This chapter includes:

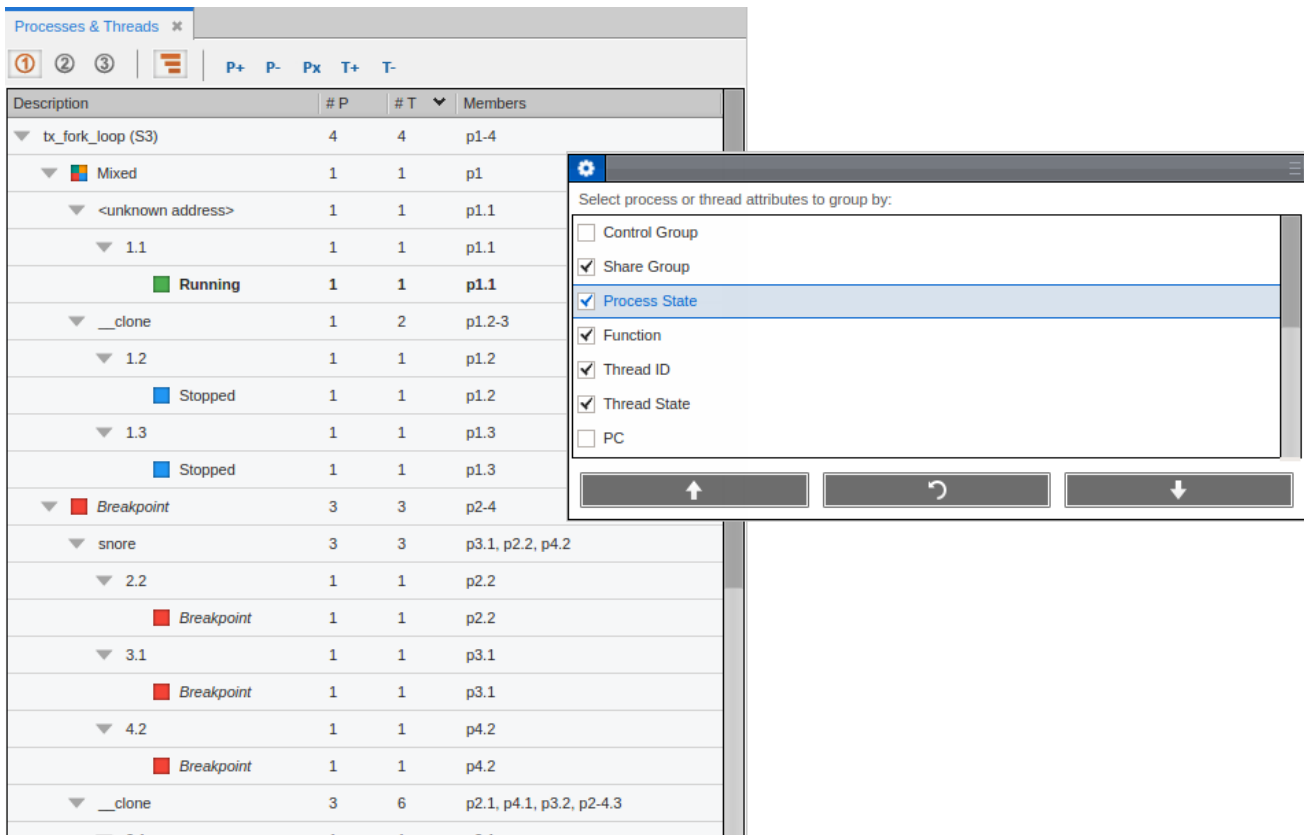
- [Processes & Threads View Basics](#)
- [Customize the Display](#)
- [The Processes & Threads View in Relation to Other Views](#)
- [Displaying a Thread Name](#)
- [Process and Thread Attributes](#)

Processes & Threads View Basics

In Figure 83, the processes and threads are grouped by share group, process state, function, thread ID, and thread state.

- There are four processes and a total of 12 threads (not all visible).
- The share group is the set of processes executing the same program, and the executable program name is indicated in its attribute.
- The focus is on thread *1.1*, thread 1 of process 1, as indicated by its **bold** format.
- The thread of focus determines the display in the Call Stack, Local Variables (VAR) view, the Source view, and the Data view.

Figure 83, Processes & Threads Tree View



Note that the status of processes and threads is highlighted by colored icons for easy identification. The “Mixed” icon identifies a process whose threads are in different states.

RELATED TOPICS

Controlling how action points works with processes and threads [Action Points](#) in the [Preferences](#) chapter

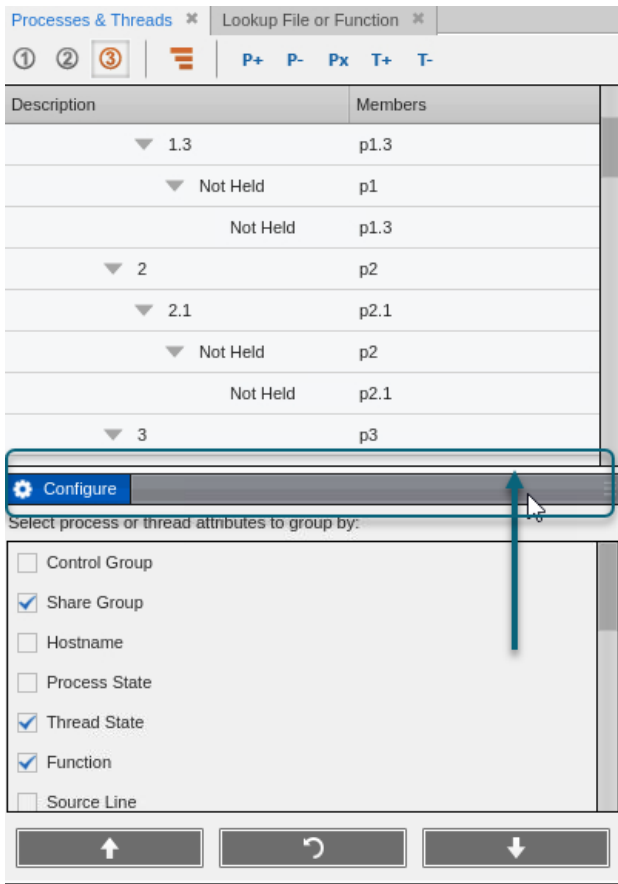
Action Point Properties for controlling whether it stops a group of processes, a single process (which includes all its threads), or a single thread. [Controlling an Action Point's Width](#)

Customize the Display

Use the *Configure* panel to “group by” selected attributes

The **Configure** panel displays process and thread attributes by which you can group the display.

If the panel is not visible, drag the panel header to open it.



Checked selections appear in the View pane in the order that they appear in the “Group by” list. To change the order, select items in the list and use the up and down arrows. **Reset** restores the initial order. To hide the list, double-click its banner.

The Processes & Threads View Layout

- The **Description** column shows the aggregate groupings that are active for the current program state.

- The columns **#P** and **#T** show the number of processes and threads with a given set of attributes. For example, three threads from different processes are stopped at a breakpoint in the `snore` function.
 - The **Members** column summarizes the processes and threads in **ptlist** format. For example, for the `snore` function, `p3.1`, `p2.2`, `p4.2` indicates that *thread 1 of process 3* and *thread 2 of processes 2 and 4* are stopped at this function.
- For more information on **ptlists**, see "[Compressed List Syntax \(ptlist\)](#)" in the *TotalView Reference Guide*.

Customizing the Processes & Threads View Pane

To control which columns display, right-click in the banner and select column names to display or hide.

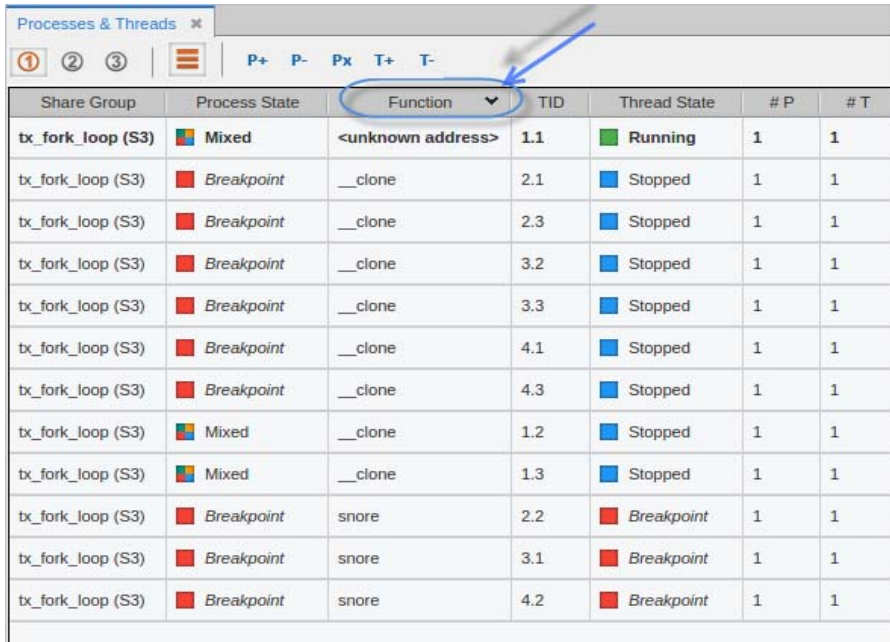
To view either a tree or table representation of process and thread state, click the view toggle icon for either a tree view (📁) or a tabular view (📄):

Figure 84, Processes & Threads Tabular View

Share Group	Process State	Function	TID	Thread State	# P	# T	Members
tx_fork_loop (S3)	Breakpoint	__clone	2.1	Stopped	1	1	p2.1
tx_fork_loop (S3)	Breakpoint	__clone	2.3	Stopped	1	1	p2.3
tx_fork_loop (S3)	Breakpoint	__clone	3.2	Stopped	1	1	p3.2
tx_fork_loop (S3)	Breakpoint	__clone	3.3	Stopped	1	1	p3.3
tx_fork_loop (S3)	Breakpoint	__clone	4.1	Stopped	1	1	p4.1
tx_fork_loop (S3)	Breakpoint	__clone	4.3	Stopped	1	1	p4.3
tx_fork_loop (S3)	Breakpoint	snore	2.2	Breakpoint	1	1	p2.2
tx_fork_loop (S3)	Breakpoint	snore	3.1	Breakpoint	1	1	p3.1
tx_fork_loop (S3)	Breakpoint	snore	4.2	Breakpoint	1	1	p4.2
tx_fork_loop (S3)	Mixed	<unknown address>	1.1	Running	1	1	p1.1
tx_fork_loop (S3)	Mixed	__clone	1.2	Stopped	1	1	p1.2
tx_fork_loop (S3)	Mixed	__clone	1.3	Stopped	1	1	p1.3

To sort the display, click on any of the column headers in either the tree or tabular view. To change the sort order between descending and ascending, click the column header again.

Figure 85, Reordering the Processes & Threads View



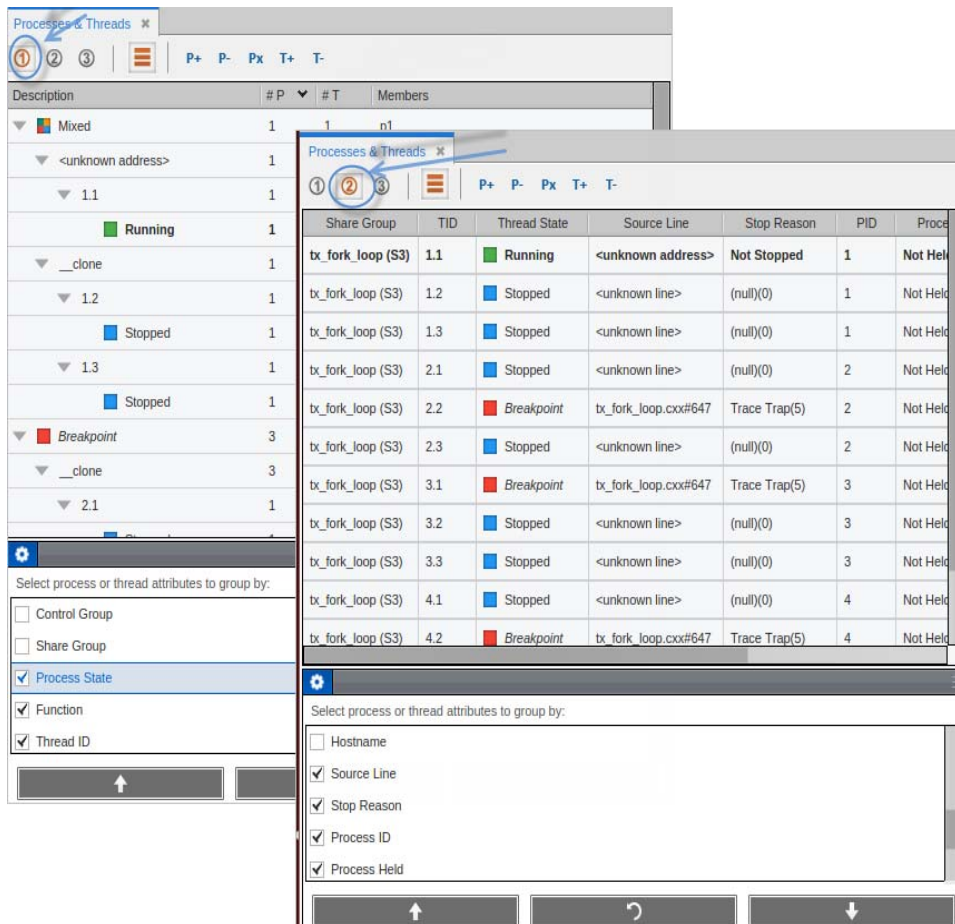
The screenshot shows the 'Processes & Threads' window with a table of threads. The table is sorted by the 'Function' column, as indicated by a blue arrow pointing to the 'Function' header. The table contains 12 rows of thread data.

Share Group	Process State	Function	TID	Thread State	# P	# T
tx_fork_loop (S3)	Mixed	<unknown address>	1.1	Running	1	1
tx_fork_loop (S3)	Breakpoint	__clone	2.1	Stopped	1	1
tx_fork_loop (S3)	Breakpoint	__clone	2.3	Stopped	1	1
tx_fork_loop (S3)	Breakpoint	__clone	3.2	Stopped	1	1
tx_fork_loop (S3)	Breakpoint	__clone	3.3	Stopped	1	1
tx_fork_loop (S3)	Breakpoint	__clone	4.1	Stopped	1	1
tx_fork_loop (S3)	Breakpoint	__clone	4.3	Stopped	1	1
tx_fork_loop (S3)	Mixed	__clone	1.2	Stopped	1	1
tx_fork_loop (S3)	Mixed	__clone	1.3	Stopped	1	1
tx_fork_loop (S3)	Breakpoint	snore	2.2	Breakpoint	1	1
tx_fork_loop (S3)	Breakpoint	snore	3.1	Breakpoint	1	1
tx_fork_loop (S3)	Breakpoint	snore	4.2	Breakpoint	1	1

Here, the table has been sorted by **Function**, identified by the down arrow in the Function column header.

To customize the view itself, use the three View icons (① ② ③). Each view is independent and can have any combination and order of attributes. This provides a convenient way to have several different views into your program.

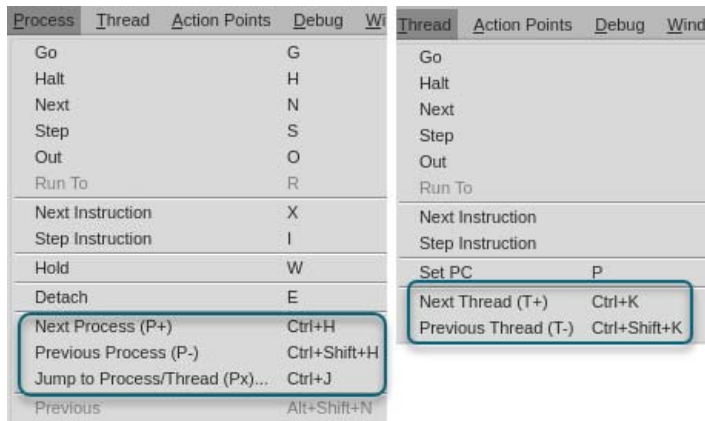
Figure 86, Processes & Threads – Viewing Different Attributes



Cycling Through Threads and Processes

The Processes & Threads view displays a maximum of 100 items, while some complex, multithreaded programs may have thousands of processes and threads. Those that are not displayed are shown as group "...". Threads that are not displayed cannot be assigned the focus.

In cases like these, use the **P+**, **P-**, **T+**, and **T-** buttons on the toolbar (| P+ P- Px T+ T-) to cycle through processes in the current control group or threads within the process in focus. Use these buttons to assign a focus and display any process or thread. The **Process** and **Thread** menus also include these options:



P+ and **P-** loop through all processes. For example, if a program has 200 processes and the process in focus is #200, **P+** moves the focus to process #1; if the current process in focus is #1, **P-** focuses on process #200. The same is true for **T+** and **T-**, which cycle through threads.

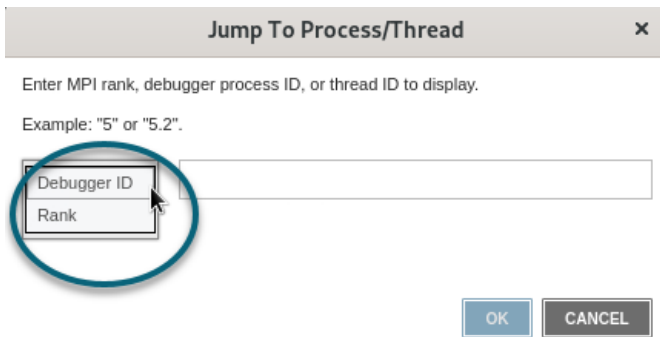
Use **Px** to jump to a specific process or thread instead of looping through one by one.

Table 5: P+/P-/T+/T- cycling buttons

Button	Description	Keyboard Shortcut
P+	Cycles to the first thread of the next process within a focus group.	Ctrl+H
P-	Cycles to the first thread of the previous process within a focus group.	Ctrl+Shift+H
T+	Cycles to the next thread within a focus group.	Ctrl+K
T-	Cycles to the previous thread within a focus group.	Ctrl+Shift+K
Px	Opens a Jump To Process/Thread dialog box.	Ctrl+J

Jump to a specific process or thread

Click **Px** to open a **Jump To** dialog box where you can specify a specific process or thread to focus on.



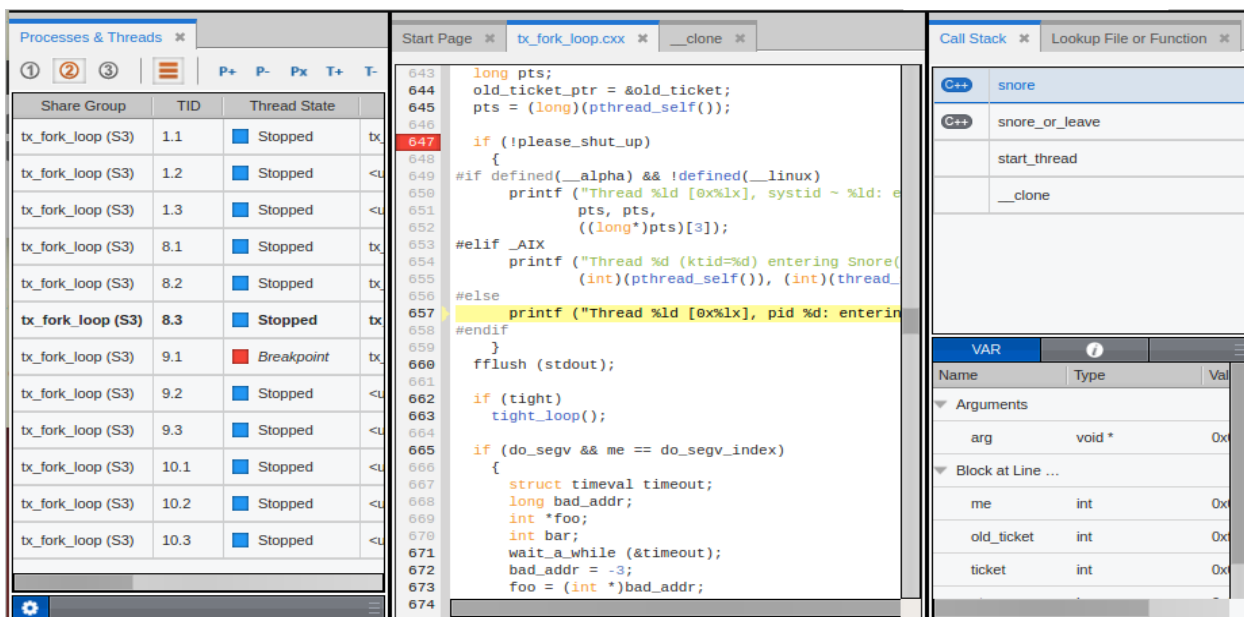
Choose either a debugger ID or rank:

- **Debugger ID:** Either the process or thread ID, for instance, **3** or **3.2**. If you enter only **3**, the focus moves to the first thread in the process, or **3.1**.
- **Rank:** For MPI programs, the rank, or unique identifier of the process, from 0 to *n*.

The Processes & Threads View in Relation to Other Views

The Processes & Threads view displays the state of the processes under debugger control at any given moment. The view updates each time one or more processes moves to some other part of the code. If you have set breakpoints, often some number of threads will be stopped at one, but even if all threads are running, this state will be shown in the view.

Figure 87, Relationships of Processes & Threads, Source, Call Stack, and Data Views



If you are displaying the Thread IDs, one of the lines is highlighted in bold (Thread ID 8.3 above). That line determines the display in the Source view, as well as the Call Stack, Local Variable (VAR), and Data views.

If you are not displaying individual thread IDs, the line representing the aggregate that contains the current thread of focus is bold. If you double-click on another line, the display changes to represent the source location, call stack, and data values pertinent to that line.

Displaying a Thread Name

In complex, multithreaded programs with perhaps thousands of threads, it may be useful to name certain threads, for instance, if particular threads are dedicated to performing special functions. This can be helpful when sorting or identifying threads in your programs.

If you set a thread name in your program, the name is displayed in the TotalView UI:

- In the Processes & Threads view (if Thread Name is selected in the **Group By** pane)
- In TotalView's title bar

To display a thread name in the TotalView UI, first set the name in your program using the `pthread_setname_np()` method.

For example:

```
int rc = pthread_setname_np(thread, "MyThreadName");
```

Unless explicitly set by the program, threads are not named.

RELATED TOPICS

[TV::thread](#) properties

[TV::thread](#)

Thread names displayed in the UI

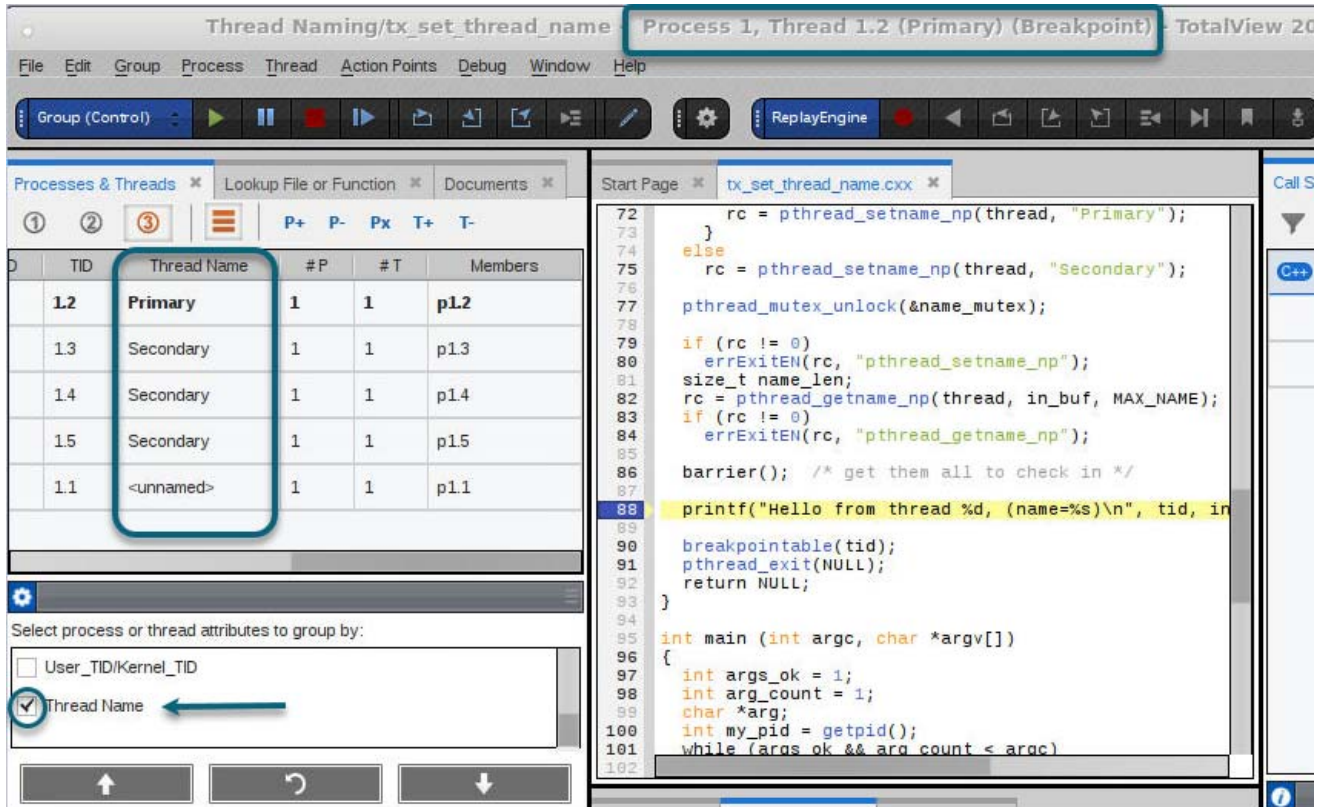
[Thread Names in the UI](#)

[dstatus](#) options relating to thread names [dstatus](#)

Thread Names in the UI

When set, thread names are displayed in both the title bar and the Processes & Threads view, when **Thread Name** is selected in the **Group by** pane:

Figure 88, Thread names in the UI



This program sets these thread names:

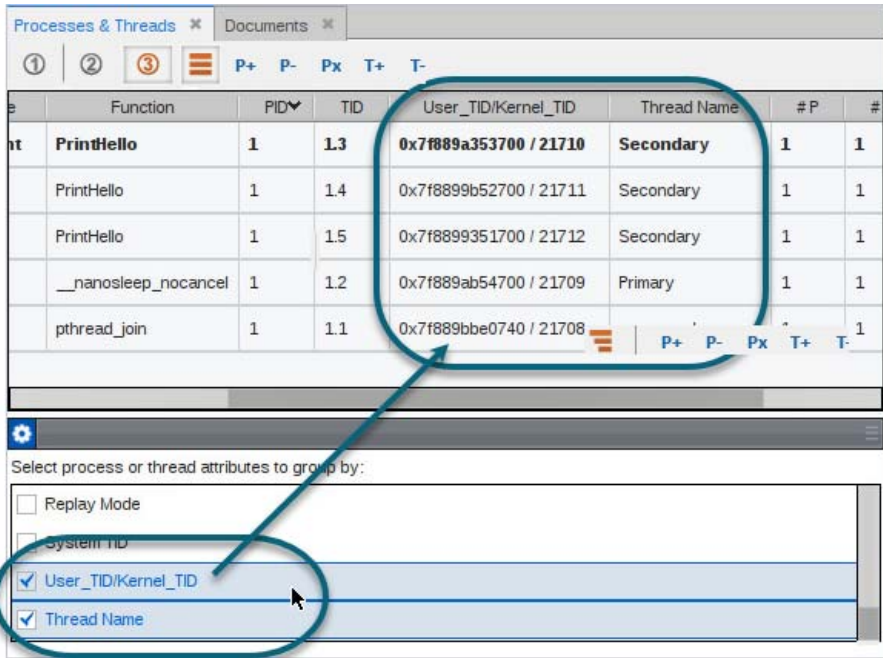
```

pthread_mutex_lock(&name_mutex);
int rc;
if (!first_in)
{
    first_in = true;
    rc = pthread_setname_np(thread, "Primary");
}
else
    rc = pthread_setname_np(thread, "Secondary");

```

The first thread to enter a function is named "Primary", and all subsequent threads are called "Secondary."

Also relevant to thread names are three other properties, the **systid** (the target system thread ID) and the **utid / ktid** (“thread user ID” / “thread kernel ID”). You can display these in the Processes & Threads view by selecting them in the **Group by** pane:



For more information, see [Thread Properties](#) and [Process and Thread Attributes](#).

Thread Properties

TV::thread includes these properties relevant to thread naming:

- **thread_name**: The name given to a thread by the application.
- **thread_ktid**: The kernel thread id
- **thread_utid**: User thread ID (**pthread_t**)

These properties are read-only, so can be accessed but not set.

Thread Options on dstatus

The command **dstatus** has options and properties related to thread names:

- **-thread_name**: Displays any thread names in a program, if they exist
- Properties on the **-group_by** option:

- **systid**: Either the user thread ID (**utid**) or the kernel thread ID (**ktid**) if no **utid** exists
- **utid_ktid**: "**utid / ktid**" or just **ktid** if no **utid** exists.
- **tname**: thread name or "<unnamed>" if no thread name exists.

Process and Thread Attributes



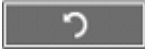
Table 6: Processes and Threads Attribute Descriptions

Property	Applies to	Description
Control Group	Processes	Control group of the processes in your program. Processes in the same program are placed in the same control group by default. If there is only one control group in the debug session, this property is omitted from the display.
Share Group	Processes	Share group of the processes within a control group. Processes that are running the same main executable are placed in the same share group by default.
Hostname	Processes	The hostname or IP address of where the process is running.
Process State	Processes	The process execution state, e.g., Nonexistent, Running, Stopped, Breakpoint, Watchpoint, etc. The process execution state derives from the execution state of the threads it contains.
Thread State	Threads	The thread execution state, e.g., Running, Stopped, Breakpoint, Watchpoint, etc.
Function	Threads	The function name of the location of the stopped thread. Displays the function name or "<unknown address>" if the thread is running or the function name is not known.
Source Line	Threads	The source code line of the location of the stopped thread. Displays the source file name and line number, or "<unknown line>" if the thread is running or the source line is not known.
PC	Threads	The program counter of the location of the stopped thread. Displays the program counter value, or "<unknown address>" if the thread is running.
Action Point ID	Threads	The action point ID (breakpoint or watchpoint) of the location of the stopped thread. Displays "ap(<i>id</i>)", where <i>id</i> is the action point ID, or "none" if the thread is not stopped at an action point.
Stop Reason	Threads	More detailed information on why the process/thread has stopped.
Process ID	Processes	The debugger process ID (dpid) of the process. Displays dpid .
Thread ID	Threads	The dpid and debugger thread ID (dtid) of the thread. Displays dpid.dtid .
Thread Index	Threads	The thread index part of the Thread ID. For example, if the Thread ID is 3.2, the Thread Index is 0.2.
Process Held	Processes	The 'hold' state of the process. This may be because of an explicit hold request, or because the process is waiting at a barrierpoint.

Table 6: Processes and Threads Attribute Descriptions

Property	Applies to	Description
Thread Held	Threads	The 'hold' state of the thread. This may be because of an explicit hold request, or because the thread is waiting at a barrierpoint.
Replay Mode	Processes	The Replay state (Replay or Record) of a process that has Replay enabled.
System TID	Thread	The systID , which is the user thread ID (user TID) if it exists. Otherwise, the system kernel ID (Kernel TID).
User_TID/ Kernel_TID	Thread	User thread ID / Kernel thread ID
Thread Name	Thread	The name of the thread, if set. If unset, "unnamed" is displayed.

The attributes drawer has three buttons for changing the order of attributes. Changing the order matters only for attributes that are selected. The order of selected attributes controls the order they appear in the Processes & Threads view.

Button	Action
	Moves the selected attribute up the list. If it passes another selected attribute, this changes the display in the Processes & Threads view.
	Moves the selected attribute down the list. If it passes another selected attribute, this changes the display in the Processes & Threads view.
	Resets the initial order and the initial selection of attributes.

Debugging Python

- Overview
- Python Debugging Requirements
- Starting a Python Debugging Session
- Debugging Python and C/C++ with TotalView
- Viewing and Comparing Python and C/C++ Variables
- Leveraging Other Debugging Technologies for Python Debugging
- Supported Python Extension Technologies for Stack Transformations

Overview

The Python language is easily extensible with C and C++ code. This enables Python applications to access legacy algorithms, specialized hardware, and to perform highly specialized computing.

C/C++ Python extensions enable developers to "glue" together different parts of a program, creating a mixed language application. Understanding and debugging the interdependencies and data exchange between language barriers in a mixed language application is a real challenge for developers.

TotalView supports debugging Python extensions, shows a clean set of stack frames across the language barriers, and allows both Python and C/C++ variables to be examined and compared.

The debugger does not yet support setting breakpoints and stepping actual Python code as it does with C and C++, but it excels at making it easy to set up your debug session, examine the data exchange between the language barriers, and debug your C/C++ code.

NOTE: The TotalView installation includes some example Python/C mixed language programs in `install_dir/toolworks/totalview.version/platform/examples/PythonExamples`. The `README.TXT` file in the `PythonExamples` subdirectory details requirements and instructions for building and executing these programs.

Python Debugging Requirements

Python Version

To debug C/C++ Python extensions, install the debugging information for your version of the Python interpreter. This provides the necessary insight into the Python data structures for the debugger to extract Python stack and variable information.

Packaged versions of the debugging symbol interpreter can be installed with:

CentOS/RedHat Enterprise/Fedora Linux:

```
sudo yum install python-devel
sudo debuginfo-install glibc
sudo debuginfo-install python
```

Ubuntu:

```
sudo apt-get install python-dev
sudo apt-get install python-dbg
```

Limitations and Extensions:

The following functionality and limitations exist:

- **Python version:** Python 2.7, and Python 3.5 and above.
- **Python compiled with debug information:**

Debug information is required. If you are building your own Python version, the easiest way is to use the `--with-pydebug` flag:

```
configure --with-pydebug
```

This option incorporates the `-g` option (debug information) and the `-O0` option (no optimization). Optimizations can prevent TotalView from obtaining Python function, line, number, variable, or frame information, in which case it displays "Optimized out." See [Building and Using a Debug Version of Python](#) for details on building a debug version of the interpreter.

For a pre-built Python distribution, check the documentation, installation guide and any configure files for information on how it was compiled.

- **Python type support:** Current support for Python types includes scalar types `int`, `float`, `long`, `complex`, `str`, and Numpy `ndarray`. Future support will include other sequence, mapping, and set types.

- **Python extension technologies:** Current support for the many Python extension technologies includes:
 - SWIG to perform stack frame transformations
 - ctypes to call functions in DLLs or shared libraries
 - pybind11 for operability between C++11 and Python

Support for other Python extensions will be added to the product.

- **Python distributions:** Python debugging support has been tested on Python distributed with various operating systems. Support of the [Enthought](#) Python 3.5 distribution has also been validated. The [Anaconda](#) Python distribution is not supported due to the unavailability of debug information with the distribution.

If you have feedback or feature requests on Python debugging in TotalView, please let us know at <https://totalview.io/support>.

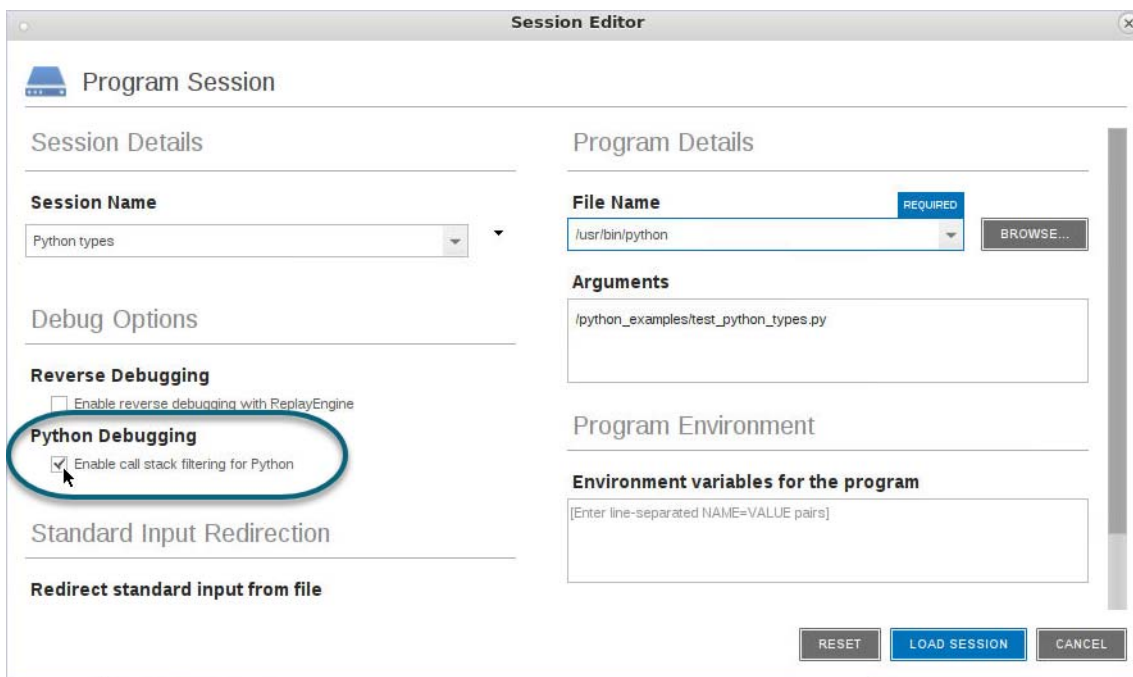
Starting a Python Debugging Session

To set up a Python and C/C++ debugging session with TotalView:

1. Set up a new Program Session.

- Enter the path to the debug Python interpreter into the **File Name** field.
- Enter the name of the Python file to run as an argument to the interpreter in the **Arguments** field,
- Select the checkbox under **Python Debugging**, “Enable call stack filtering for Python”, then click **Load Session**.

Figure 89, Set up a Python Debugging Session



NOTE: The Python-specific option “Enable call stack filtering for Python” ensures that the call stack will display Python calls. Once selected, this option is saved with the session and will be active on next session load.

For faster startup, provide the information as command line arguments to TotalView, for example:

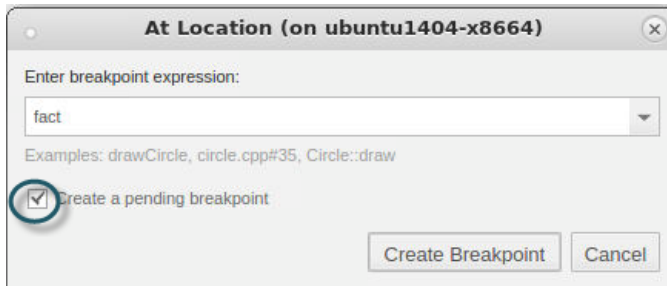
```
totalview --args /usr/bin/python test_python_types.py
```

2. **Set breakpoints in C/C++ code and begin debugging.**

To set a breakpoint in the C/C++ Python extension code, use the **At Location** dialog, available via the **Action Points > At Location** menu.

Enter a Python extension function name or file #line location, and check **“Create a pending breakpoint”** to create a breakpoint in code that TotalView is not yet aware of. Click **Create Breakpoint**.

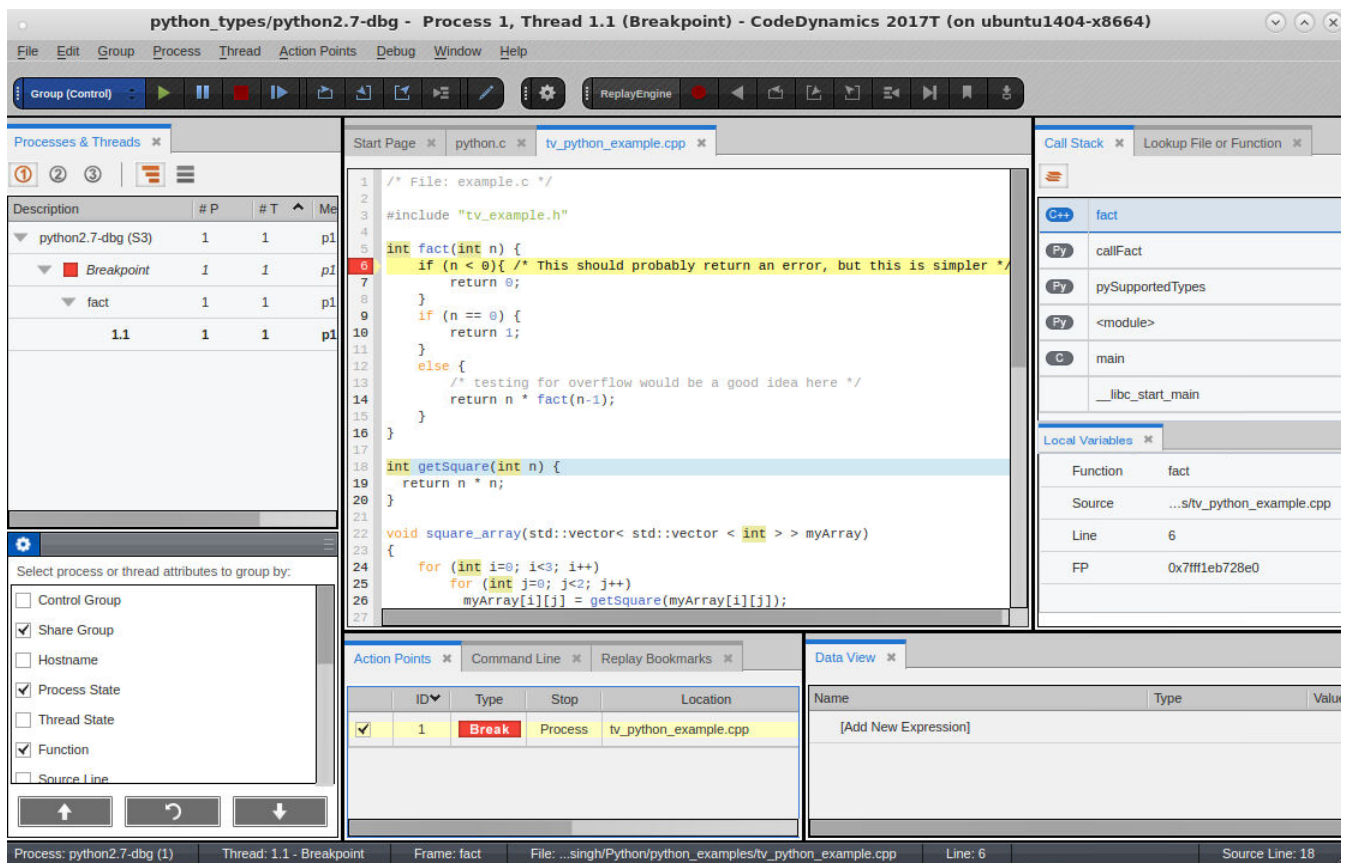
Figure 90, Create a Breakpoint on a Python Extension Function



Debugging Python and C/C++ with TotalView

With the Python session set up and a breakpoint set in the Python extension, start running the Python interpreter by clicking **Go** (▶). TotalView stops when your breakpoint is hit.

Figure 91, Stopped at Python extension function and clean integrated call stack



Note the Call Stack in Figure 91 that displays both C and Python code.

This view has been transformed to hide function calls that just facilitate Python and C/C++ working together. See [Transforming the Stack](#) for how this works.

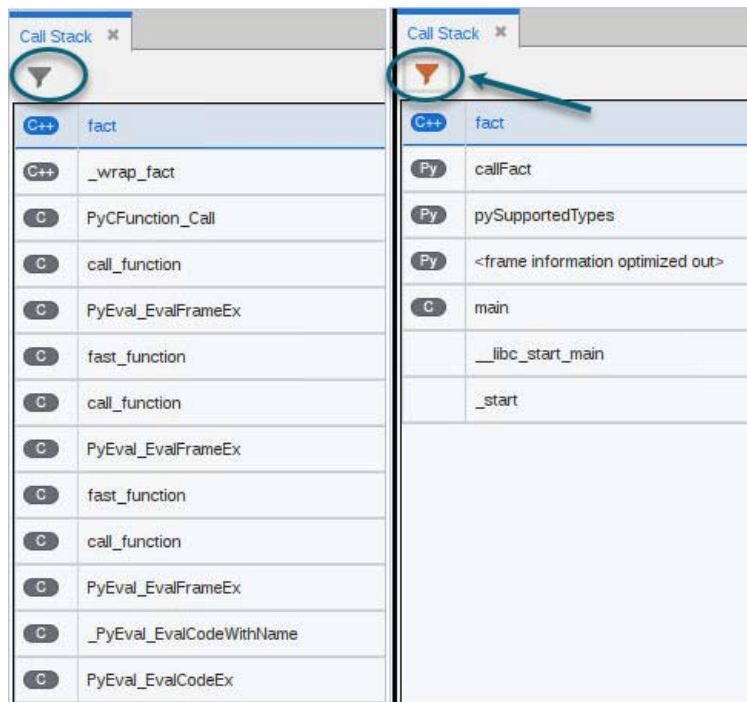
Transforming the Stack


One of the advanced features that TotalView provides is a fully unified Call Stack of all the Python and C/C++ frames. Further, the Call Stack removes all the noisy "glue" calls that tie together the two languages, displaying a concise, developer-oriented view of the call from Python into C/C++.

NOTE: The ability to transform stack frames is a general capability in TotalView known as the Stack Transformation Facility (STF). To learn more about the STF and how it works, see *Part 2, Transformations in the TotalView Reference Guide*.

Figure 92 shows an untransformed stack on the left and a transformed stack on the right. The transformed stack is how the developer conceptually thinks about the calls in a program.

Figure 92, Untransformed Python Stack vs. Transformed Python Stack




NOTE: If the Call Stack has not been transformed to display Python calls, click the transform button (). To preserve this setting, edit the session to ensure that “Enable call stack filtering for Python” is checked under the Python Debugging section. See [Set up a Python Debugging Session](#).

Controlling the transform feature

When TotalView detects you are debugging a Python program, TotalView enables transforming the Python call frame information from the Python interpreter.

You can disable or re-enable stack trace filtering using one of the following methods:

- Click the transform button () in the Call Stack view to toggle the transform on or off.
- Enter the following command in the Command Line view:
`dstacktransform | disable | enable`
- Use a state variable to control the filtering of the stack:
 - `stack_trace_transform_enabled` (defaults to false)
This variable controls whether any stack filtering occurs.

Controlling the transformation of the stack is handled by TotalView's *Stack Transformation Facility* (STF), a rule-driven capability that allows stack frames to be matched against regular expressions, and then applying filters to the matching frames.

RELATED TOPICS

Creating stack transformations	dstacktransform in the <i>TotalView Reference Guide</i>
General information on creating custom type transformations	"Creating Type Transformations" in the <i>TotalView Reference Guide</i>

Viewing and Comparing Python and C/C++ Variables

To compare the data from both sides of the language barriers, click on either the C/C++ frame or the Python frame and observe the values of the local variables in the Local Variables view.

You can drag variables into the Data View to examine and compare them. Alternatively, right-click on the variable name in the Source View and select **Add to Data View...** from the context menu.

Figure 93 displays variable `a` from Python frame `callFact()` added to the Data View. This variable value is passed as argument `n` in the C frame `fact()`. Adding `n` to the Data View allows the values to be compared side-by-side.

NOTE: Python variables are currently read-only in the Data View, indicated by a lock icon.

Figure 93, Comparing Python and C/C++ variables in the Data View

The screenshot displays a debugger interface with several panels. The top-left panel shows C++ source code for a factorial function. The top-right panel shows the Call Stack with entries for 'fact' (C++) and 'callFact' (Py). The bottom-left panel shows a table of Action Points with a break point at line 6. The bottom-right panel shows the Data View with variables 'n' and 'a'.

```
1 /* File: example.c */
2
3 #include "tv_example.h"
4
5 int fact(int n) {
6     if (n < 0){ /* This should probably return an error, but this is simpler */
7         return 0;
8     }
9     if (n == 0) {
10        return 1;
11    }
12    else {
13        /* testing for overflow would be a good idea here */
14        return n * fact(n-1);
15    }
16 }
17
18 int getSquare(int n) {
19     return n * n;
20 }
21
22
```

Name	Type	Value
Arguments		
n	int	0x000000

ID	Type	Stop	Location
1	Break	Process	tv_python_example.cpp

Name	Type	Value
n	int	0x000000
a	int	0x000000

Leveraging Other Debugging Technologies for Python Debugging

TotalView provides multiple powerful debugging features, such as its reverse debugging engine ReplayEngine, which records the execution of the debugging session and then jumps back through execution to understand how the program ran. In addition, MemoryScape easily identifies memory leaks and other memory problems.

Both technologies work while performing Python debugging and enable advanced debugging and analysis of the C/C++ code. See the [ReplayEngine User Guide](#) to learn more about reverse debugging. For information on MemoryScape, see the book *Debugging Memory Problems with MemoryScape* in the product distribution at `<installdir>/<totalview_version>/doc/pdf` or on the [TotalView documentation](#) website, [Debugging Memory Problems with MemoryScape](#).

Supported Python Extension Technologies for Stack Transformations

Natively, Python provides the foreign function library `ctypes`, which provides an infrastructure for calling functions in shared libraries and the exchange of C compatible data types between the language barriers.

The library `ctypes` is not the only solution for calling C; numerous other "glue" technologies exist, implementing an array of approaches to facilitate calling between and exchanging data between Python and C and C++.

Common Python Extension Technologies that Support Stack Transformations

Table 7: Python Extension Technologies for Stack Transformations

Python C/C++ "Glue" Technology	Description
ctypes	A foreign function library for Python. https://docs.python.org/3/library/ctypes.html
Cython	A superset of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes. https://cython.org/
SWIG	A software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python. http://www.swig.org/Doc3.0/Python.html
CFFI	Foreign Function Interface for Python calling C code. http://cffi.readthedocs.io/en/latest/index.html
PyQt/PySide and SIP	SIP is a tool that makes it easy to create Python bindings for C and C++ libraries. https://www.riverbankcomputing.com/software/sip/intro https://www.riverbankcomputing.com/static/Docs/sip/
Boost.Python	A C++ library which enables seamless interoperability between C++ and the Python programming language. http://www.boost.org/doc/libs/1_63_0/libs/python/doc/html/index.html
pybind11	Seamless operability between C++11 and Python. https://pybind11.readthedocs.io/en/stable/index.html

Python Extension Filters Supported by TotalView

has built-in logic to identify and transform the low-level calls in the Python interpreter functions and extract the Python call and variable information. Filtering out the Python extension "glue" code requires further rule definitions tailored to the specific technology. The following table shows the current Python extension filters supported by TotalView.

Table 8: Python Extension Filters Supported by TotalView

Python C/C++ "Glue" Technology	Description
ctypes	A foreign function library for Python. https://docs.python.org/3/library/ctypes.html
SWIG	A software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python. http://www.swig.org/Doc3.0/Python.html

Support for more Python extensions will be added over time but you can also define your own transformation and filter rules as well. Check out the `dstacktransform` documentation for details on creating your own stack transformations.

Using the Command Line Interface (CLI)

- Access to the CLI
- Introduction to the CLI
- About the CLI and Tcl
- Starting the CLI in a Terminal Window
- About CLI Output
- Using Command Arguments
- Using Namespaces
- About the CLI Prompt
- Using Built-in and Group Aliases
- How Parallelism Affects Behavior
- Controlling Program Execution Using CLI Commands
- Examples of Using the CLI

Access to the CLI

TotalView's default settings display the Command Line view, which provides access to the CLI.

Figure 94, Command Line View

```

Command Line

Thread 3.1 has appeared
Thread 4.1 has appeared
Thread 4.1 has appeared
Thread 4.1 has exited
Thread 3.1 has appeared
Thread 3.1 has exited
Thread 4.2 has appeared
Thread 4.3 has appeared
Thread 2.2 has appeared
Thread 2.3 has appeared
dl.<> dbreak 681
Thread 1.2 has appeared
Thread 1.3 has appeared
Thread 3.2 has appeared
Thread 3.3 has appeared
1
Thread 4.1 hit breakpoint 1 at line 681 in "snore(void*)"
Thread 2.3 hit breakpoint 1 at line 681 in "snore(void*)"
Thread 1.2 hit breakpoint 1 at line 681 in "snore(void*)"
Thread 3.2 hit breakpoint 1 at line 681 in "snore(void*)"
dl.<> help
Help with no arguments prints this message.
Help with an argument prints information about one topic.

Help is available on the following specific commands:
alias      capture  dactions  dassign  dattach  dbarrier  dbreak    dcache
dcalltree  dcheckpoint  dcont    dcuda    ddelete  ddetach  ddisable
ddlopen    ddown    denable   dexamine dflush   dfocus    dga        dhistory
dgo        dggroups dhalt     dheap    dhold    dkill     dlappend  dlist
dload      dmstat   dnext     dnexti   dout      dprint    dptsets   drerun
drestart   drun     dsession  dset     dstatus   dstep     dstepi    dunhold
dunset    duntil   dup        dwait    dwatch    dwhat     dwhere    dworker
exit      help     quit      stty     unalias   TV::query TV::hostname

Help is available on the following additional (general) topics:
flush_cache_commands  ptlist  accessors  addressing  aix_malloc
arguments  startup_commands  commands  errorCodes  groups  history
output  properties  ptset  ptexpr  script_commands  search_variables
stepping  startup  spurs  tabs  tcl  variables

dl.<> help dbreak
dbreak { <source-loc-expr> | -address <addr> }
    [ -g|-p|-t ]
    [ [ -l lang ] -e {expr} ]
    [ -pending ]
Default aliases: b, bt

```

If the Command Line view is not present when you start TotalView, open it by:

- Right-clicking in the menu/toolbar area and selecting it from the context menu.
- Selecting from the submenu **Window | View**.

Both of these are toggles that can also be used to close the view.

You can also access the CLI through a separate terminal window, as described in [Starting the CLI in a Terminal Window](#).

The Command Line view gives you access to the **help** command that provides information on many of the common CLI commands, as shown in [Figure 94](#) for **dhistory**.

The Command Line view has two main uses, as represented in [Figure 95](#).

Figure 95, Uses of the Command Line View

```
d1.<> dfocus 2.1
d2.1
Thread 4.2 hit breakpoint 1 at line 681 in "snore(void*)"
Thread 3.3 hit breakpoint 1 at line 681 in "snore(void*)"
Thread 2.1 hit breakpoint 1 at line 681 in "snore(void*)"
Thread 1.2 hit breakpoint 1 at line 681 in "snore(void*)"
d2.1> dfocus 4.3
d4.3
d4.3> dbreak 1108
3
Thread 3.1 hit breakpoint 1 at line 681 in "snore(void*)"
Thread 1.3 hit breakpoint 1 at line 681 in "snore(void*)"
Thread 2.2 hit breakpoint 1 at line 681 in "snore(void*)"
Thread 4.3 hit breakpoint 1 at line 681 in "snore(void*)"
d4.3> history
1 return $TV::version
2 dbreak 681
3 dbreak 1059
4 help
5 help dfocus
6 dfocus 2.1
7 dfocus 4.3
8 dbreak 1108
9 history
d4.3> !6
dfocus 2.1
d2.1
d2.1>
```

- It shows the history of the processes and threads in the debugging session, as recorded by the debugger's output to the CLI.
- It provides a command line interface to the debugger. This interface is extremely powerful, allowing you to invoke any CLI command, as described in the *TotalView Reference Guide*. These commands give you fine-grained control of the debugger and the debugging session, well beyond what you can do in the UI.

Note that the Command Line view supports command history. In [Figure 95](#), typing **history** shows the list of commands executed so far. Entering the command **!6** re-executes the command **dfocus 2.1**. Use the up-arrow and down-arrow keys to move up and down the history list.

This view also supports copy and paste, which works the same as most command line interfaces:

1. Select text by clicking and dragging.
2. Right-click and select Copy.
3. Right-click and select Paste to copy the selected text to the cursor position.

The remainder of this chapter describes the many ways to use the CLI to enhance your debugging session.

Introduction to the CLI

The two components of the Command Line Interface (CLI) are the Tcl-based programming environment and the commands added to the Tcl interpreter that lets you debug your program. This chapter looks at how these components interact, and describes how you specify processes, groups, and threads.

This chapter emphasizes interactive use of the CLI rather than using the CLI as a programming language because many of its concepts are easier to understand in an interactive framework. However, everything in this chapter can be used in both environments.

This chapter contains the following sections:

- [About the CLI and Tcl](#)
- [Starting the CLI in a Terminal Window](#)
- [About CLI Output](#)
- [Using Command Arguments](#)
- [Using Namespaces](#)
- [About the CLI Prompt](#)
- [Using Built-in and Group Aliases](#)
- [How Parallelism Affects Behavior](#)
- [Controlling Program Execution Using CLI Commands](#)
- [Examples of Using the CLI](#)

About the CLI and Tcl

The CLI is built in version 8.0 of Tcl, so TotalView CLI commands are built into Tcl. This means that the CLI is *not* a library of commands that you can bring into other implementations of Tcl. Because the Tcl you are running is the standard 8.0 version, the CLI supports all libraries and operations that run using version 8.0 of Tcl.

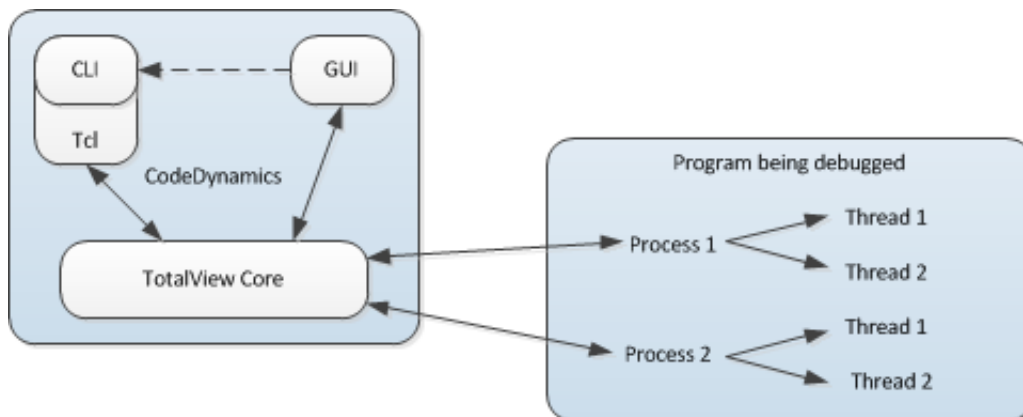
Integrating CLI commands into Tcl makes them intrinsic Tcl commands. This lets you enter and execute all CLI commands in exactly the same way as you enter and execute built-in Tcl commands. As CLI commands are also Tcl commands, you can embed Tcl primitives and functions in CLI commands, and embed CLI commands in sequences of Tcl commands.

For example, you can create a Tcl list that contains a list of threads, use Tcl commands to manipulate that list, and then use a CLI command that operates on the elements of this list. You can also create a Tcl function that dynamically builds the arguments that a process uses when it begins executing.

Integration of the CLI and the UI

Figure 96 illustrates the relationships between the CLI, the GUI, the TotalView core, and your program:

Figure 96, How the CLI Interacts with TotalView



The CLI and the GUI are components that communicate with the TotalView core, which drives the debugging session. In this figure, the dotted arrow between the GUI and the CLI indicates that you can invoke the CLI from the GUI, but the reverse is not true: you cannot invoke the GUI from the CLI.

In turn, the TotalView core communicates with the processes that make up your program, receives information back from these processes, and passes information back to the component that sent the request. If the GUI is also active, the core also updates the GUI's views. For example, stepping your program with the CLI changes the PC in the source view, updates data values, and so on.

Invoking CLI Commands

You interact with the CLI by entering a CLI or Tcl command. (Entering a Tcl command does exactly the same thing in the CLI as it does when interacting with a Tcl interpreter.) Typically, the effect of executing a CLI command is one or more of the following:

- The CLI displays information about your program.
- A change takes place in your program's state.
- A change takes place in the information that the CLI maintains about your program.

After the CLI executes your command, it displays a prompt. Although CLI commands are executed sequentially, commands executed by your program might not be. For example, the CLI does not require that your program be stopped when it prompts for and performs commands. It only requires that the last CLI command be complete before it can begin executing the next one. In many cases, the processes and threads being debugged continue to execute after the CLI has finished doing what you asked it to do.

If you need to stop an executing CLI command or Tcl macro, press Ctrl+C while the command is executing. If the CLI is displaying its prompt, typing Ctrl+C stops any executing processes.

Because actions are occurring constantly, state information and other kinds of messages that the CLI displays are usually mixed in with the commands that you type. You might want to limit the amount of information TotalView displays by setting the **VERBOSE** variable to **WARNING** or **ERROR**. (For more information, see the *"TotalView Variables"* chapter in the *TotalView Reference Guide*.)

Starting the CLI in a Terminal Window

In the GUI, the Command Line view represents a command window with the CLI enabled.

In a terminal window, you start the CLI by typing **totalviewcli** (assuming that the TotalView binary directory is in your path.)

If you have problems entering and editing commands, it might be because you have invoked the CLI from a shell or process that manipulates your **stty** settings. You can eliminate these problems if you use the **stty sane** CLI command. (If the **sane** option isn't available, you have to change values individually.)

If you start the CLI with the **totalviewcli** command, you can use all of the command-line options that you can use when starting TotalView, except those that have to do with the GUI. (In some cases, TotalView displays an error message if you try. In others, it just ignores what you did.)

Information on command-line options is in the "TotalView Command Syntax" chapter of the *TotalView Reference Guide*.

Startup Example

The following is a very small CLI script:

```
dload fork_loop
dset ARGS_DEFAULT {0 4 -wp}
dstep
catch {make_actions fork_loop.cxx} msg
puts $msg
```

This script loads the **fork_loop** executable, sets its default startup arguments, and steps one source-level statement.

If you stored this in a file named **fork_loop.tvd**, you could tell TotalView to start the CLI and execute this file by entering the following command:

```
totalviewcli -s fork_loop.tvd
```

The following example places a similar set of commands in a file that you invoke from the shell:

```
#!/bin/sh
# Next line executed by shell, but ignored by Tcl because: \
exec totalviewcli -s "$0" "$@"
dload fork_loop
dset ARGS_DEFAULT {0 4 -wp}
dstep
catch {make_actions fork_loop.cxx} msg
puts $msg
```

These two examples are essentially the same except for the first few lines in the second example. In the second example, the shell ignores the backslash continuation character; Tcl processes it. This means that the shell executes the **exec** command while Tcl ignores it.

Starting Your Program

The CLI lets you start debugging operations in several ways. To execute your program from within the CLI, enter a **dload** command followed by the **drun** command.

If your program is launched from a starter program such as **srn** or **yod**, use the **drerun** command rather than **drun** to start your program. If you use **drun**, default arguments to the process are suppressed; **drerun** passes them on.

The following example uses the **totalviewcli** command to start the CLI. This is followed by **dload** and **drun** commands. Since this was not the first time the file was run, breakpoints exist from a previous session.

In this listing, the CLI prompt is "d1.<>". The information preceding the greater-than symbol (>) symbol indicates the processes and threads upon which the current command acts. The prompt is discussed in [About the CLI Prompt](#).

```
% totalviewcli
d1.<> dload arraysAlpha #load the arraysAlpha program
1
d1.<> dactions # Show the action points
No matching breakpoints were found
d1.<> dlist -n 10 75
75 reall6_array (i, j) = 4.093215 * j+2
76 #endif
77 26 continue
78 27 continue
79
80 do 40 i = 1, 500
81 denorms(i) = x'00000001'
82 40 continue
83 do 42 i = 500, 1000
84 denorms(i) = x'80000001'
d1.<> dbreak 80 # Add two action points
1
d1.<> dbreak 83
2
d1.<> drun # Run the program to the action point
```

This two-step operation of loading and running supports setting action points before execution begins, as well as executing a program more than once. At a later time, you can use **drerun** to restart your program, perhaps sending it new arguments. In contrast, reentering the **dload** command reloads the program into memory (for example, after editing and recompiling the program).

The **dload** command always creates a new process. The new process is in addition to any existing processes for the program because the CLI does not shut down older processes when starting the new one.

The **dkill** command terminates one or more processes of a program started by using a **dload**, **drun**, or **drrun** command. The following example continues where the previous example left off:

```
d1.<> dkill # kills process
d1.<> drun # runs program from start
d1.<> dlist -e -n 3 # shows lines about current spot
79
80@> do 40 i = 1, 500
81 denorms(i) = x'00000001'
d1.<> dwhatmaster_array # Tell me about master_array
In thread 1.1:
Name: master_array; Type: integer(100);
Size: 400 bytes; Addr: 0x140821310
Scope: ##arraysAlpha#arrays.F#check_fortran_arrays
(Scope class: Any)
Address class: proc_static_var
(Routine static variable)
d1.<> dgo # Start program running
d1.<> dwhat denorms # Tell me about denorms
In thread 1.1:
Name: denorms; Type: <void>; Size: 8 bytes;
Addr: 0x1408214b8
Scope: ##arraysAlpha#arrays.F#check_fortran_arrays
(Scope class: Any)
Address class: proc_static_var
(Routine static variable)
d1.<> dprint denorms(0) # Show me what is stored
denorms(0) = 0x0000000000000001 (1)
d1.<>
```

Because information is interleaved, you may not realize that the prompt has re-appeared. It is always safe to use the Enter key to have the CLI redisplay its prompt. If a prompt isn't displayed after you press Enter, you know that the CLI is still executing.

About CLI Output

A CLI command can either print its output to a window or return the output as a character string. If the CLI executes a command that returns a string value, it also prints the returned string. Most of the time, you won't care about the difference between *printing* and *returning-and-printing*. Either way, the CLI displays information in your window. And, in both cases, printed output is fed through a simple *more* processor (see below).

In the following two cases, it matters whether the CLI directly prints output or returns and then prints it:

- When the Tcl interpreter executes a list of commands, the CLI only prints the information returned from the last command. It doesn't show information returned by other commands.
- You can only assign the output of a command to a variable if the CLI returns a command's output. You can't assign output that the interpreter prints directly to a variable, or otherwise manipulate it, unless you save it using the **capture** command.

For example, the **dload** command returns the ID of the process object that was just created. The ID is normally printed—unless, of course, the **dload** command appears in the middle of a list of commands; for example:

```
{dload test_program;dstatus}
```

In this example, the CLI doesn't display the ID of the loaded program since the **dload** command was not the last command.

When information is returned, you can assign it to a variable. For example, the next command assigns the ID of a newly created process to a variable:

```
set pid [dload test_program]
```

Because the **help** command only prints its output without returning a string, the following does not work:

```
set htext [help]
```

This statement assigns just an empty string to **htext**.

To save the output of a command that prints its output, use the **capture** command. For example, the following example writes the **help** command's output into a variable:

```
set htext [capture help]
```

You can capture the output only from commands. You can't capture the informational messages displayed by the CLI that describe process state. If you are using the UI, TotalView also writes this information to the Input/Output view.

'more' Processing

When the CLI displays output, it sends data through a simple *more*-like process. This prevents data from scrolling off the screen before you view it. After you see the **MORE** prompt, press Enter to see the next screen of data. If you enter **q**, the CLI discards any data it hasn't yet displayed.

You can control the number of lines displayed between prompts by using the **dset** command to set the **LINES_PER_SCREEN** CLI variable. (For more information, see the *TotalView Reference Guide*.)

Using Command Arguments

The default command arguments for a process are stored in the **ARGS(*num*)** variable, where *num* is the CLI ID for the process. If you don't set the **ARGS(*num*)** variable for a process, the CLI uses the value stored in the **ARGS_DEFAULT** variable. TotalView sets the **ARGS_DEFAULT** variable when you use the **-a** option when starting the CLI or the GUI.

The **-a** option passes everything that follows on the command line to the program.

For example:

```
totalviewcli -a argument-1, argument-2, ...
```

To set (or clear) the default arguments for a process, you can use the **dset** (or **dunset**) command to modify the **ARGS()** variables directly, or you can start the process with the **drun** command. For example, the following clears the default argument list for process 2:

```
dunset ARGS(2)
```

The next time process 2 is started, the CLI uses the arguments contained in **ARGS_DEFAULT**.

You can also use the **dunset** command to clear the **ARGS_DEFAULT** variable; for example:

```
dunset ARGS_DEFAULT
```

All commands (except the **drun** command) that can create a process—including the **dgo**, **drrun**, **dcont**, **dstep**, and **dnext** commands—pass the default arguments to the new process. The **drun** command differs in that it replaces the default arguments for the process with the arguments that are passed to it.

Using Namespaces

CLI interactive commands exist in the primary Tcl namespace (`::`). Some of the TotalView state variables also reside in this namespace. Seldom-used functions and functions that are not primarily used interactively reside in other namespaces. These namespaces also contain most TotalView state variables. (The variables that appear in other namespaces are usually related to TotalView preferences.) TotalView uses the following namespaces:

TV::

Contains commands and variables that you use when creating functions. They can be used interactively, but this is not their primary role.

TV::GUI::

Contains state variables that define and describe properties of the user interface, such as window placement and color.

If you discover other namespaces beginning with **TV**, you have found a namespace that contains private functions and variables. These objects can (and will) disappear, so don't use them. Also, don't create namespaces that begin with **TV**, since you can cause problems by interfering with built-in functions and variables.

The CLI **dset** command lets you set the value of these variables. You can have the CLI display a list of these variables by specifying the namespace; for example:

```
dset TV::
```

You can use wildcards with this command. For example, **dset TV::au*** displays all variables that begin with "au".

About the CLI Prompt

The appearance of the CLI prompt lets you know that the CLI is ready to accept a command. This prompt lists the current focus, and then displays a greater-than symbol (>) and a blank space. The current focus is the processes and threads to which the next command applies. Here are some examples:

d1.<>

The current focus is the default focus set for each command, which is first user thread in process 1.

g2.3>

The current focus is process 2, thread 3; commands act on the entire group.

t1.7>

The current focus is thread 7 of process 1.

gW3.>

The current focus is all worker threads in the control group that contains process 3.

p3/3

The current focus is all processes in process 3, group 3.

You can change the prompt's appearance by using the **dset** command to set the **PROMPT** state variable; for example:

```
dset PROMPT "Kill this bug! > "
```

Using Built-in and Group Aliases

Many CLI commands have an alias that lets you abbreviate the command's name. (An alias is one or more characters that Tcl interprets as a command or command argument.)

The **alias** command, which is described in the *TotalView Reference Guide*, lets you create your own aliases.

For example, the following command halts the current group:

```
dfocus g dhalt
```

Using an abbreviation is easier. The following command does the same thing:

```
f g h
```

You often type less-used commands in full, but some commands are almost always abbreviated. These commands include **dbreak (b)**, **ddown (d)**, **dfocus (f)**, **dgo (g)**, **dlist (l)**, **dnext (n)**, **dprint (p)**, **dstep (s)**, and **dup (u)**.

The CLI also includes uppercase group versions of aliases for a number of commands, including all stepping commands. For example, the alias for **dstep** is **s**; in contrast, the alias for **dfocus g dstep** is **S**. The first command steps the process. The second steps the control group.

Group aliases differ from the group-level command that you type interactively, as follows:

- They do not work if the current focus is a list. The **g** focus specifier modifies the current focus, and can only be applied if the focus contains just one term.
- They always act on the group, no matter what width is specified in the current focus. Therefore, **dfocus t S** does a step-group command.

How Parallelism Affects Behavior

A parallel program consists of some number of processes, each involving some number of threads. Processes fall into two categories, depending on when they are created:

- **Initial process**

A pre-existing process from the normal run-time environment (that is, created outside TotalView), or one that was created as TotalView loaded the program.

- **Spawned process**

A new process created by a process executing under CLI control.

TotalView assigns an integer value to each individual process and thread under its control. This process/thread identifier can be the system identifier associated with the process or thread. However, it can be an arbitrary value created by the CLI. Process numbers are unique over the lifetime of a debugging session; in contrast, thread numbers are only unique while the process exists.

Process/thread notation lets you identify the component that a command targets. For example, if your program has two processes, and each has two threads, four threads exist:

- Thread 1 of process 1
- Thread 2 of process 1
- Thread 1 of process 2
- Thread 2 of process 2

You identify the four threads as follows:

- 1.1—Thread 1 of process 1
- 1.2—Thread 2 of process 1
- 2.1—Thread 1 of process 2
- 2.2—Thread 2 of process 2

RELATED TOPICS

An overview of threads and processes and how TotalView organizes them into groups

[What Is a Group? and How TotalView Creates Groups](#)

More on TotalView thread/process width

[Executing at Process Width and Executing at Thread Width](#)

Types of IDs

Multi-threaded, multi-process, and distributed programs contain a variety of IDs. The following types are used in the CLI and the GUI:

System PID

This is the process ID and is generally called the PID.

System TID

This is the ID of the system kernel or user thread. On some systems (for example, AIX), the TIDs have no obvious meaning. On other systems, they start at 1 and are incremented by 1 for each thread.

TotalViewthread ID

This is usually identical to the system TID. On some systems (such as AIX) where the threads have no obvious meaning, TotalView uses its own IDs.

pthread ID

This is the ID assigned by the Posix pthreads package. If this differs from the system TID, the TID is a pointer value that points to the pthread ID.

Debugger PID

This is an ID created by TotalView to identify processes. It is a sequentially numbered value beginning at 1 that is incremented for each new process. If the target process is killed and restarted (that is, you use the **dkill** and **drun** commands), the TotalView PID does not change. The system PID changes, however, since the operating system has created a new target process.

Controlling Program Execution Using CLI Commands

Knowing what's going on and where your program is executing is simple in a serial debugging environment. Your program is either stopped or running. When it is running, an event such as arriving at a breakpoint can occur, stopping the program. Sometime later, you tell the serial program to continue executing.

Multi-process and multi-threaded programs are more complicated. Each thread and each process has its own execution state. When a thread (or set of threads) triggers a breakpoint, TotalView must also determine how other threads and processes respond, stopping some and letting others continue to run.

RELATED TOPICS

Stepping commands	Stepping and Program Execution
The dload command	dload in "CLI Commands" in the <i>TotalView Reference Guide</i>
The dattach command	dattach in "CLI Commands" in the <i>TotalView Reference Guide</i>
The drun command	drun in "CLI Commands" in the <i>TotalView Reference Guide</i>
The dkill command	dkill in "CLI Commands" in the <i>TotalView Reference Guide</i>

Advancing Program Execution

Debugging begins by entering a **dload** or **dattach** command. If you use the **dload** command, you must use the **drun** command (or perhaps **drerun** if there's a starter program) to start the program executing. These three commands work at the process level and you can't use them to start individual threads. This is also true for the **dkill** command.

To advance program execution, you enter a command that causes one or more threads to execute instructions. The commands are applied to a P/T set. (See "**Compressed List Syntax**" in the *TotalView Reference Guide*.)

Because the set doesn't have to include all processes and threads, you can cause some processes to be executed while holding others back. You can also advance program execution by increments, *stepping* the program forward, and you can define the size of the increment. For example, **dnext 3** executes the next three statements, and then pauses what you've been stepping.

Typically, debugging a program means that you have the program run, and then you stop it and examine its state. In this sense, a debugger can be thought of as a tool that lets you alter a program's state in a controlled way, and debugging is the process of stopping a process to examine its state. However, the term *stop* has a slightly differ-

ent meaning in a multi-process, multi-threaded program. In these programs, *stopping* means that the CLI holds one or more threads at a location until you enter a command to start executing again. Other threads, however, may continue executing.

Using Action Points

Action points tell the CLI to stop a program's execution. You can specify the following types of action points:

- A *breakpoint* (see **dbreak** in the *TotalView Reference Guide*) stops the process when the program reaches a location in the source code.
- A *watchpoint* (see **dwatch** in the *TotalView Reference Guide*) stops the process when the value of a variable is changed.
- A *barrierpoint* (see **dbarrier** in the *TotalView Reference Guide*), as its name suggests, effectively prevents processes from proceeding beyond a point until all other related processes arrive. This gives you a method for synchronizing the activities of processes. (You can set a barrierpoint only on processes; you cannot set them on individual threads.)
- An *evalpoint* (see **dbreak** in the *TotalView Reference Guide*) lets you programmatically evaluate the state of the process or variable when execution reaches a location in the source code. An evalpoint typically does not stop the process; instead, it performs an action. In most cases, an evalpoint stops the process when some condition that you specify is met.

Each action point is associated with an action point identifier. You use these identifiers when you need to refer to the action point. Like process and thread identifiers, action point identifiers are assigned numbers as they are created. The ID of the first action point created is 1, the second ID is 2, and so on. These numbers are never reused during a debugging session.

The CLI and the GUI let you assign only one action point to a source code line, but you can make this action point as complex as you need it to be.

Examples of Using the CLI

The CLI is a command-line debugger that is completely integrated with TotalView. You can use it and never use the TotalView GUI, or you can use it and the GUI simultaneously. Because the CLI is embedded in a Tcl interpreter, you can also create debugging functions that exactly meet your needs. When you do this, you can use these functions in the same way that you use TotalView' built-in CLI commands.

This section contains macros that show how the CLI programmatically interacts with your program and with TotalView. Reading examples without bothering too much about details gives you an appreciation for what the CLI can do and how you can use it. With a basic knowledge of Tcl, you can make full use of all CLI features.

In each macro in this chapter, all Tcl commands that are unique to the CLI are displayed in bold. These macros perform the following tasks:

- **Setting the CLI EXECUTABLE_PATH Variable**
- **Initializing an Array Slice**
- **Printing an Array Slice**
- **Writing an Array Variable to a File**
- **Automatically Setting Breakpoints**

Setting the CLI EXECUTABLE_PATH Variable

The following macro recursively descends through all directories, starting at a location that you enter. (This is indicated by the *root* argument.) The macro ignores directories named in the *filter* argument. The result is set as the value of the CLI **EXECUTABLE_PATH** state variable.

See also the *TotalView Reference Guide's* entry for the **EXECUTABLE_PATH** variable

```
# Usage:
#
# rpath [root] [filter]
#
# If root is not specified, start at the current
# directory. filter is a regular expression that removes
# unwanted entries. If it is not specified, the macro
# automatically filters out CVS/RCS/SCCS directories.
#
# The search path is set to the result.

proc rpath {{root "."} {filter "/(CVS|RCS|SCCS)(/|$)"} } {
# Invoke the UNIX find command to recursively obtain
```

```

# a list of all directory names below "root".
set find [split [exec find $root-type d-print] \n]

set npath ""

# Filter out unwanted directories.
foreach path $find {
if {![regexp $filter $path]} {
append npath ":"
append npath $path
}
}

# Tell TotalView to use it.
dset EXECUTABLE_PATH $npath
}

```

In this macro, the last statement sets the **EXECUTABLE_PATH** state variable. This is the only statement that is unique to the CLI. All other statements are standard Tcl.

The **dset** command, like most interactive CLI commands, begins with the letter **d**. (The **dset** command is only used in assigning values to CLI state variables. In contrast, values are assigned to Tcl variables by using the standard Tcl **set** command.)

Initializing an Array Slice

The following macro initializes an array slice to a constant value:

```

array_set (var lower_bound upper_bound val) {
for {set i $lower_bound} {$i <= $upper_bound} {incr i}{
dassign $var\($i) $val
}
}

```

The CLI **dassign** command assigns a value to a variable. In this case, it is setting the value of an array element. Use this function as follows:

```

d1.<> dprint list3
list3 = {
(1) = 1 (0x00000001)
(2) = 2 (0x00000001)
(3) = 3 (0x00000001)
}
d1.<> array_set list 2 3 99
d1.<> dprint list3
list3 = {
(1) = 1 (0x00000001)
(2) = 99 (0x00000063)
(3) = 99 (0x00000063)
}

```

For more information on slices, see the section “*Displaying Array Slices*,” in the *Classic TotalView User Guide* in the product distribution at `<installdir>/totalview.<version>/doc/pdf` or on the [TotalView Documentation](#) website, [Displaying Array Slices](#).

Printing an Array Slice

The following macro prints a Fortran array slice. This macro, like others shown in this chapter, relies heavily on Tcl and uses unique CLI commands sparingly.

```
proc pf2Dslicing {anArray i1 i2 j1 j2 {i3 1} {j3 1} \
  {width 20}} {
  for {set i $i1} {$i <= $i2} {incr i $i3} {
    set row_out ""
    for {set j $j1} {$j <= $j2} {incr j $j3} {
      set ij [capture dprint $anArray($i,$j)]
      set ij [string range $ij \
        [expr [string first "=" $ij] + 1] end]
      set ij [string trimright $ij]
      if {[string first "-" $ij] == 1} {
        set ij [string range $ij 1 end]}
      append ij " "
      append row_out " " \
        [string range $ij 0 $width] " "
    }
  }
  puts $row_out
}
```

The CLI’s **dprint** command lets you specify a slice. For example, you can type: **dprint a(1:4,1:4)**.

After invoking this macro, the CLI prints a two-dimensional slice (**i1:i2:i3, j1:j2:j3**) of a Fortran array to a numeric field whose width is specified by the **width** argument. This width does not include a leading minus sign (-).

All but one line is standard Tcl. This line uses the **dprint** command to obtain the value of one array element. This element’s value is then captured into a variable. The CLI **capture** command allows a value that is normally printed to be sent to a variable. For information on the difference between values being displayed and values being returned, see [About CLI Output](#).

The following shows how this macro is used:

```
d1.<> pf2Dslicing a 1 4 1 4
0.841470956802 0.909297406673 0.141120001673-0.756802499294
0.909297406673-0.756802499294-0.279415488243 0.989358246326
0.141120001673-0.279415488243 0.412118494510-0.536572933197
-0.756802499294 0.989358246326-0.536572933197-0.287903308868
d1.<> pf2Dslicing a 1 4 1 4 1 1 17
0.841470956802 0.909297406673 0.141120001673-0.756802499294
0.909297406673-0.756802499294-0.279415488243 0.989358246326
0.141120001673-0.279415488243 0.412118494510-0.536572933197
-0.756802499294 0.989358246326-0.536572933197-0.287903308868
d1.<> pf2Dslicing a 1 4 1 4 2 2 10
```

```
0.84147095 0.14112000
0.14112000 0.41211849
d1.<> pf2Dslice a 2 4 2 4 2 2 10
-0.75680249 0.98935824
0.98935824 -0.28790330
d1.<>
```

Writing an Array Variable to a File

It often occurs that you want to save the value of an array so that you can analyze its results at a later time. The following macro writes array values to a file:

```
proc save_to_file {var fname} {
  set values [capturedprint$var]
  set f [open $fname w]

  puts $f $values
  close $f
}
```

The following example shows how you might use this macro. Using the **exec** command displays the file that was just written.

```
d1.<> dprint list3
list3 = {
(1) = 1 (0x00000001)
(2) = 2 (0x00000002)
(3) = 3 (0x00000003)
}
d1.<> save_to_file list3 foo
d1.<> exec cat foo
list3 = {
(1) = 1 (0x00000001)
(2) = 2 (0x00000002)
(3) = 3 (0x00000003)
}
d1.<>
```

Automatically Setting Breakpoints

In many cases, your knowledge of what a program is doing lets you make predictions as to where problems are occurring. The following CLI macro parses comments that you can include in a source file and, depending on the comment's text, sets a breakpoint or an evalpoint.

(For detailed information on action points, see the section [Breakpoints](#).)

Following this macro is an excerpt from a program that uses it.

```
#make_actions: Parse a source file, and insert
# evaluation and breakpoints according to comments.
#
proc make_actions {{filename ""}} {
```

```

if {$filename == ""} {
puts "You need to specify a filename"
error "No filename"
}

# Open the program's source file and initialize a
# few variables.
set fname [set filename]
set fsource [open $fname r]
set lineno 0
set incomment 0

# Look for "signals" that indicate the type of
# action point; they are buried in the comments.
while {[gets $fsource line] !=-1} {
incr lineno
set bpline $lineno

# Look for a one-line evalpoint. The
# format is ... /* EVAL: some_text */.
# The text after EVAL and before the "*/" in
# the comment is assigned to "code".
if [regexp "/\\* EVAL: *(.*)\\*/" $line all code] {
dbreak $fname\#$bpline -e $code
continue
}

# Look for a multiline evalpoint.
if [regexp "/\\* EVAL: *(.*)" $line all code] {
# Append lines to "code".
while {[gets $fsource interiorline] !=-1} {
incr lineno

# Tabs will confuse dbreak.
regsub-all \t $interiorline \
" " interiorline

# If "*/" is found, add the text to "code",
# then leave the loop. Otherwise, add the
# text, and continue looping.
if [regexp "(.*)\\*/" $interiorline \
all interiorcode]{
append code \n $interiorcode
break
} else {
append code \n $interiorline
}
}
dbreak $fname\#$bpline -e $code
continue
}

# Look for a breakpoint.
if [regexp "/\\* STOP: .*" $line] {
dbreak $fname\#$bpline
continue
}

# Look for a command to be executed by Tcl.
if [regexp "/\\* *CMD: *(.*)\\*/" $line all cmd] {
puts "CMD: [set cmd]"
eval $cmd
}

```

```

}
close $fsource
}

```

Like the previous macros, almost all of the statements are Tcl. The only purely CLI commands are the instances of the **dbreak** command that set evalpoints and breakpoints.

The following excerpt from a larger program shows how to embed comments in a source file that is read by the **make_actions** macro:

```

...
struct struct_bit_fields_only {
unsigned f3 : 3;
unsigned f4 : 4;
unsigned f5 : 5;
unsigned f20 : 20;
unsigned f32 : 32;
} sbfo, *sbfop = &sbfo;
...
int main()
{
struct struct_bit_fields_only *lbfop = &sbfo;
...
int i;
int j;
sbfo.f3 = 3;
sbfo.f4 = 4;
sbfo.f5 = 5;
sbfo.f20 = 20;
sbfo.f32 = 32;
...
/* TEST: Check to see if we can access all the
values */
i=i; /* STOP: */
i=1; /* EVAL: if (sbfo.f3 != 3) $stop; */
i=2; /* EVAL: if (sbfo.f4 != 4) $stop; */
i=3; /* EVAL: if (sbfo.f5 != 5) $stop; */
...
return 0;
}

```

The **make_actions** macro reads a source file one line at a time. As it reads these lines, the regular expressions look for comments that begin with **/* STOP**, **/* EVAL**, and **/* CMD**. After parsing the comment, it sets a breakpoint at a *stop* line, an evalpoint at an *eval* line, or executes a command at a *cmd* line.

Using evalpoints can be confusing because evalpoint syntax differs from that of Tcl. In this example, the **\$stop** function is built into the CLI. Stated differently, you can end up with Tcl code that also contains C, C++, Fortran, and TotalView functions, variables, and statements. Fortunately, you only use this kind of mixture in a few places and you'll know what you're doing.

Reverse Connections

- [About Reverse Connections](#)
- [Starting a Reverse Connect Session](#)
- [Reverse Connect Examples](#)
- [Troubleshooting Reverse Connections](#)

The organization of modern HPC systems often makes it difficult to deploy tools such as TotalView. For example, the compute nodes in a cluster may not have access to any X libraries or X forwarding, so launching a GUI on a compute node is not possible.

Using the Reverse Connect feature, you can run the TotalView UI on a front-end node to debug a job executing on compute nodes.

The basic process is to embed the **tvconnect** command in a batch script; when the batch job runs, the **tvconnect** process connects with the TotalView client to start the debugger server process on the batch node. The TotalView client would typically run on a front-end node, where the application is built and batch jobs are submitted.

About Reverse Connections

When using reverse connect, TotalView is started in two stages:

1. Run the **tvconnect** command to create a debugging request, typically from a batch job on a batch node or compute node in a cluster. The **tvconnect** command accepts the name of the program to debug, along with any arguments to pass to the program.

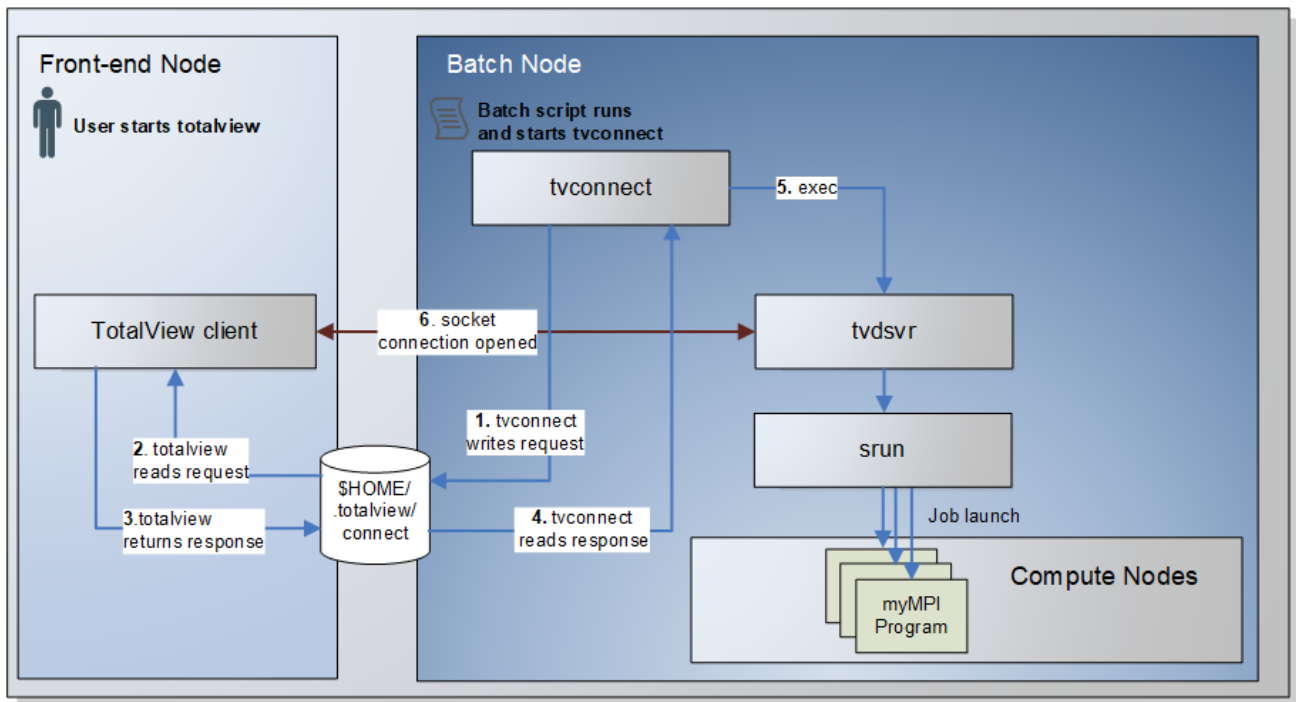
The **tvconnect** process blocks for a TotalView session to accept the request.

2. Start TotalView on another node, which is typically a front-end node. When the UI opens, TotalView looks for a request, and if it finds one, confirms via a pop-up to accept it. If the request is accepted, TotalView starts a debugger server on the node where the **tvconnect** process is running, and loads the program that was passed to the **tvconnect** command. If the request is rejected, the **tvconnect** process exits with an error.

At that point, you can debug the program in the normal way within the TotalView UI.

Here's a little more detail on how the process works.

Figure 97, Reverse connection flow



Typically, a **tvconnect** command is added to a batch script, placed in front of the command to debug. For example:

```
tvconnect srun -n4 myMPIprogram
```

Once a batch script runs and starts the **tvconnect** command and a TotalView front-end UI is started:

1. The **tvconnect** command creates a request in the `$HOME/.totalview/connect` directory and blocks indefinitely until the request is either accepted or rejected. If the **tvconnect** process is killed with a **SIGINT** or **SIGTERM**, the **tvconnect** process deletes the request it created.
2. TotalView reads the request file written by the **tvconnect** process.
3. TotalView accepts or rejects the request, sending back a response.
4. **tvconnect** reads the response. If it was accepted:
5. **tvconnect** execs **tvdsvr**, the command that allows TotalView to control and debug a program on a remote machine.
6. **tvdsvr** opens a connection to the TotalView UI. TotalView then loads the program and any program arguments, using the parameters provided to **tvconnect**. In this example, **srun** was loaded to debug an MPI job.

NOTE: TotalView does not look for reverse connect requests once it starts to debug a program, i.e., it automatically listens only if no other debug session is active. You can choose, however, to continue to listen for connection requests while debugging. See [Listening for Reverse Connections](#).

Reverse connections are also supported by the CLI **dload** command, which has options to either accept or reject reverse connections. In addition, some command line arguments and special environment variables are available that can be used to modify some behavior.

RELATED TOPICS

dload command's reverse connect options

list_reverse_connect under **dload** in the *TotalView Reference Guide*

reject_reverse_connect in the *TotalView Reference Guide*

accept_reverse_connect in the *TotalView Reference Guide*

Environment variables specific to reverse connections

[Reverse Connection Environment Variables](#)

RELATED TOPICS

State variable **TV::reverse_connect_wanted**

TV::reverse_connect_wanted in the *TotalView Reference Guide*

Command line arguments specific to reverse connections

-reverse_connect and **-no_reverse_connect** in the "Command Syntax: chapter of the *TotalView Reference Guide*

Reverse Connection Environment Variables

TotalView supports two special reverse-connection specific environment variables:

- **TV_REVERSE_CONNECT_DIR**
- **TVCONNECT_OPTIONS**

TV_REVERSE_CONNECT_DIR

The environment variable **TV_REVERSE_CONNECT_DIR** identifies the directory where the request and response files will be written and read.

The default location is the user's `$HOME/.totalview/connect` directory.

To customize the location for your reverse connection files, set this environment variable before starting **tvconnect** and TotalView:

```
setenv TV_REVERSE_CONNECT_DIR /home/tv-reverse-connect/tmp
```

Reverse Connection Directory Requirements

The directory that will contain the generated reverse connect files must:

- Be owned by the same user that is running the **tvconnect** process and the TotalView client.
- Have permissions that allow access only by the user. No "Group" or "Other" permissions are allowed.

By default, **tvconnect** creates the connect directory with the following permissions:

```
>ls -l ~/.totalview/
total 80
drwx----- 2 smith tss 4096 Jul 23 12:11 connect
```

TV_CONNECT_OPTIONS

The environment variable **TVCONNECT_OPTIONS** supports the ability to add extra arguments to the **tvconnect** command. One such option might be `-ipv6_support`, which adds support for IPv6 addresses. For example:

```
setenv TVCONNECT_OPTIONS="-ipv6_support"  
tvconnect ~/tx_hello
```

or just:

```
env TVCONNECT_OPTIONS="-ipv6_support" tvconnect ~/tx_hello
```

Starting a Reverse Connect Session

1. Run the **tvconnect** command on a “back-end” compute node. This node does not need access to X libraries to launch a UI. Provide as an argument the program to debug. For example, at its most simple:

```
tvconnect /home/totalview/tests/myTest
```

This command creates a request file in the user’s `$HOME/.totalview/connect` directory. The file contains all the information needed to launch a debugging session on a “front-end” where TotalView is installed with UI capabilities. The request includes such things as the remote host name, the IP address, the user’s home directory, and other information required to launch the debugging session.

This command then blocks waiting for a response.

2. Start TotalView with no arguments on the server where you will perform debugging:

```
totalview
```

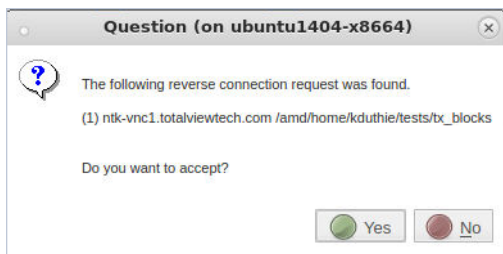
When TotalView launches, it automatically listens for reverse connection requests, briefly displaying a notice:



Listening for Reverse Connections

NOTE: TotalView automatically listens for connections at launch *only when invoked without specifying a debug target*. If TotalView starts debugging a different program, it does not automatically listen for reverse connections.

If one or more requests are found, it then launches a pop-up to confirm that you want to accept the reverse debugging request:



If you select “No,” the back-end **tvconnect** stops waiting and exits with the following error message:
“Reverse connect request was rejected.”

If you select “Yes,” your program to debug launches in the usual way, and you can start your debugging session.

Note that the UI displays the node where TotalView is running in parenthesis in the title bar:

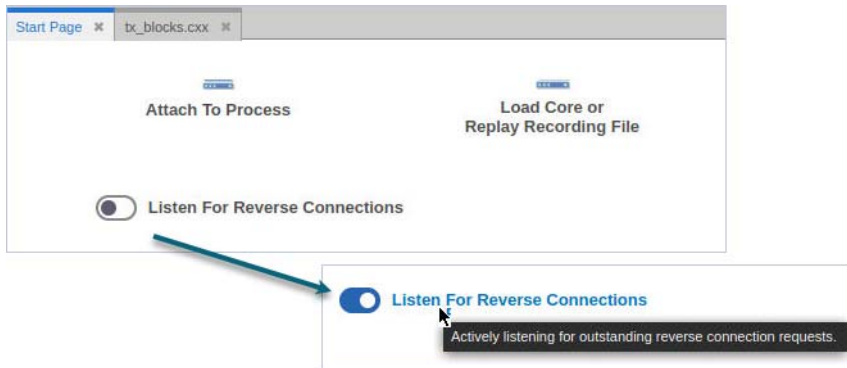
TotalView for HPC 2019 (on ubuntu1404-x8664)

Listening for Reverse Connections

Once you start a debugging session, TotalView automatically *stops* listening for reverse connection requests.

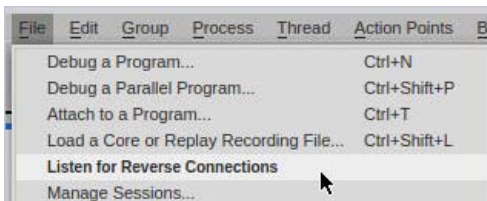
To turn back on listening mode, you have two options. Either:

- Go to the Start Page and toggle on the “Listen for Reverse Connections” switch:



or

- Choose the “Listen for Reverse Connections” option from the File menu:



Reverse Connect Examples

Initiate a reverse connection request by specifying any debug target program as the argument to **tvconnect**. The specified program must be accessible by the TotalView front-end UI that wishes to accept the request.

Here's a simple example:

```
tvconnect /home/totalview/tests/myTest
```

To start it on an MPI job, for example:

```
tvconnect srun -n 4 /home/fullpath/tx_mpi_test
```

CLI Example

This example illustrates the usage of the reverse connection **dload** options.

Assume that:

- **tvconnect** was started on machine1 specifying a program with a full path
- **tvconnect** was started on machine2 specifying the program **tx_hello**
- The program was in the current working directory and no path was added to the program.

```
d1.<> dload -list_reverse_connect
(1) machine1.totalviewtech.com /home/user/tests/tx_blocks
(2) machine2.totalviewtech.com tx_hello

d1.<> dload -accept_rc 1
d1.<> dload -reject_rc 2
```

MPI Batch Script Example

This is a simple example for invoking an MPI job from a batch script.

1. Create your job script. For example, create a batch script containing the following:

```
#-----
#!/bin/tcsh
#SBATCH -p pdebug
#SBATCH -J myJob
#SBATCH -N 2
# Wait for a front end TV to accept this reverse connection request
tvconnect srun -n4 myMPIprogram
echo 'DONE!'
#-----
```

Once the script is run, the **tvconnect** command creates the request file with the necessary details, then holds and waits for a connection request.

2. Start TotalView on your front-end node and accept the request. The debugger begins debugging **srun**. Pressing **Go** starts the MPI job and TotalView will attach to the MPI processes running on the compute nodes in the normal way.

NOTE: If the application is in your system path, TotalView will find it. i.e., you do not need to enter the full path in your command.

it is not required that you include the full path to your application in the command, if the application is in your path. In the above case, `myMPIprogram` was in the current working directory when the batch job was submitted. The request file reports the current working directory, so that the front-end TotalView can find the application even if it was not started from same directory.

Troubleshooting Reverse Connections

Most of the issues that can result in a failed reverse connection have to do with the reverse connect directory where TotalView writes and reads connection requests.

Stale Files in the Reverse Connect Directory

In some cases, TotalView may leave stale request or response files in the reverse connect directory, which could result in a failed connection attempt.

If you have no pending reverse connect requests, it is safe to remove the entire directory or its contents. For example, use `rm -rf $HOME/.totalview/connect` to remove the default directory and all its files. The **tvconnect** process will recreate the directory when needed.

Directory Permissions

You must be the owner of the reverse connect directory, and its permissions must allow "user" access only. The directory cannot be owned by a different user, and cannot have any "group" or "other" permission bits set.

User ID Issues

The **tvconnect** process and the TotalView client must be running with the same effective user id (**eu**id). The **eu**id of the processes run by the client must match that of the reverse connect directory.

Reverse Connect Directory Environment Variable

If the TotalView client fails to find a **tvconnect** request, make sure that the **tvconnect** process is writing its request file to the same directory being read by the TotalView client.

For example, the client may read the wrong directory if your home directory on the node where **tvconnect** is running is different than the node where the TotalView client is running.


To work around this problem, set the environment variable **TV_REVERSE_CONNECT_DIR**.

When setting this variable, make sure that it points to a directory accessible by both the **tvconnect** and TotalView client processes, and that it meets all the ownership and permission requirements.

Preferences

- About Preferences
- Action Points
- Display Settings
- Tool Bar
- Search Path
- Parallel Configuration
- Signal Actions
- Remote Connection Settings

About Preferences

Customize aspects of TotalView using the Settings toolbar  or by selecting **File | Preferences**, including:

- Action Points
- Display Settings
- Tool Bar
- Search Path
- Parallel Configuration
- Remote Connection Settings
- Signal Actions

Preferences take effect after TotalView restarts.

DISPLAY

Display Settings
Customize user interface display settings.

Appearance
Choose the best interface style for your development.

Light Dark

User Interface Style
Choose the type of user interface.

New Interface
Modern, dockable style user interface with improved low to medium scale multi-process and multi-thread dynamic analysis and debugging.

Classic Interface
Traditional, dedicated window for very high-scale multi-process dynamic analysis and debugging.

Font Size
Choose the font size for the user interface.

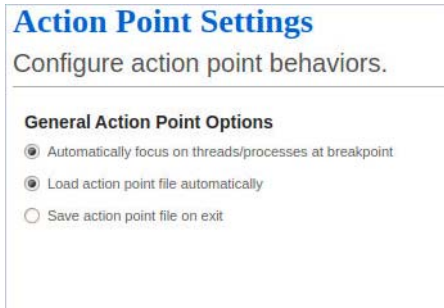
Small Medium Large

Display Settings changes will take effect the next time the product is started

Action Points

The Action Points tab controls action point behavior.

Figure 98, Preferences: Action Points



Action Point Options:

- **Automatically focus on threads/processes at breakpoint**

If selected, TotalView automatically focuses on the active thread and/or process when your program reaches a breakpoint. Unlike other action point preferences, this preference changes how existing action points behave. For more information, see the **TV::GUI::pop_at_breakpoint** variable in the *TotalView Reference Guide*.

- **Load action point file automatically**

When set, TotalView automatically loads action points when it loads a file. For more information, see the Action Point > Load All command and the **TV::auto_load_actionpoints** variable in the *TotalView Reference Guide*.

- **Save action point file on exit**

When set, TotalView automatically saves action points to an action points file when you exit. For more information, see the Action Point > Save All command and the **TV::auto_save_breakpoints** variable in the *TotalView Reference Guide*.

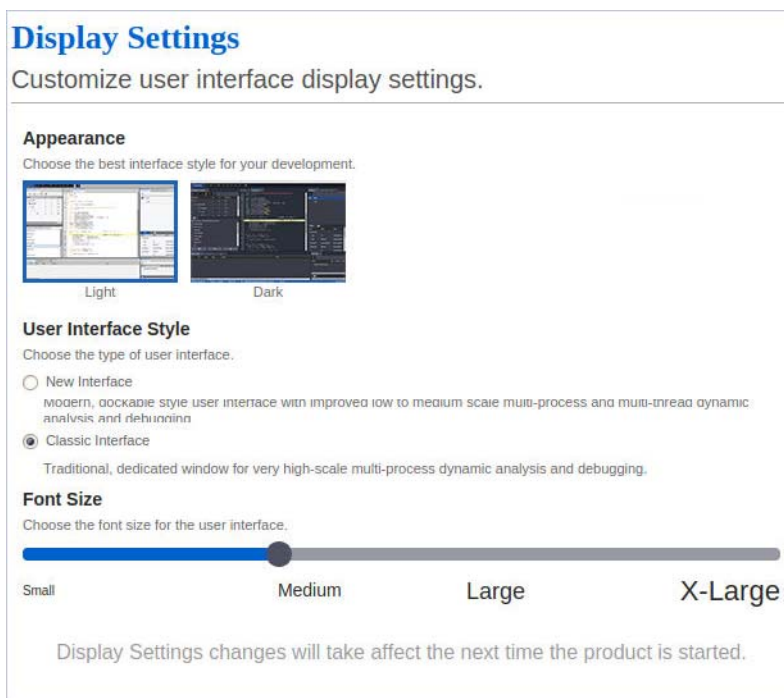
Display Settings

The Display tab manages display settings.

- **Appearance:** light or dark theme
- **User Interface Style:** new UI or Classic
- **Font Size:** small to extra large

NOTE: These preference settings require a restart.

Figure 99, Preferences: Display Settings



Appearance

- **Light:** Light-colored-background with dark text
- **Dark:** Dark-colored background with light text

NOTE: This setting is overridden if you provide the option `-theme light/dark` when starting TotalView.

User Interface Style

- **New Interface:** Modern, dockable style user interface with improved low to medium scale multi-process and multi-thread dynamic analysis and debugging.
- **Classic Interface:** Traditional, dedicated window for very high-scale multi-process dynamic analysis and debugging.

NOTE: The User Interface Style preference is ignored if you start TotalView using the command line option `-newUI` or have set the `TVNEWUI` environment variable.

Font Size

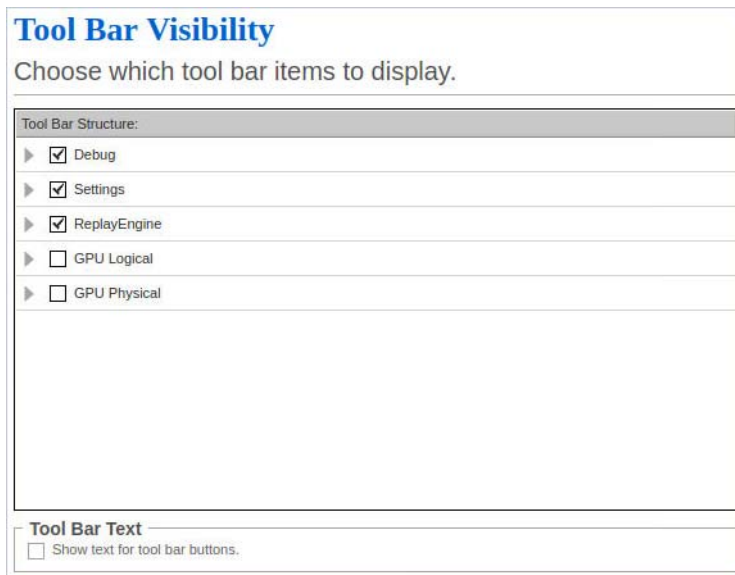
Adjust the slider to change the font size for the UI from small to extra large. The default is medium.

Tool Bar

This tab controls toolbar display. You can also display or hide the toolbars by right-clicking in the menu/toolbar area and selecting its toggle menu items.

Use the checkbox under **Tool Bar Text** to show or hide toolbar text.

Figure 100, Preferences: Tool Bar Visibility



Search Path

The Search Path tab customizes the locations in which TotalView searches for executables, source files, and object files.

TotalView uses a number of system variables to determine where to look for source files, executables, and object files. You can see these variables in the Command Line view by entering `dset *PATH*` to display all variables with "PATH" in the name:

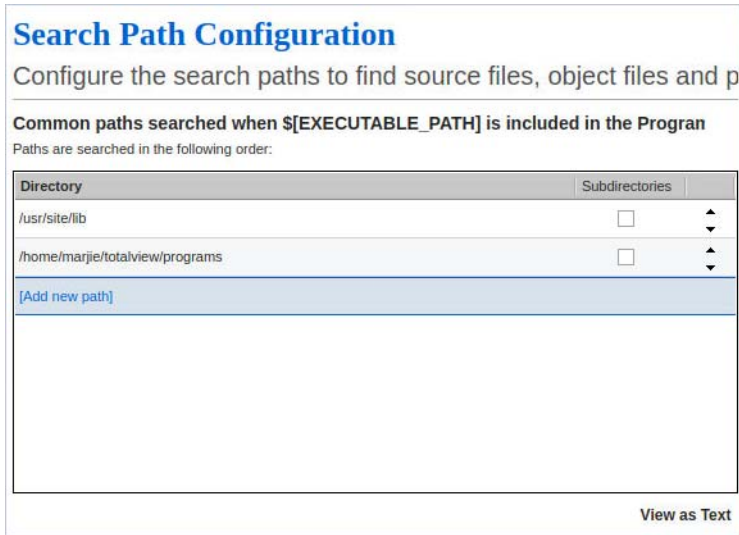
```
d1.<> dset *PATH*
EXECUTABLE_PATH {}
EXECUTABLE_SEARCH_PATH {{EXECUTABLE_PATH}}:{{PATH}}:.}
OBJECT_SEARCH_PATH {{COMPILATION_DIRECTORY}}:{{EXECUTABLE_PATH}}:{{EXECUTABLE_DIRECTORY}}:
$link({EXECUTABLE_DIRECTORY}):.:$TOTALVIEW_SRC}
SHARED_LIBRARY_SEARCH_PATH {{EXECUTABLE_DIRECTORY}}
SOURCE_SEARCH_PATH {{COMPILATION_DIRECTORY}}:{{EXECUTABLE_PATH}}:{{EXECUTABLE_DIRECTORY}}:
$link({EXECUTABLE_DIRECTORY}):.:$TOTALVIEW_SRC}
...
```

Notice that `EXECUTABLE_PATH` is initially undefined, and that it is included in the other search paths (except for `SHARED_LIBRARY_SEARCH_PATH`).

NOTE: You should *not* remove `EXECUTABLE_PATH` from the other variables or this preferences mechanism will not work.

If you have additional locations where you want TotalView to search for source files, executables, and objects files, you can define `EXECUTABLE_PATH` in the Preferences dialog.

Figure 101, Preferences: Search Path Configuration



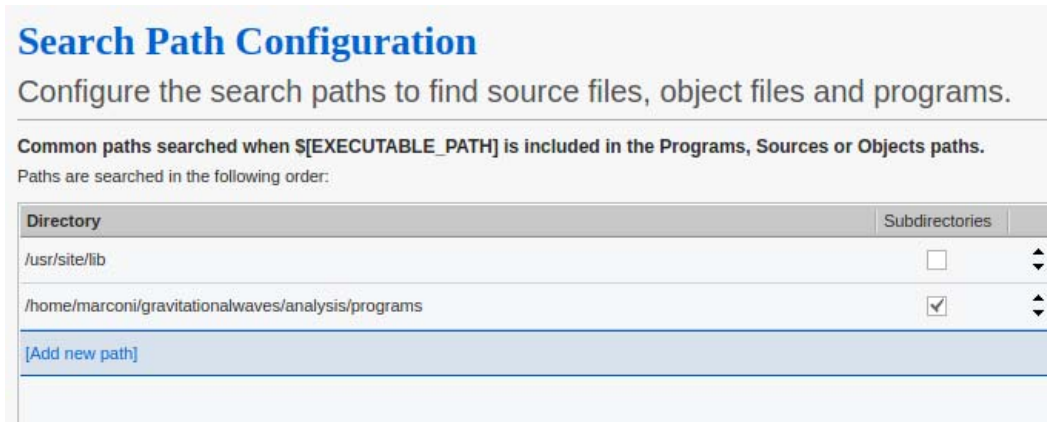
To add a path, click on **Add new path** and start typing. The interface accepts only one path per line. If you type in two or more paths separated by colons, they are divided into separate lines when you click Apply. Click the Subdirectories box to search recursively beginning at the specified directory as root. Use the up and down icons to adjust the search order.

NOTE: Long search paths have a tendency to slow performance as TotalView frequently traverses the search path to create an accurate target environment.

Selecting the Subdirectories checkbox if the root directory is at the top of a deep directory tree adds multiple paths to the search list, potentially degrading performance.

When you are done, click **Apply** to save the changes, or **OK** to save and close the dialog.

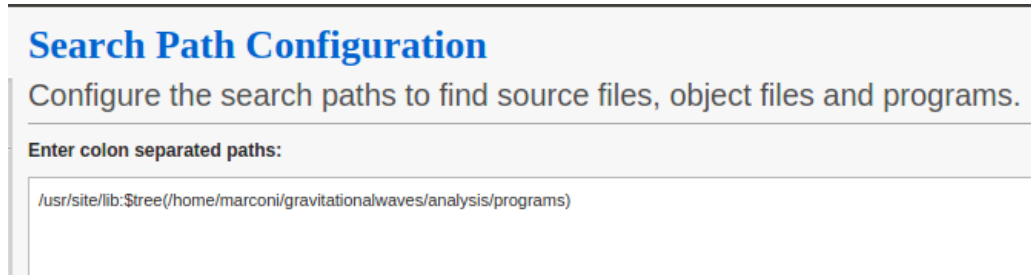
Figure 102, Preferences: Search Path Configuration, Add a New Path



To edit an entry, double-click on the entry to engage editing mode. To delete an entry, right-click on the entry, which brings up a context menu with a delete function.

The View as Text button shows you the `EXECUTABLE_PATH` variable as it would appear in the command line interface.

Figure 103, Preferences: Search Path Configuration, Edit an Entry



This view is also editable, allowing you to add, remove, or reorder paths as you would on the command line. Notice that the path specified for recursive search is enclosed in the `$tree()` function, which automatically adds the paths below the specified root.

Parallel Configuration

This tab allows you to configure the parallel attach behaviors for debugging sessions, choosing:

- How to attach to started processes
- Whether processes should continue running or should stop after attaching
- **dbfork** options

NOTE: You can start MPI jobs in two ways. You can directly invoke TotalView on your program (which is identical to entering arguments into the [Parallel Session Dialog](#) or other types of debugging sessions available from the File menu) or by directly or indirectly involving a starter program such as **poe** or **mpirun**. TotalView refers to these configuration settings only when it is directly invoked on a starter program. For programs started by TotalView, the program actually executes in the same way as a non-parallel program. That is, all processes created are part of the same control group, and TotalView allows this control group to run freely.

Figure 104, Preferences: Parallel Configuration

Parallel Configuration

Configure the parallel attach behaviors for debugging sessions.

Attach Behavior
Select how the debugger attaches to started processes.

Ask what to do

Attach to all processes

Attach to none of the processes

After Attach Behavior
Select the action after attaching to the processes

Ask what to do

Stop the processes

Continue the processes

dbfork Attach Behavior
Select how to attach to forked processes when using TotalView's dbfork helper library.

Attach to forked processes linked with dbfork

Do not attach to forked processes linked with dbfork

- **Attach Behavior:** Choose the action to take when processes begin execution. Either automatically attach to all executing processes, attach to none, or launch a pop-up asking what to do. The default is “Attach to all processes.”
- **After Attach Behavior:** Choose the action to take after attaching to started processes. Either stop the processes to easily set breakpoints, launch a pop-up dialog asking what to do, or continue running the processes. The default is “Ask what to do.”
- **dbfork Attach Behavior:** Choose whether or not the debugger attaches to child processes linked with **dbfork**. The default is “Attach.”

RELATED TOPICS

Linking with the dbfork Library

[Linking with the dbfork Library](#) and the **TV::dbfork** variable

More about controlling how TotalView attaches to processes

The **TV::parallel_attach** variable in the *TotalView Reference Guide*

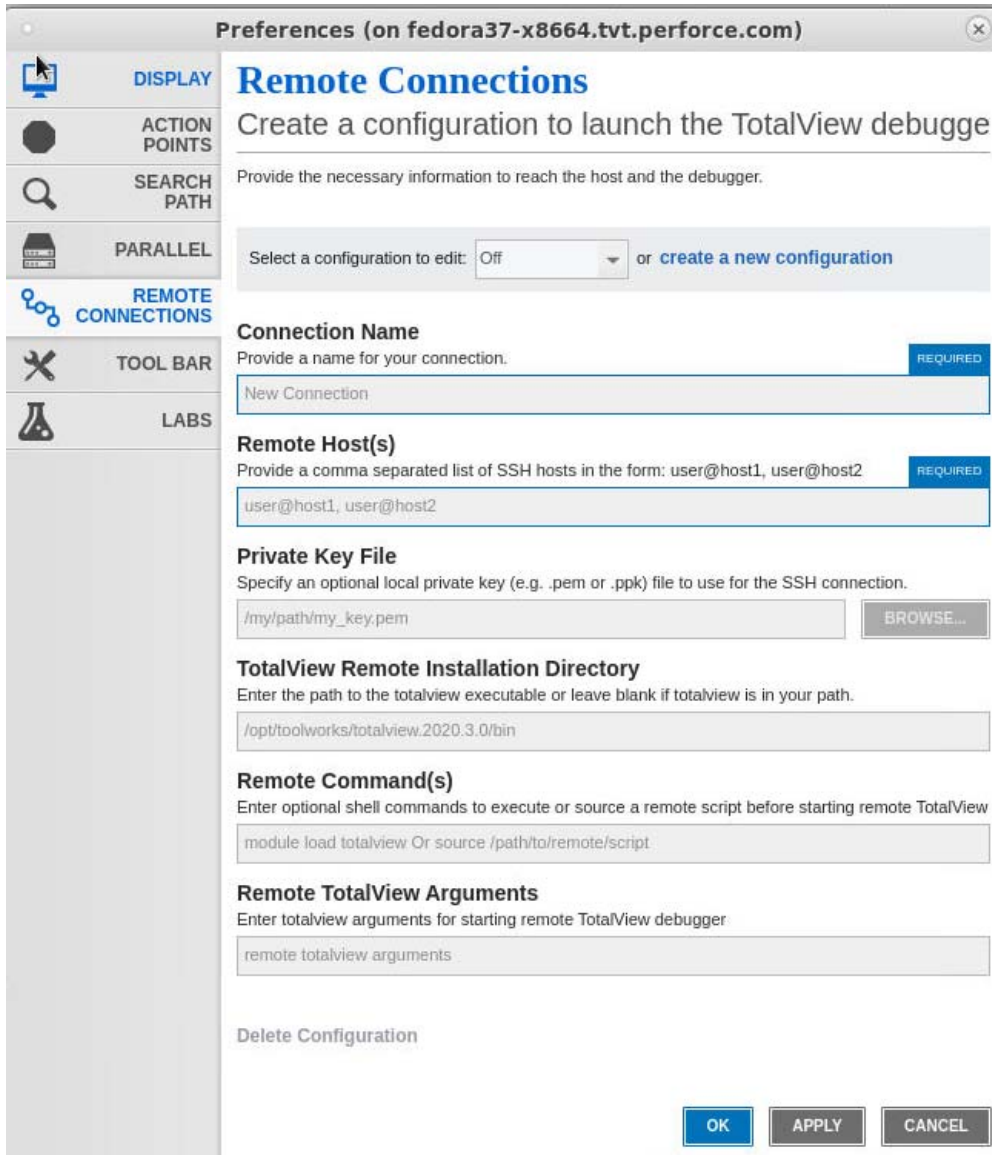
More about how TotalView stops processes.

The **TV::parallel_stop** variable in the *TotalView Reference Guide*

Remote Connection Settings

This tab configures remote connections.

Figure 105, Preferences: Remote Connection Settings



For details, see [TotalView Remote Connections](#).

Signal Actions

The **Signals** tab customizes the way TotalView handles signals. See [Default Signal Handling](#) for the default settings.

Figure 106, Preferences: Signal Action Settings

Signal Action Settings
Configure signal handling behaviors.

Search...

Name	Number	Action	Description
SIGHUP	1	<input type="radio"/> Error <input type="radio"/> Stop <input checked="" type="radio"/> Resend <input type="radio"/> Ignore	Hangup
SIGINT	2	<input type="radio"/> Error <input checked="" type="radio"/> Stop <input type="radio"/> Resend <input type="radio"/> Ignore	Interrupt (rubout)
SIGQUIT	3	<input type="radio"/> Error <input type="radio"/> Stop <input checked="" type="radio"/> Resend <input type="radio"/> Ignore	Quit (ASCII FS)
SIGILL	4	<input checked="" type="radio"/> Error <input type="radio"/> Stop <input type="radio"/> Resend <input type="radio"/> Ignore	Illegal instruction (not reset when caught)
SIGTRAP	5	[Reserved by Totaview]	Trace trap (not reset when caught)
SIGABRT	6	<input checked="" type="radio"/> Error <input type="radio"/> Stop <input type="radio"/> Resend <input type="radio"/> Ignore	Abort process
SIGBUS	7	<input checked="" type="radio"/> Error <input type="radio"/> Stop <input type="radio"/> Resend <input type="radio"/> Ignore	Bus error
SIGFPE	8	<input checked="" type="radio"/> Error <input type="radio"/> Stop <input type="radio"/> Resend <input type="radio"/> Ignore	Floating point exception
SIGKILL	9	<input type="radio"/> Error <input type="radio"/> Stop <input checked="" type="radio"/> Resend <input type="radio"/> Ignore	Kill (cannot be caught or ignored)
SIGUSR1	10	<input type="radio"/> Error <input type="radio"/> Stop <input checked="" type="radio"/> Resend <input type="radio"/> Ignore	User defined signal 1
SIGSEGV	11	<input checked="" type="radio"/> Error <input type="radio"/> Stop <input type="radio"/> Resend <input type="radio"/> Ignore	Segmentation violation
SIGUSR2	12	<input type="radio"/> Error <input type="radio"/> Stop <input checked="" type="radio"/> Resend <input type="radio"/> Ignore	User defined signal 2
SIGPIPE	13	<input checked="" type="radio"/> Error <input type="radio"/> Stop <input type="radio"/> Resend <input type="radio"/> Ignore	Write on a pipe with no one to read it
SIGALRM	14	<input type="radio"/> Error <input type="radio"/> Stop <input checked="" type="radio"/> Resend <input type="radio"/> Ignore	Alarm clock
SIGTERM	15	<input checked="" type="radio"/> Error <input type="radio"/> Stop <input type="radio"/> Resend <input type="radio"/> Ignore	Software termination signal from kill
SIGSTKFLT	16	<input type="radio"/> Error <input type="radio"/> Stop <input checked="" type="radio"/> Resend <input type="radio"/> Ignore	Stack Fault

Reset

The default behavior for each signal is preselected in the Action column. (The list of signal names and numbers is specific to your operating system.)

You can also set signal action values in the CLI using the state variable `TV::signal_handling_mode` or the command-line option `-signal_handling_mode`.

NOTE: The way TotalView handles signals is impacted by the system on which you are running the debugger. For example, on some systems, hardware registers affect how TotalView and your program handle signals such as **SIGFPE**. For more information, see [Architectures](#) in the *TotalView Reference Guide*.

Table 9: Signal Handling Actions

Action	Description
Error	Stops the process, places it in the error state, and displays an error. Select this action for severe error conditions, such as SIGSEGV and SIGBUS. (TotalView stops all related processes by default. To change this behavior, set the state variable <code>TV::stop_relatives_on_proc_error</code> to false .)
Stop	Stops the process and places it in the stopped state. Select this action to handle this signal as a SIGSTOP signal.
Resend	Sends the signal back to the process. This setting is useful when testing your program's signal handling routines. TotalView sets certain signals to Resend by default because most programs have handlers to handle program termination.
Ignore	Discards the signal and continues the process. The process receives no notification that a signal was generated. Caution: Setting a fatal signal such as SIGBUS or SIGSEGV to Ignore can result in a signal/re-signal loop.

RELATED TOPICS

- Using the CLI to set signal actions. See the `TV::signal_handling_mode` state variable, which you can add to a `.tvdrc` startup file. Alternatively, set a signal action for a single TotalView session using the command-line option `-signal_handling_mode`.
- Controlling whether all related processes stop for an error setting. See the state variable `TV::stop_relatives_on_proc_error` in the *TotalView Reference Guide*.
- How a system's hardware registers may impact the way TotalView and your program handle SIGFPE. See [Architectures](#) in the *TotalView Reference Guide*.

Default Signal Handling

Table 10 lists how TotalView handles UNIX signals by default. The table lists most common signals, but does not list every possible signal produced by all the operating systems TotalView supports.

Table 10: Default UNIX Signal Handling

Signal	Error	Stop	Resend	Ignore
SIGABRT	x			
SIGALRM			x	
SIGBUS	x			
SIGCHLD			x	
SIGCONT			x	
SIGFPE	x			
SIGHUP			x	
SIGILL	x			
SIGINT		x		
SIGIO			x	
SIGKILL			x	
SIGPIPE	x			
SIGPROF			x	
SIGPWR	x			
SIGQUIT			x	
SIGRTMIN			x	
SIGSEGV	x			
SIGSTKFLT			x	
SIGSTOP <i>(Reserved by TotalView)</i>	x			
SIGSYS	x			
SIGTERM	x			
SIGTRAP <i>(Reserved by TotalView)</i>	x			
SIGTSTP		x	x	
SIGTTIN		x	x	

Table 10: Default UNIX Signal Handling

Signal	Error	Stop	Resend	Ignore
SIGTTOU		x	x	
SIGURG			x	
SIGUSR1			x	
SIGUSR2			x	
SIGVTALRM			x	
SIGWINCH			x	
SIGXCPU	x			
SIGXFSZ	x			

PART III Parallel Debugging

- **About Parallel Debugging in TotalView**

An overview of some of TotalView's features that support parallel debugging

- **Setting Up Parallel Sessions**

How to set up MPI and other parallel sessions in TotalView

- **Debugging OpenMP Applications**

Configuring TotalView for debugging OpenMP applications, using the OpenMP view, and filtering the call stack.

- **Controlling fork, vfork, and execve Handling**

Controlling how TotalView handles system calls to **execve()**, **fork()**, **vfork()**, and **clone()**.

- **Group, Process, and Thread Control**

The TotalView process/thread model and how to manage debugging multiple processes and threads.

- **Scalability in HPC Computing Environments**

TotalView's features and configurations related to scalability.

About Parallel Debugging in TotalView

TotalView is designed to debug multi-process, multi-threaded programs, with a rich feature set to support fine-grained control over individual or multiple threads and processes. This level of control makes it possible to quickly resolve problems like deadlocks or race conditions in a complex program that spawns thousands of processes and threads across a broad network of computer nodes.

When your program creates processes and threads, TotalView can automatically bring them under its control, whether they are local or remote. If the processes are already running, TotalView can attach to them as well, avoiding the need to run multiple debuggers.

TotalView places a debugger server process on each remote node as it is launched that then communicates with the main TotalView client process. This debugging architecture gives you a central location from which you can manage and examine all aspects of your program.

This chapter introduces some tools and features that support parallel debugging.

- [Parallel Program Execution Models](#)
- [Viewing Process and Thread State](#)
- [Controlling Program Execution](#)
- [Configuring TotalView for Parallel Debugging](#)

Parallel Program Execution Models

TotalView supports the popular parallel execution models including MPI, OpenMP, SGI shared memory (shmem), Global Arrays, UPC, CAF, fork/exec, and pthreads.

RELATED TOPICS

Starting a parallel debugging session	Debug a Parallel Program
OpenMP applications	Debugging OpenMP Applications
Hybrid OpenMP/MPI programs	Hybrid Programming: Combining OpenMP with MPI
Setting up parallel programming sessions	"Setting Up Parallel Debugging Sessions" and "Setting Up MPI Debugging Sessions" in the <i>Classic TotalView User Guide</i>.

Viewing Process and Thread State

You can quickly view process and thread state in the Process and Threads view.

The Process and Threads view displays all processes and threads being debugged, along with their process state (i.e., stopped, running, at breakpoint, etc.). Selecting a process or thread sets the focus, which then determines the display in the Call Stack, Local Variables view, the Source view, and the Data View.

Figure 107, The Process and Threads view

Description	# P	# T	Members
tx_fork_loop (S3)	4	4	p1-4
Mixed	1	1	p1
<unknown address>	1	1	p1.1
1.1	1	1	p1.1
Running	1	1	p1.1
__clone	1	2	p1.2-3
1.2	1	1	p1.2
Stopped	1	1	p1.2
1.3	1	1	p1.3
Stopped	1	1	p1.3
Breakpoint	3	3	p2-4
snore	3	3	p3.1, p2.2, p4.2
2.2	1	1	p2.2
Breakpoint	1	1	p2.2
3.1	1	1	p3.1
Breakpoint	1	1	p3.1
4.2	1	1	p4.2
Breakpoint	1	1	p4.2
__clone	3	6	p2.1, p4.1, p3.2, p2-4.3

RELATED TOPICS

Controlling individual threads and processes to analyze and isolate problems [Group, Process, and Thread Control](#)

The Process and Threads view

[Processes & Threads View Basics](#)

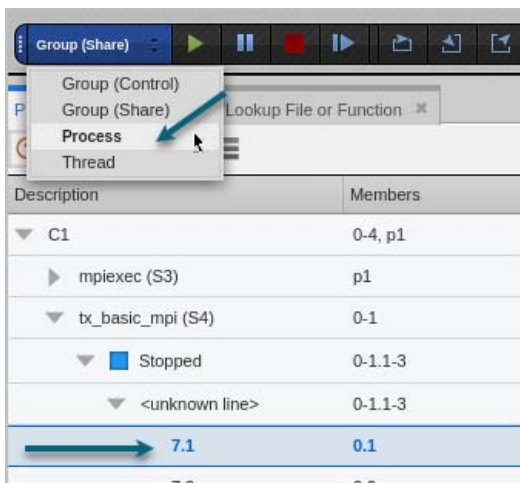
Controlling Program Execution

Commands controlling execution operate on the current focus — either an individual thread or process, or a group of threads and processes. You can individually stop, start, step, and examine any thread or process, or perform these actions on a group.

In the UI, the focus of any execution command is the thread selected in the Processes and Threads view, and the width control selected from the **Focus** menu.

In [Figure 108](#), the focus is on thread 0.1, and the width selected in the Focus menu is Process.

Figure 108, Selecting the focus in the UI



In the CLI, use the **dfocus** command with an *arena specifier* to define the thread of interest, the process of interest, the width, and optionally, a group on which to operate.

You can also synchronize execution across threads or processes using a *barrierpoint*, which holds any threads or processes in a group until each reaches a particular point.

RELATED TOPICS

The dfocus command	dfocus in "CLI Commands" in the <i>TotalView Reference Guide</i>
Focus and execution commands	Individual Execution Commands
Setting the focus to Group (Share) to run just a single share group	Executing a Single Share Group
Using the CLI to control program execution	Controlling Program Execution Using CLI Commands

RELATED TOPICS

Holding an individual process or thread to synchronize control execution	Holding and Releasing Processes and Threads
Introduction to barrier points	Synchronizing Execution with Barrier Points
Finely controlling focus using arenas	Arenas and P/T Sets

TotalView Groups

TotalView automatically organizes your processes and threads into groups, allowing you to view, execute, and control any individual thread, process, or group of threads and processes. TotalView defines built-in groups that help support full, asynchronous debugging control over your program.

For example, you can:

- Single step one or a small set of processes rather than all of them
- Use **Group > Detach** and **Process > Detach** to isolate certain processes or groups and remove them from debugging control.
- Use **Run To** or breakpoints to control large groups of processes
- Control breakpoints when using the **fork()** or **execve()** functions
- Share action points across multiple processes or set them in individual processes

RELATED TOPICS

Using Run To to control stepping and execution	Synchronizing Processes and Threads
Setting the focus to Group (Share) to run just a single share group	Executing a Single Share Group
Control breakpoints when using fork() or execve()	Setting Breakpoints When Using the fork()/execve() Functions
Control an action point's scope, or <i>width</i> , to stop a group of processes, a single process or a single thread	Controlling an Action Point's Width

Synchronizing Execution with Barrier Points

You can synchronize execution of threads and processes either manually using a *hold* command, or automatically by setting an action point called a *barrierpoint*. These two tools can be used together for fine-grained execution control. For instance, if a process or thread is held at a barrier point you can manually release it and then run it without first waiting for all other processes or threads in the group to reach that barrier.

When a process or a thread is *held*, it ignores any command to resume executing. For example, assume that you place a hold on a process in a control group that contains three processes. If you select **Group > Go**, two of the three processes resume executing. The held process ignores the **Go** command.

RELATED TOPICS

[Setting barrier points](#)

[Setting a Barrier Breakpoint](#)

Configuring TotalView for Parallel Debugging

TotalView provides features and performance enhancements for scalable debugging in today's HPC computing environments. In most cases, these require no special configuration. You can, however, customize a debugging session when working with multiprocess/multithreaded applications.

For example, you can configure how TotalView handles a **dlopen** event to support better performance if multiple shared libraries are being opened, or the debugger is processing multiple processes. You can also customize how TotalView handles system calls to **execve()** and **fork()**, in addition to other enhancements.

RELATED TOPICS

Configuring TotalView for scalability

[Scalability in HPC Computing Environments](#)

Configuring **dlopen** system calls

[dlopen Options for Scalability in the *TotalView Reference Guide*](#)

Setting Up Parallel Sessions

You can set up parallel debugging or MPI sessions using the UI's Session Editor or from a shell, depending on your system and its requirements.

NOTE: If you are using TotalView Developer / Developer for HPC, all your program's processes must execute on the computer on which you installed TotalView.

Included are these topics:

- Parallel Program Setup in the UI
- Non-MPI Program Setup
 - The SLURM Resource Manager
 - Cray XT/XE/XK/XC Applications
 - Global Arrays Applications (**Classic UI Only**)
 - Shared Memory (SHMEM) Code
 - UPC Programs
 - CoArray Fortran (CAF) Programs
- MPI Program Setup
 - MPICH Applications
 - MPICH2 Applications
 - Cray MPI Applications
 - IBM MPI Parallel Environment (PE) Applications
 - Open MPI Applications
 - QSW RMS Applications

This chapter also includes these topics specific to MPI applications:

[Troubleshooting MPI Startup](#)

[Using ReplayEngine with Infiniband MPIs](#)

[MPI Startup Customizations](#)

Parallel Program Setup in the UI

To set up a parallel session in the UI, from the Start Page, select **Debug a Parallel Program** to launch the Parallel Session dialog. See [Debug a Parallel Program](#).

Programs started using the UI have some limitations: program launch does not use the information you set for single-process and bulk server launching, and you cannot use the Attach Subset command.

MPI programs use a starter program such as **mpirun**. You can start these MPI programs in two ways: with the starter program under TotalView control, or using the UI, in which case the starter program is not under TotalView control.

Non-MPI Program Setup

This section discusses parallel computing models that are not MPI-based.

- The SLURM Resource Manager
- Cray XT/XE/XK/XC Applications
- Global Arrays Applications (**Classic UI Only**)
- Shared Memory (SHMEM) Code
- UPC Programs
- CoArray Fortran (CAF) Programs

The SLURM Resource Manager

TotalView supports the SLURM resource manager. Here is some information copied from the SLURM website (<https://hpc.llnl.gov/documentation/tutorials/livermore-computing-linux-commodity-clusters-overview-part-one>).

SLURM is an open-source resource manager designed for Linux clusters of all sizes. It provides three key functions. First it allocates exclusive and/or non-exclusive access to resources (computer nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (typically a parallel job) on a set of allocated nodes. Finally, it arbitrates conflicting requests for resources by managing a queue of pending work.

SLURM is not a sophisticated batch system, but it does provide an Applications Programming Interface (API) for integration with external schedulers such as the Maui Scheduler. While other resource managers do exist, SLURM is unique in several respects:

- Its source code is freely available under the GNU General Public License.
- It is designed to operate in a heterogeneous cluster with up to thousands of nodes.
- It is portable; written in C with a GNU autoconf configuration engine. While initially written for Linux, other UNIX-like operating systems should be easy porting targets. A plugin mechanism exists to support various interconnects, authentication mechanisms, schedulers, etc.
- SLURM is highly tolerant of system failures, including failure of the node executing its control functions.
- It is simple enough for the motivated end user to understand its source and add functionality.

Cray XT/XE/XK/XC Applications

The Cray XT/XE/XK/XC series of supercomputers are supported by the TotalView Linux x86_64 and Linux ARM64 (aarch64) distributions. The discussion here is based on running applications using the Cray Linux Environment (CLE). TotalView supports launching application program

s using either PBS Pro with ALPS **aprun** or SLURM **srun**.

RELATED TOPICS

Tips for parallel debugging

“General Parallel Debugging Tips” in the *Classic TotalView User Guide*'s chapter “Debugging Strategies for Parallel Applications.”

Setting up a parallel debugging session using the Session Editor [Debug a Parallel Program](#)

Starting TotalView on Cray

Because the configuration of most Cray systems typically varies from site to site, the following provides only general guidelines for starting TotalView on your application. Please consult your site's documentation for the specific steps needed to debug a program using TotalView on your Cray system.

File System Considerations

Place your application to debug on a file system that is shared across all Cray node types, such as the service, elogin, login/MOM, and/or compute nodes. This allows you to compile and debug your application across node types.

Further, make sure that your `$HOME/.totalview` directory is on a shared file system that is common across all Cray node types to ensure that you can use the **tvconnect** feature in batch scripts. For more information, see [Reverse Connections](#).

Starting TotalView

TotalView typically runs interactively. If your site has not designated any compute nodes for interactive processing, you can allocate compute nodes for interactive use. Use PBS Pro's **qsub -I** or SLURM's **salloc** command to allocate an interactive job. Be sure that your X11 **DISPLAY** environment variable is propagated or set properly. See "man qsub" or "man salloc" for more information on interactive jobs.

If TotalView is installed on your system, load it into your user environment:

```
module load totalview
```

Use the following command to start TotalView where *mpi_starter* is the MPI starter program for your system, such as **aprun** or **srun**.

The CLI:

```
totalviewcli -tv_options -args mpi_starter [mpi_options] application_name
[application_arguments]
```

The GUI:

```
totalview -tv_options -args mpi_starter [mpi_options] application_name [application_arguments]
```

TotalView is not able to stop your program before it calls **MPI_Init()** when using ALPS. While this is typically at the beginning of **main()**, the actual location depends on how you've written the program. This means that if you set a breakpoint before the **MPI_Init()** call, your program will not hit it because the statement upon which you set the breakpoint will have already executed. On the other hand, SLURM will stop your program before it enters **main()**, which allows you to debug the statements before **MPI_Init()** is called.

Example 1: Interactive Jobs Using qsub and aprun

This example shows how you can start TotalView on a program named **a.out** running in an interactive job using **qsub** and **aprun**.

```
n9610@crystal:~> qsub -I -X -l nodes=4
qsub: waiting for job 599856.sdb to start
qsub: job 599856.sdb ready

Directory: /home/users/n9610
Mon Nov  4 17:23:28 CST 2019
n9610@crystal2:~> cd shared/rctest
n9610@crystal2:~/shared/rctest> module load totalview
n9610@crystal2:~/shared/rctest> totalview -verbosity errors -args aprun -n 4 ./a.out
```

Example 2: Interactive jobs using salloc and srun

Similarly, you can debug an interactive job using **salloc** and **srun** if your Cray system uses SLURM.

This example shows how to submit a SLURM batch job using **tvconnect** in the batch script. After the batch job starts running, TotalView is started to accept the reverse-connect request.

```
n9610@jupiter-elogin:~/shared/rctest> cat slurm-script.bash
#!/bin/bash -x
#SBATCH --qos=debug
#SBATCH --time=00:30:00
#SBATCH --nodes=4
#SBATCH --tasks-per-node=1
#SBATCH --constraint=haswell

module load totalview
tvconnect srun -n 4 tx_basic_mpi
n9610@jupiter-elogin:~/shared/rctest> sbatch -C BW28 slurm-script.bash
Submitted batch job 1374150
n9610@jupiter-elogin:~/shared/rctest> squeue -u $USER

  JOBID  USER ACCOUNT          NAME ST REASON   START_TIME           TIME  TIME_LEFT  NODES CPUS
1374150  n9610  (null) slurm-script.b  R  None    2019-11-05T09:31:53  0:16    29:44     4    8

n9610@jupiter-elogin:~/shared/rctest> module load totalview
n9610@jupiter-elogin:~/shared/rctest> totalview
```


Support for Cray Abnormal Termination Processing (ATP)

Cray's ATP module stops a running job at the moment it crashes. This allows you to attach TotalView to the held job and begin debugging it. To hold a job as it is crashing you must set the `ATP_HOLD_TIME` environment variable before launching your job with `aprun` or `srun`.

When your job crashes, the MPI starter process outputs a message stating that your job has crashed and that ATP is holding it. You can now attach TotalView to the `aprun` or `srun` process using the normal attach procedure (see [Attach to Process](#)).

For more information on ATP, see the Cray [intro_atp](#) man page.

Special Requirements for Using ReplayEngine

On Crayx86_64 systems, the MPIs use RDMA techniques, similar to Infiniband MPIs. When using ReplayEngine on MPI programs, certain environment variable settings must be in effect for the MPI rank processes. These settings ensure that memory mapping operations are visible to ReplayEngine. The required environment variable settings are:

- `MPICH_SMP_SINGLE_COPY_OFF=1`
- `LD_PRELOAD`: Set to include a preload library, which can be found under the TotalView installation directory at `toolworks/totalview.<version>/linux-x86-64/lib/undodb_infiniband_preload_x64.so`.

When using APLS, these settings may be applied with the `aprun -e` option. For example, to have TotalView launch an MPI program with ReplayEngine enabled, use a command similar to this:

```
totalview -replay -args aprun -n 8 \
-e MPICH_SMP_SINGLE_COPY_OFF=1 \
-e LD_PRELOAD=/<path>/undodb_infiniband_preload_x64.so \
myprogram
```

When using SLURM, these settings may be applied with the `srun --export` option. For example:

```
totalview -replay -args srun -n 8 \
--export=ALL,MPICH_SMP_SINGLE_COPY_OFF=1,LD_PRELOAD=/<path>/undodb_infiniband_preload_x64.so \
myprogram
```

Global Arrays Applications (Classic UI Only)

The following paragraphs, which are copied from the Global Arrays home site (<http://hpc.pnl.gov/globalarrays>), describe the global arrays environment:

“The Global Arrays (GA) toolkit provides a shared memory style programming environment in the context of distributed array data structures (called “global arrays”). From the user perspective, a global array can be used as if it was stored in shared memory. All details of the data distribution, addressing, and data access are encapsulated in the global array objects. Information about the actual data distribution and locality can be easily obtained and taken advantage of whenever data locality is important. The primary target architectures for which GA was developed are massively-parallel distributed-memory and scalable shared-memory systems.

“GA divides logically shared data structures into “local” and “remote” portions. It recognizes variable data transfer costs required to access the data depending on the proximity attributes. A local portion of the shared memory is assumed to be faster to access and the remainder (remote portion) is considered slower to access. These differences do not hinder the ease-of-use since the library provides uniform access mechanisms for all the shared data regardless where the referenced data is located. In addition, any processes can access a local portion of the shared data directly/in-place like any other data in process local memory. Access to other portions of the shared data must be done through the GA library calls.

“GA was designed to complement rather than substitute for the message-passing model, and it allows the user to combine shared-memory and message-passing styles of programming in the same program. GA inherits an execution environment from a message-passing library (w.r.t. processes, file descriptors etc.) that started the parallel program.”

The global arrays environment has a few unique attributes. Using TotalView, you can:

- Display a list of a program's global arrays.
- Dive from this list of global variables to see the contents of a global array in C or Fortran format.
- Cast the data so that TotalView interprets data as a global array handle. This means that TotalView displays the information as a global array. Specifically, casting to **\$GA** forces the Fortran interpretation; casting to **\$ga** forces the C interpretation; and casting to **\$Ga** uses the language in the current context.

In the Array Statistics view, the commands that operate on a local array, such as slicing and obtaining statistics, also operate on global arrays.

The command used to start TotalView depends on your operating system. For example, the following command starts TotalView on a program invoked using **prun** using three processes:

```
totalview prun -a -N 3 boltz.x
```

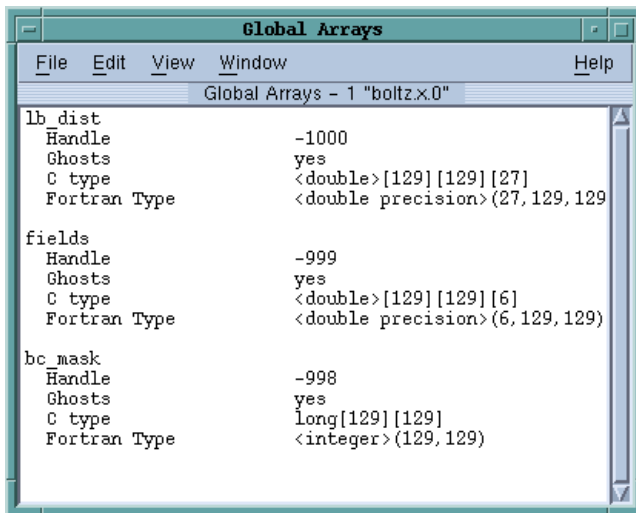
Before your program starts parallel execution, a Question dialog launches so you can stop the job to set breakpoints or inspect the program before it begins execution.

After your program hits a breakpoint, you can either use the CLI command **dga** to inspect your program's global arrays, or use the Classic UI's Global Arrays Window.

```
dga
```

Classic UI only:

Figure 109, Global Arrays Windows (Classic UI only)



The arrays named in this window are displayed using their C and Fortran type names. Diving on the line that contains the type definition displays Variable Windows that contain information about that array.

After TotalView displays this information, you can use other standard commands and operations on the array. For example, you can use the slice and filter operations and the commands that visualize, obtain statistics, and show the nodes from which the data was obtained.

You can cast the variable into a global array using either **\$ga** for a C Language cast or **\$GA** for a Fortran cast.

RELATED TOPICS

The **dga** command to view global arrays **dga** in the *TotalView Reference Guide*

The Global Array window in the Classic UI "Process Window: Tools Menu Commands: Tools > Global Arrays in the Classic UI Online Help

Shared Memory (SHMEM) Code

TotalView supports programs using the distributed memory access Shared Memory (SHMEM) library on Quadrics RMS systems and SGI Altix systems. The SHMEM library allows processes to read and write data stored in the memory of other processes. This library also provides collective operations.

Debugging a SHMEM RMS or SGI Altix program is no different than debugging any other program that uses a starter program. For example:

```
totalview srun -a my_program
```

UPC Programs

TotalView supports debugging UPC programs on Linux x86 platforms. This section discusses only the UPC-specific features of TotalView. It is not an introduction to the UPC Language. For an introduction to the UPC language, see [Introduction to Unified Parallel C: A PGAS C](#).

NOTE: When debugging UPC code, TotalView requires help from a UPC assistant library that your compiler vendor provides. You need to include the location of this library in your `LD_LIBRARY_PATH` environment variable. TotalView also provides assistants that you can use.

Topics in this section are:

- [Invoking TotalView](#)
- [Viewing Shared Objects \(Classic UI Only\)](#)
- [Displaying Pointer to Shared Variables \(Classic UI Only\)](#)

Invoking TotalView

The way in which you invoke TotalView on a UPC program is straight-forward. However, this procedure depends on the parallel technology you are using. Here are a couple of examples:

- For Quadrics RMS:

```
totalview prun -a prog_upc_args
```

- For MPICH and LAM

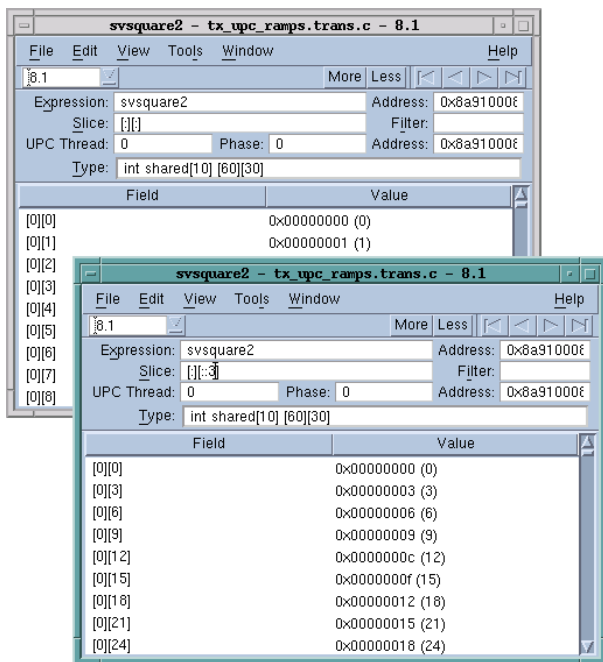
```
totalview mpirun -a -np 2 prog_upc_args
```

Viewing Shared Objects (Classic UI Only)

TotalView displays UPC shared objects, and fetches data from the UPC thread with which it has an affinity. For example, TotalView always fetches shared scalar variables from thread 0.

The upper-left screen in [Figure 110](#) displays elements of a large shared array. You can manipulate and examine shared arrays the same as any other array. For example, you can slice, filter, obtain statistical information, and so on. The lower-right screen shows a slice of this array.

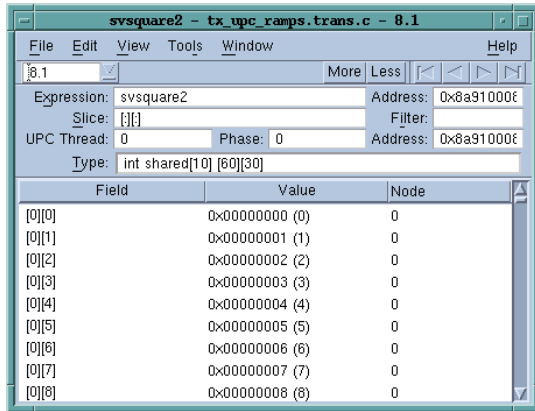
Figure 110, A Sliced UPC Array



In this figure, TotalView displays the value of a pointer-to-shared variable whose target is the array in the **Shared Address** area. As usual, the address in the process appears in the top left of the display.

Since the array is shared, it has an additional property: the element's affinity. You can display this information if you right-click your mouse on the header and tell TotalView to display Nodes.

Figure 111, UPC Variable Window Showing Nodes

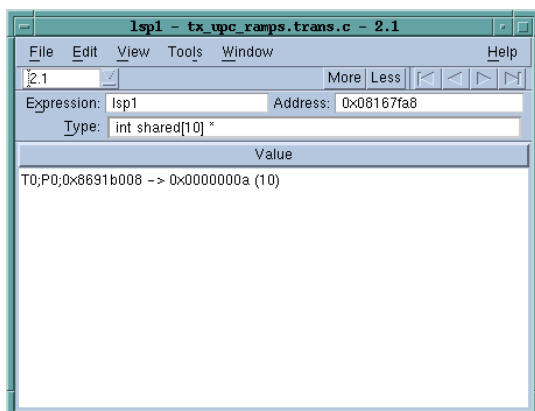


You can also use the **Tools > Visualize Distribution** command to visualize this array. For more information on visualization, see “Array Visualizer” in the chapter “Visualizing Programs and Data” in the *Classic TotalView User Guide*.

Displaying Pointer to Shared Variables (Classic UI Only)

TotalView understands pointer-to-shared data and displays the components of the data, as well as the target of the pointer to shared variables. For example, Figure 112 shows this data being displayed:

Figure 112, A Pointer to a Shared Variable



In this figure, note the following:

- Because the **Type** field displays the full type name, this is a pointer to a shared **int** with a block size of 10.
- TotalView also displays the **upc_threadof ("T0")**, the **upc_phaseof ("P0")**, and the **upc_addrfield (0x0x10010ec4)** components of this variable.

In the same way that TotalView normally shows the target of a pointer variable, it also shows the target of a UPC pointer variable. When dereferencing a UPC pointer, TotalView fetches the target of the pointer from the UPC thread with which the pointer has affinity.

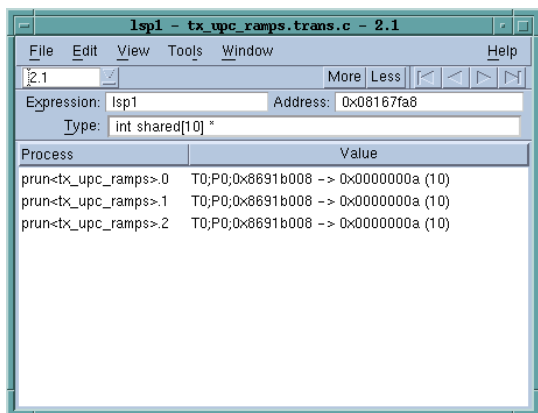
You can update the pointer by selecting the pointer value and editing the thread, phase, or address values. If the phase is corrupt, you'll see something like the following in the **Value** area:

```
T0;P6;0x3ffc0003b00 <Bad phase [max 4]> ->
                                0xc0003c80 (-1073726336)
```

In this example, the pointer is invalid because the phase is outside the legal range. TotalView displays a similar message if the thread is invalid.

Since the pointer itself is not shared, you can use the **TView > Show Across** commands to display the value from each of the UPC threads.

Figure 113, Pointer to a Shared Variable



CoArray Fortran (CAF) Programs

TotalView has partial support for debugging CoArray Fortran (CAF) programs on Cray platforms. This section discusses the parts of TotalView that support CAF-specific features. CoArray Fortran allows a programmer to distribute parts of an array over a set of processes using an augmented Fortran array syntax. The processes in a CAF job share the same executable. The processes are assigned "image ids" starting at image one.

When debugging CAF code, TotalView requires help from a CAF assistant library that your compiler vendor provides. You need to include the location of this library in your `LD_LIBRARY_PATH` environment variable. TotalView also provides assistants that you can use.

Because TotalView support is partial, expressions that attempt to re-cast CAF types or change the visible slices of CAF types are likely to fail.

Invoking TotalView

CAF programs commonly rely on an underlying parallel protocol such as MPI. They are started the same way as other programs using the same parallel technology.

On Cray machines that use `aprun`, invoking a four-image job on TotalView may look like this:

```
totalview aprun -a -n 4 caf_program caf_program_args
```

Viewing CAF Programs (Classic UI Only)

For a CAF program, the process id in the TotalView Process window shows the CAF image id. TotalView shows the correct dimensions and co-dimensions of arrays and the co-dimensions of scalars.

When diving on a CAF array or scalar, TotalView shows the data local to the current image. Diving across processes shows the entire distributed array.

Figure 114, Diving on CAF array “y”

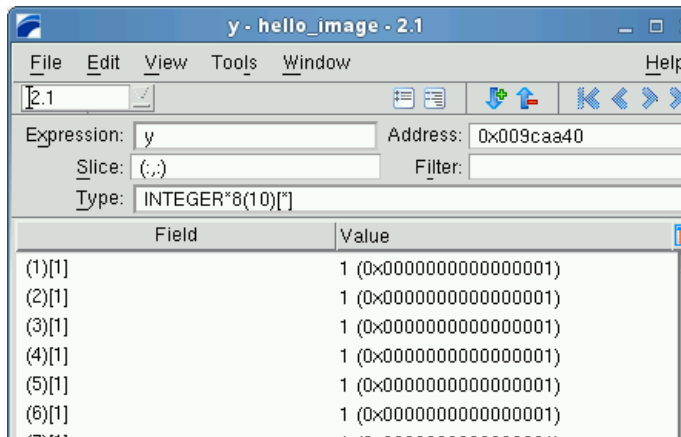
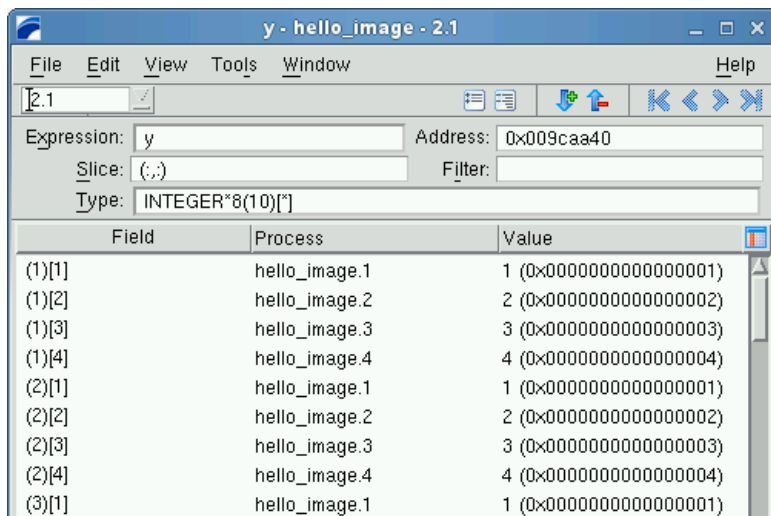


Figure 115, Diving on CAF array “y” across processes



If you use the **array viewer**, **statistics**, and **visualizer** commands from the Tools menu when viewing a CAF array across processes, the commands treat the co-array dimensions much like standard array dimensions.

Using CLI with CAF

The **dprint** command in the CLI displays the data in CAF arrays in a similar way to the above. When the focus is a process, **dprint** lists the local values. When the focus is the shared group containing the CAF images, **dprint** lists the entire co-array.

MPI Program Setup

NOTE: This section provides non-UI setup information for MPI programs. In most cases, you can just launch an MPI session from the UI without referring to the details here. See [Debug a Parallel Program](#) in the *Creating and Managing Sessions* chapter.

- MPICH Applications
- MPICH2 Applications
- Cray MPI Applications
- IBM MPI Parallel Environment (PE) Applications
- Open MPI Applications
- QSW RMS Applications

RELATED TOPICS

Setting up a parallel debugging session using the [Debug a Parallel Program](#) Session Editor

Scalability configuration in TotalView [Scalability in HPC Computing Environments](#)

Creating startup profiles for environments not defined by TotalView. These definitions will appear in the Additional Starter Arguments field of the Debug New Parallel Program dialog box. [MPI Startup Customizations](#)

MPICH Applications

To debug Message Passing Interface/Chameleon Standard (MPICH) applications, you must use MPICH version 1.2.3 or later on a homogeneous collection of computers. If you need a copy of MPICH, you can obtain it at no cost from Argonne National Laboratory at <http://www.mpich.org/downloads>. (We strongly urge that you use a later version of MPICH. For versions that work with TotalView, see [TotalView Supported Platforms](#) document in the TotalView distribution at `<installdir>/totalview.<version>/doc/pdf` or [TotalView Supported Platforms](#) on the TotalView documentation website.)

The MPICH library should use the **ch_p4**, **ch_p4mpd**, **ch_shmem**, **ch_lfshmem**, or **ch_mpl** devices.

- For networks of workstations, the default MPICH library is **ch_p4**.

- For shared-memory SMP computers, use **ch_shmem**.
- On an IBM SP computer, use the **ch_mpl** device.

The MPICH source distribution includes all these devices. Choose the one that best fits your environment when you configure and build MPICH.

When configuring MPICH, you must ensure that the MPICH library maintains all of the information that TotalView requires. This means that you must use the **-enable-debug** option with the MPICH **configure** command. (Versions earlier than 1.2 used the **--debug** option.)

RELATED TOPICS

More information on debugging MPICH applications

“MPICH Debugging Tips” in the *Classic TotalView User Guide’s* chapter “Debugging Strategies for Parallel Applications.”

Starting TotalView on an MPICH Job

NOTE: Before you can bring an MPICH job under TotalView’s control, both TotalView and the **tvdsrv** must be in your path, most easily set in a login or shell startup script.

For version 1.1.2, the following command-line syntax starts a job under TotalView control:

```
mpirun [ MPICH-arguments ] -tv program [ program-arguments ]
```

For example:

```
mpirun -np 4 -tv sendrecv
```

The MPICH **mpirun** command obtains information from the **TOTALVIEW** environment variable and then uses this information when it starts the first process in the parallel job.

For Version 1.2.4, the syntax changes to the following:

```
mpirun -dbg=totalview [ other_mpitch-args ] program [ program-args ]
```

For example:

```
mpirun -dbg=totalview -np 4 sendrecv
```

In this case, **mpirun** obtains the information it needs from the **-dbg** command-line option.

In other contexts, setting this environment variable means that you can use different versions of TotalView or pass command-line options to TotalView.

For example, the following is the C shell command that sets the **TOTALVIEW** environment variable so that **mpirun** passes the **-no_stop_all** option to TotalView:

```
setenv TOTALVIEW "totalview -no_stop_all"
```

TotalView begins by starting the first process of your job, the master process, under its control. You can then set breakpoints and begin debugging your code.

On the IBM SP computer with the **ch_mpl** device, the **mpirun** command uses the **poe** command to start an MPI job. While you still must use the MPICH **mpirun** (and its **-tv** option) command to start an MPICH job, the way you start MPICH differs. For details on using TotalView with **poe**, see [Starting TotalView on a PE Program](#).

Starting TotalView using the **ch_p4mpd** device is similar to starting TotalView using **poe** on an IBM computer or other methods you might use on Sun and HP platforms. In general, you start TotalView using the **totalview** command, with the following syntax;

```
totalview mpirun [ totalview_args ] -a [ mpich-args ] program [ program-args ]
```

```
totalviewcli mpirun [ totalview_args ] \ -a [ mpich-args ] program [
program-args ]
```

As your program executes, TotalView automatically acquires the processes that are part of your parallel job as your program creates them. Before TotalView begins to acquire them, it asks if you want to stop the spawned processes. If you click **Yes**, you can stop processes as they are initialized. This lets you check their states or set breakpoints that are unique to the process. TotalView automatically copies breakpoints from the master process to the slave processes as it acquires them. Consequently, you don't have to stop them just to set these breakpoints.

If you're using the UI, TotalView updates the view to show these newly acquired processes.

RELATED TOPICS

More information on attaching to processes

"Attaching to Processes Tips" in the *Classic TotalView User Guide's* chapter "Debugging Strategies for Parallel Applications."

Attaching to an MPICH Job

You can attach to an MPICH application even if it was not started under TotalView control. To attach to an MPICH application:

1. Start TotalView.

Select **Attach to process** on the Start a Debugging Session dialog. A list of processes running on the selected host displays in the **Attach to Running Program(s)** dialog.

1. Attach to the first MPICH process in your workstation cluster by diving into it.

```
dattach executable pid
```

2. On an IBM SP with the **ch_mpi** device, attach to the **poe** process that started your job. For details, see [Starting TotalView on a PE Program](#).

Normally, the first MPICH process is the highest process with the correct program name in the process list. Other instances of the same executable can be:

- The **p4** listener processes if MPICH was configured with **ch_p4**.
 - Additional slave processes if MPICH was configured with **ch_shmem** or **ch_lfshmem**.
 - Additional slave processes if MPICH was configured with **ch_p4** and has a file that places multiple processes on the same computer.
1. After attaching to your program's processes, a dialog launches where you can choose to also attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPICH processes.

Classic UI Only: As an alternative, use the **Group > Attach Subset** command to predefine what TotalView should do.

RELATED TOPICS

More information on attaching to processes "Attaching to Processes Tips" in the *Classic TotalView User Guide's* chapter "Debugging Strategies for Parallel Applications."

In some situations, the processes you expect to see might not exist (for example, they may crash or exit). TotalView acquires all the processes it can and then warns you if it cannot attach to some of them. If you attempt to dive into a process that no longer exists (for example, using a message queue display), you are alerted that the process no longer exists.

Using MPICH P4 procgroup Files

If you're using MPICH with a P4 **procgroup** file (by using the **-p4pg** option), you must use the *same* absolute path name in your **procgroup** file and on the **mpirun** command line. For example, if your **procgroup** file contains a different path name than that used in the **mpirun** command, even though this name resolves to the same executable, TotalView assumes that it is a different executable, which causes debugging problems.

The following example uses the same absolute path name on the TotalView command line and in the **procgroup** file:

```
% cat p4group
local 1 /users/smith/mympichexe
bigiron 2 /users/smith/mympichexe
% mpirun -p4pg p4group -tv /users/smith/mympichexe
```

In this example, TotalView does the following:

1. Reads the symbols from **mympichexe** only once.
2. Places MPICH processes in the same TotalView share group.
3. Names the processes **mympichexe.0**, **mympichexe.1**, **mympichexe.2**, and **mympichexe.3**.

If TotalView assigns names such as **mympichexe<mympichexe>.0**, a problem occurred and you need to compare the contents of your **procgroup** file and **mpirun** command line.

MPICH2 Applications

NOTE: You should be using MPICH2 version 1.0.5p4 or higher. Earlier versions had problems that prevented TotalView from attaching to all the processes or viewing message queue data.

Downloading and Configuring MPICH2

You can download the current MPICH2 version from:

<http://www.mpich.org/downloads/versions/>

If you wish to use all of the TotalView MPI features, you must configure MPICH2. Do this by adding one of the following to the **configure** script that is within the downloaded information:

--enable-debuginfo

or

--enable-totalview

The **configure** script looks for the following file:

python2.x/config/Makefile

It fails if the file is not there.

The next steps are:

1. Run **make**
2. Run **make install**

This places the binaries and libraries in the directory specified by the optional **--prefix** option.

3. Set the `PATH` and `LD_LIBRARY_PATH` to point to the MPICH2 **bin** and **lib** directories.

Starting TotalView Debugging on an MPICH2 Hydra Job

As of MPICH2 1.4.1, the default job type for MPICH2 is Hydra. If you are instead using MPD, see [Starting TotalView Debugging on an MPICH2 MPD Job](#).

Start a Hydra job as follows:

totalview -args mpiexec mpiexec-args program program-args

You may not see sources to your program at first. If you do see the program, you can set breakpoints. In either case, press the **Go** button to start your process. TotalView displays a dialog box when your program goes parallel that allows you to stop execution so you can set breakpoints.

(This is the default behavior. You can change it using the options within **File > Preferences > Parallel** page. See [Parallel Configuration](#).)

Starting TotalView Debugging on an MPICH2 MPD Job

You must start the **mpd** daemon before starting an MPICH2 MPI job.

NOTE: As of MPICH2 1.4.1, the default job type is Hydra, rather than MPD, so if you are using the default, there is no need to start the daemon. See [Starting TotalView Debugging on an MPICH2 Hydra Job](#).

Starting the MPI MPD Job with MPD Process Manager

To start the `mpd` daemon, use the **mpdboot** command. For example:

```
mpdboot -n 4 -f hostfile
```

where:

-n 4

The number of hosts on which you wish to run the daemon. In this example, the daemon runs on four hosts

-f hostfile

Lists the hosts on which the application will run. In this example, a file named **hostfile** contains this list.

You are now ready to start debugging your application.

Starting an MPICH2 MPD Job

Start an MPICH2 MPD job in one of the following ways:

`mpiexec mpi-args -tv program -a program-args`

This command tells MPI to start TotalView. You must have set the TOTALVIEW environment variable with the path to TotalView's executable when you start a program using **mpiexec**. For example:

```
setenv TOTALVIEW \  
    /opt/totalview/bin/totalview
```

This method of starting TotalView does not let you restart your program without exiting TotalView and you will not be able to attach to a running MPI job.

`totalview python -a `which mpiexec` -tvsu mpiexec-args program program-args`

This command lets you restart your MPICH2 job. It also lets you attach to a running MPICH2 job by using the **Attach to a Running Program** dialog box. You need to be careful that you attach to the right instance of python as it is likely that a few instances are running. The one to which you want to attach has no attached children—child processes are indented with a line showing the connection to the parent.

You may not see sources to your program at first. If you do see the program, you can set breakpoints. In either case, press the **Go** button to start your process. TotalView displays a dialog box when your program goes parallel that allows you to stop execution. (This is the default behavior. You can change it using the options within **File > Preferences > Parallel** page.)

You will also need to set the TOTALVIEW environment variable as indicated in the previous method.

Cray MPI Applications

Specific information on debugging Cray MPI applications is discussed in [Cray XT/XE/XK/XC Applications](#) for information.

IBM MPI Parallel Environment (PE) Applications

You can debug IBM MPI Parallel Environment (PE) applications on the IBM RS/6000 and SP platforms.

To take advantage of TotalView's ability to automatically acquire processes, you must be using release 3,1 or later of the Parallel Environment for AIX.

Topics in this section are:

- [Preparing to Debug a PE -Application](#)
- [Starting TotalView on a PE Program](#)
- [Setting Breakpoints](#)
- [Starting Parallel Tasks](#)
- [Attaching to a PE Job](#)

Preparing to Debug a PE -Application

The following sections describe what you must do before TotalView can debug a PE application.

Using Switch-Based Communications

If you're using switch-based communications (either *IP over the switch* or *user space*) on an SP computer, configure your PE debugging session so that TotalView can use *IP over the switch* for communicating with the TotalView Server (**tvdsvr**). Do this by setting the **-adapter_use** option to **shared** and the **-cpu_use** option to **multiple**, as follows:

- If you're using a PE host file, add **shared multiple** after all host names or pool IDs in the host file.
- Always use the following arguments on the **poe** command line:

```
-adapter_use shared -cpu_use multiple
```

If you don't want to set these arguments on the **poe** command line, set the following environment variables before starting **poe**:

```
setenv MP_ADAPTER_USE shared  
setenv MP_CPU_USE multiple
```

When using *IP over the switch*, the default is usually **shared adapter use** and **multiple cpu use**; we recommend that you set them explicitly using one of these techniques. You must run TotalView on an SP or SP2 node. Since TotalView will be using *IP over the switch* in this case, you cannot run TotalView on an RS/6000 workstation.

Performing a Remote Login

You must be able to perform a remote login using the **ssh** command. You also need to enable remote logins by adding the host name of the remote node to the **/etc/hosts.equiv** file or to your **.rhosts** file.

When the program is using switch-based communications, TotalView tries to start the TotalView Server by using the **ssh** command with the switch host name of the node.

Setting Timeouts

If you receive communications timeouts, you can set the value of the **MP_TIMEOUT** environment variable; for example:

```
setenv MP_TIMEOUT 1200
```

If this variable isn't set, TotalView uses a **timeout** value of 600 seconds.

Starting TotalView on a PE Program

The following is the syntax for running Parallel Environment (PE) programs from the command line:

```
program [ arguments ] [ pe_arguments ]
```

You can also use the **poe** command to run programs as follows:

```
poe program [arguments] [pe_arguments]
```

If, however, you start TotalView on a PE application, you must start **poe** as TotalView's target using the following syntax:

```
{ totalview | totalviewcli } poe -a program [arguments] [PE_arguments]
```

For example:

```
totalview poe -a sendrecv 500 -rmpool 1
```

Setting Breakpoints

After TotalView is running, start the **poe** process using the **Process > Go** command.

```
dfocus p dgo
```

A dialog box launches in the UI (the CLI prints a query) to determine if you want to stop the parallel tasks.

If you want to set breakpoints in your code before they begin executing, answer **Yes**. TotalView initially stops the parallel tasks, which also allows you to set breakpoints. You can now set breakpoints and control parallel tasks in the same way as any process controlled by TotalView.

If you have already set and saved breakpoints with the **Action Points > Save** command and you want to reload the file, answer **No**. After TotalView loads these saved breakpoints, the parallel tasks begin executing.

```
dactions-save filename dactions-load filename
```

Starting Parallel Tasks

After you set breakpoints, you can start all of the parallel tasks with the **Group > Go** command.

```
dfocus G dgo Abbreviation: G
```

NOTE: No parallel tasks reach the first line of code in your main routine until all parallel tasks start.

Be cautious in placing breakpoints at or before a line that calls **MPI_Init()** or **MPL_Init()** because timeouts can occur while your program is being initialized. After you allow the parallel processes to proceed into the **MPI_Init()** or **MPL_Init()** call, allow all of the parallel processes to proceed through it within a short time.

RELATED TOPICS

Saving action points

[Saving and Loading Action Points](#)

More information on debugging with IBM PE

“IBM PE Debugging Tips” in the *Classic TotalView User Guide's* chapter “Debugging Strategies for Parallel Applications.”

Attaching to a PE Job

To take full advantage of TotalView's **poe**-specific automation, you need to attach to **poe** itself, and let TotalView automatically acquire the **poe** processes on all of its nodes. In this way, TotalView acquires the processes you want to debug.

Attaching from a Node Running poe

To attach TotalView to **poe** from the node running **poe**:

1. Start TotalView in the directory of the debug target.

If you can't start TotalView in the debug target directory, you can start TotalView by editing the **tvdsvr** command line before attaching to **poe**.

For details, see “Setting the Single-Process Server Launch Command” in the *Classic TotalView User Guide's* chapter “Setting Up Remote Debugging Sessions.”

2. In the **File > Attach to a Program**, then

find the **poe** process list, and attach to it by diving into it. When necessary, TotalView launches **tvdsvrs**. TotalView also updates the Root Window and opens a Process Window for the **poe** process.

```
dattachpoe pid
```

3. Locate the process you want to debug by selecting it in the Processes and Threads view which should display it in the Source pane. If your source code files are not displayed, use **File > Preferences > Search Path** command to add directories to your search path.

Attaching from a Node Not Running poe

The procedure for attaching TotalView to **poe** from a node that is not running **poe** is essentially the same as the procedure for attaching from a node that is running **poe**. Since you did not run TotalView from the node running **poe** (the startup node), you won't be able to see **poe** in the Process and Threads view, and you won't be able to start it by diving into it.

To place **poe** in this list:

1. Connect TotalView to the startup node.
For details, see “Setting the Single-Process Server Launch Command” in the *Classic TotalView User Guide’s* chapter “Setting Up Remote Debugging Sessions.”
2. Select the **File > Attach to a Program**.
3. Look for the process named **poe** and continue as if attaching from a node that is running **poe**.

```
dattach-r hostname poe poe-pid
```

Open MPI Applications

Open MPI is an open source implementation of both the MPI-1 and MPI-2 documents that combines some aspects of four different (and now no longer under active development) MPI implementations: FT-MPI from the University of Tennessee, LA-MPI from Los Alamos National Laboratory, LAM/MPI from Indiana University, and PACX-MPI from the University of Stuttgart.

For more information on Open MPI, see <https://www.open-mpi.org/>.

Debug an Open MPI program similarly to most MPI programs, using the following syntax if TotalView is in your path:

```
mpirun -tv args prog prog_args
```

As an alternative, you can invoke TotalView on **mpirun**.

```
totalview -args mpirun args ./prog
```

For example, to start TotalView on a four-process MPI program:

```
totalview -args mpirun -np 4 ./mpi_program
```

Alternatively, use the Session Manager’s **Parallel Session** dialog (accessed via **Process > Modify Arguments**) to enter the parallel session details in the UI.

QSW RMS Applications

Starting TotalView on an RMS Job

To start a parallel job under TotalView control, use TotalView as if you were debugging **prun**:

```
{ totalview | totalviewcli } prun -a prun-command-line
```

TotalView starts and shows you the machine code for RMS **prun**. Since you're not usually interested in debugging this code, use the **Process > Go** command to let the program run.

```
dfocus
pdgo
```

The RMS **prun** command executes and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at startup. If you answer **yes**, TotalView halts them before they enter the main program. You can then create breakpoints.

Attaching to an RMS Job

To attach to a running RMS job, attach to the RMS **prun** process that started the job.

You attach to the **prun** process the same way you attach to other processes.

After you attach to the RMS **prun** process, you have the option to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPI processes.

Classic UI Only: As an alternative, use the **Group > Attach Subset** command to predefine what TotalView should do.

RELATED TOPICS

Attaching to processes using **prun**

[Attach to Process](#)

Using the **Group > Attach Subset** command to specify TotalView behavior when attaching to an RMS **prun** process

"**Group > Attach Subset**" in the *Classic TotalView User Guide* (In-Product_Help: TotalView Online Help: Process Window: Group Menu Commands: Group > Attach Subset)

Tips when attaching to processes in a multi-process job

"Attaching to Processes Tips" in the "Debugging Strategies for Parallel Applications" chapter of the *Classic TotalView User's Guide*

SGI MPI Applications

TotalView can acquire processes started by SGI MPI applications. This MPI is part of the Message Passing Toolkit (MPT) 1.3 and 1.4 packages.

Classic UI Only: TotalView can display the Message Queue Graph Window in the Classic UI for these releases. See “Displaying the Message Queue Graph Window” in the chapter “Debugging Strategies for Parallel Applications” in the *Classic TotalView User Guide*.

Starting TotalView on an SGI MPI Job

You normally start SGI MPI programs by using the **mpirun** command. You use a similar command to start an MPI program under debugger control, as follows:

```
{ totalview | totalviewcli } mpirun -a mpirun-command-line
```

This invokes TotalView and tells it to show you the machine code for **mpirun**. Since you’re not usually interested in debugging this code, use the **Process > Go** command to let the program run.

```
dfocus p dgo
```

The SGI MPI **mpirun** command runs and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at startup. If you answer **Yes**, TotalView halts them before they enter the main program. You can then create breakpoints.

If you set a verbosity level that allows informational messages, TotalView also prints a message that shows the name of the array and the value of the array services handle (**ash**) to which it is attaching.

Attaching to an SGI MPI Job

To attach to a running SGI MPI program, attach to the SGI MPI **mpirun** process that started the program. The procedure for attaching to an **mpirun** process is the same as that for attaching to any other process.

After you attach to the **mpirun** process, TotalView asks if you also want to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPICH processes.

Classic UI Only: As an alternative, use the **Group > Attach Subset** command to predefine what TotalView should do.

Attaching to an mpirun process	"Debugging an MPI Program" in the chapter "Starting TotalView" in the <i>Classic TotalView User Guide</i> .
Using the Group > Attach Subset command to specify TotalView behavior when attaching to an RMS prun process	" Group > Attach Subset " in the <i>Classic TotalView User Guide</i> (In-Product_Help: TotalView Online Help: Process Window: Group Menu Commands: Group > Attach Subset)
Using the Group > Attach Subset command to specify TotalView behavior when attaching to a process	"Attaching to Processes Tips" in the <i>Classic TotalView User Guide's</i> chapter "Debugging Strategies for Parallel Applications."

Using ReplayEngine with SGI MPI

SGI MPI uses the **xpmem** module to map memory from one MPI process to another during job startup. Memory mapping is enabled by default. The size of this mapped memory can be quite large, and can have a negative effect on TotalView's ReplayEngine performance. Therefore, mapped memory is limited by default for the **xpmem** module if Replay is enabled. The environment variable, **MPI_MEMMAP_OFF**, is set to 1 in the TotalView file **parallel_support.tvd** by adding the variable to the `replay_env:` specification as follows: **replay_env:**

```
MPI_MEMMAP_OFF=1
```

If full memory mapping is required, set the startup environment variable in the Arguments field of the Program Session dialog. Add the following to the environment variables: **MPI_MEMMAP_OFF=0**.

Be aware that the default mapped memory size may prove to be too large for ReplayEngine to deal with, and it could be quite slow. You can limit the size of the mapped heap area by using the **MPI_MAPPED_HEAP_SIZE** environment variable documented in the SGI documentation. After turning off **MEMMAP_OFF** as described above, you can set the size (in bytes) in the TotalView startup parameters.

For example:

```
MPI_MAPPED_HEAP_SIZE=1048576
```

SGI has a patch for an *MPT/XPMEM* issue. Without this patch, *XPMEM* can crash the system if ReplayEngine is turned on. To get the *XPMEM* fix for the munmap problem, either upgrade to ProPack 6 SP 4 or install SGI patch 10570 on top of ProPack 6 SP 3.

Sun MPI Applications

TotalView can debug a Sun MPI program and can display Sun MPI message queues. This section describes how to perform *job startup* and *job attach* operations.

To start a Sun MPI application:

1. Enter the following command:

```
totalview mprun [ totalview_args ] -a [ mpi_args ]
```

For example:

```
totalview mprun -g blue -a -np 4 /usr/bin/mpi/conn.x  
totalviewcli mprun [ totalview_args ] -a [ mpi_args ]
```

When the TotalView Process Window appears, select the **Go** button.-

```
dfocus p dgo
```

TotalView may display a dialog box with the following text:

```
Process mprun is a parallel job. Do you want to stop  
the job now?
```

2. If you compiled using the **-g** option, click **Yes** to display your source in the Source pane. All processes are halted.

Attaching to a Sun MPI Job

To attach to an already running **mprun** job:

1. Find the host name and process identifier (PID) of the **mprun** job by typing **mpps -b**. For more information, see the **mpps(1M)** manual page.

The following is sample output from this command:

```
JOBNAME      MPRUN_PID    MPRUN_HOST  
cre.99       12345        hpc-u2-9  
cre.100      12601        hpc-u2-8
```

2. After selecting **File > Attach to a Running Program**, type **mprun** in the **File Name** field and type the PID in the **Process ID** field.

```
dattach mprun mprun-pidFor example: dattach mprun 12601
```

3. If TotalView is running on a different node than the **mprun** job, select the host or add a new host in the **Host** field.

```
dattach -r host-name mprun mprun-pid
```

Troubleshooting MPI Startup

If you can't successfully start TotalView on MPI programs, check the following:

- Can you successfully start MPICH programs without TotalView?

The MPICH code contains some useful scripts that verify if you can start remote processes on all of the computers in your computers file. (See **tstmachines** in **mpich/util**.)

- You won't get a message queue display if you get the following warning:

The symbols and types in the MPICH library used by TotalView to extract the message queues are not as expected in the image <your image name>. This is probably an MPICH version or configuration problem.

Check that you are using MPICH Version 1.1.0 or later and that you have configured it with the **-debug** option. (You can check this by looking in the **config.status** file at the root of the MPICH directory tree.)

- Does the TotalView Server (**tvdsvr**) fail to start?
tvdsvr must be in your **PATH** when you log in. Remember that TotalView uses **ssh** to start the server, and that this command doesn't pass your current environment to remotely started processes.
- Make sure you have the correct MPI version and have applied all required patches. See the TotalView Release Notes at <https://help.totalview.io/> for up-to-date information.
- Under some circumstances, MPICH kills TotalView with the **SIGINT** signal. You can see this behavior when you use the **Group > Kill** command as the first step in restarting an MPICH job.

dfocus g ddelete

If TotalView exits and terminates abnormally with a **Killed** message, try setting the **TV::ignore_control_c** variable to true.

The Group > Kill command	Individual Execution Commands and dkill in the <i>TotalView Reference Guide</i> .
Tips for debugging MPI applications	"MPI Debugging Tips and Tools" in the chapter "Debugging Strategies for Parallel Applications" in the <i>Classic TotalView User Guide</i> .
The TotalView server, tvdsvr	" The tvdsvr Command and its Options " in the <i>TotalView Reference Guide</i>
MPI version information	The <i>TotalView Release Notes</i> on the TotalView documentation page

Using ReplayEngine with Infiniband MPIS

In general, using ReplayEngine with MPI versions that communicate over Infiniband is no different than using it with other MPIS, but its use requires certain environment settings, as described here. If you are launching the MPI job from within TotalView, these are set for you; if instead, you start the MPI program from outside TotalView, you must explicitly set your environment.

Required Environment Settings

When you start the MPI program from within TotalView with ReplayEngine enabled, TotalView inserts environment variable settings into the MPI processes to disable certain RDMA optimizations. (These are optimizations that hinder ReplayEngine's ability to identify the memory regions being actively used for RDMA, and their use can therefore result in unreasonably slow execution in record mode.) These variables are set for you, requiring no extra tasks compared to using a non-Infiniband MPI.

The inserted settings are:

- `VIADEV_USE_DREG_CACHE=0` (addresses MVAPICH1 versions)
- `MV2_DREG_CACHE_LIMIT=1` (addresses MVAPICH2 versions)
- `MV2_RNDV_PROTOCOL=R3` (addresses Intel MPI versions, also affects MVAPICH2)
- `OMPI_MCA_mpool_rdma_rcache_size_limit=1` (addresses Open MPI versions)

When the MPI program is started outside TotalView (for example, when using a command like `mpirun -tv`, or when you attach TotalView to an MPI program that is already running), you must set the relevant environment variable for your MPI version, as described above. Also, two additional environment variables are required to make the MPI program's use of RDMA memory visible to ReplayEngine, as follows:

- `IBV_FORK_SAFE`: Set to any value, for example `IBV_FORK_SAFE=1`
- `LD_PRELOAD`: Set to include a preload library, which can be found under the TotalView installation directory at `toolworks/totalview.<version>/linux-x86-64/lib/undodb_infiniband_preload_x64.so`.

For example, here's how to set the environment for the MVAPICH1 implementation of MPI:

```
mpirun_rsh -np 8 -hostfile myhosts \  
VIADEV_USE_DREG_CACHE=0 IBV_FORK_SAFE=1 \  
LD_PRELOAD=/<path>/undodb_infiniband_preload_x64.so myprogram
```

For more information, consult your MPI version documentation for specifics on setting environment variables.

Cray XT/XE/XK/XC MPIs

On Cray XT/XE/XK/XC (x86_64 only) systems, although Infiniband is not used, the MPIs do use RDMA techniques. As a result, using Replay on these systems requires some particular environmental settings. Briefly, the required settings are `MPICH_SMP_SINGLE_COPY_OFF = 1`, and `LD_PRELOAD` set to the location of the Infiniband preload library described above. Refer to [Cray XT/XE/XK/XC Applications](#) for details.

Possible Errors

ReplayEngine checks environment settings before it attaches to the MPI program, but in some cases, may not detect incompatible settings, reporting the following errors:

- If ReplayEngine finds that either the **IBV_FORK_SAFE** setting is absent, or that the preload library has not been loaded, it declines to attach and issues an error message citing unmet prerequisites. You can still attach TotalView to the program without ReplayEngine - for example, in the GUI by using the New Program dialog.
- If ReplayEngine cannot determine that the environment variable setting to disable an MPI optimization has been set, it continues to attach, but issues a warning message that it could not verify prerequisites. Depending on your program's use of memory for RDMA, you may find that it runs unreasonably slowly in record mode, or encounters errors that would not occur if ReplayEngine were not attached.

Using ReplayEngine in general

Using Replay Engine

MPI Startup Customizations

Here you will find information that will allow you to create startup profiles for environments that TotalView 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 a Parallel Program** dialog.

In general, TotalView supports various Message Passing Interface (MPI) implementations with no special configuration on your part. However, subtle differences in your environment or an implementation can cause difficulties that prevent TotalView from automatically starting your program. In these cases, you'll need to define how TotalView behaves.

Customizing Your Parallel Configuration

Select a parallel configuration in the **File > Debug a Parallel Program** dialog box. If the provided default configurations 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. Use the variable **TV::parallel_configs** to customize parallel configurations.

TotalView Customizations

Set the **TV::parallel_configs** variable, either local to your TotalView installation or globally:

- Globally, in your system's **.tvdrc** file. If you set this variable here, everyone using this TotalView version will see the definition.
- Locally, in your **.totalview/tvdrc** 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.

If you are using a locally-installed MPI implementation, add it to your PATH variable. By default, TotalView uses the information in PATH to find the parallel launcher (for example, **mpirun**, **mpiexec**, **poe**, **srun**, **prun**, **dmpirun**, and so on). Generally, if you can run your parallel job from a command line, TotalView 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 **tvdrc** file located in the **.totalview** subdirectory in your home directory.

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.

Example Parallel Configuration Definitions

This section provides three examples of customized parallel configurations. See [Customizing Your Parallel Configuration](#) 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 a Parallel Program dialog's Parallel System list.

Here are three examples:

```
dset TV::parallel_configs {
    #Argonne MPICH
    name:          MPICH;
    description:   Argonne MPICH;
    starter:       mpirun -tv -ksq %s %p %a;
    style:         setup_script;
    tasks_option:  -np;
    nodes_option:  -nodes;
    env_style:     force;
    pretest:       mpichversion;

    #Argonne MPICH2
    name:          MPICH2;
    description:   Argonne MPICH2;
```

```

starter:      $mpiexec -tvsu %s %p %a;
style:        manager_process;
tasks_option: -n;
env_option:   -env;
env_style:    assign_space_repeat;
comm_world:   0x44000000;
pretest:     mpich2version

# AIX POE
name:         poe - AIX;
description:  IBM PE - AIX;
tasks_option: -procs;
tasks_env:    MP_PROCS;
nodes_option: -nodes;
starter:     /bin/poe %p %a %s;
style:       bootstrap;
env:         NLSPATH=/usr/lib/nls/msg/%L/%N/: \
            /usr/lib/nls/msg/%L/%N.cat;
service_tids: 2 3 4;
comm_world:   0;
pretest:     test -x /bin/poe
msg_lib:     /usr/lpp/ppe.poe/lib/%m
}

```

All lines (except for comments) end with a semi-colon (;). Add spaces freely to improve the readability of these definitions as TotalView ignores them.

Notice that the MPICH2 definition contains the **\$mpiexec** variable. This variable is defined elsewhere in the **parallel_support.tvd** file as follows:

```
set mpiexec mpiexec;
```

There is no limit to how many definitions you can place within the **parallel_support.tvd** file or within a variable. The definitions you create will appear in the **Parallel** system pulldown list in the **File > Debug a Parallel Program** dialog box and can be used as an argument to the **--mpi** option of the CLI's **dload** command.

The fields that you can set are as follows:

comm_world

Use this option only 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 cannot acquire the rank for each MPI process.

description

(optional) A string describing what the configuration is used for. There is no length limit.

env

(optional) Defines environment variables that are placed in the starter program's environment. (Depending on how the starter works, these variables may not make their way into the actual ranked processes.) If you are defining more than one environment variable, define each in its own **env** clause.

The format to use is:

```
variable_name=value
```

env_option

(optional) Names the command-line option that exports environment variables to the tasks started by the launcher program. Use this option along with the **env_style** field.

env_style

(optional) Contains a list of environment variables that are passed to tasks.

assign: The argument to be inserted to the command-line option named in **env_option** is a comma-separated list of environment variable **name=value** pairs; that is,

```
NAME1=VALUE1 , NAME2=VALUE2 , NAME3=VALUE3
```

This option is ignored if you do not use an **env_option** clause.

assign_space_repeat: The argument after **env_option** is a space-separated name/value pair that is assigned to an environment variable. The command within **env_option** is repeated for each environment variable; that is, suppose you enter:

```
-env NAME1 VALUE1 -env NAME2 VALUE2  
-env NAME3 VALUE3
```

This mode is primarily used for the **mpiexec.py** MPICH2 starter program.

excenv

One of the following three strings:

export: The argument to be inserted after the command named in **env_option**. This is a comma-separated list of environment variable names; that is,

```
NAME1 , NAME2 , NAME3
```

This option is ignored if you do not use the **env_option** clause.

force: Environment variables are forced into the ranked processes using a shell script. TotalView or Memory-Scape 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 uses 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 > Debug a Parallel Program** dialog box and in the **Process > Modify Arguments** dialog box. TotalView remembers which configuration you use when starting a program so that it 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 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 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 marks as service threads.

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 uses 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 substitution 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, TotalView omits the parameter.

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 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.

Debugging OpenMP Applications

- OpenMP and the OMPD API
- Running Your Program
- Hybrid Programming: Combining OpenMP with MPI

OpenMP and the OMPD API

The OpenMP (Open Multi-Processing) standard provides a parallel programming API for defining multi-threaded, shared-memory programs. OpenMP consists of a series of compiler directives, library routines, and environment variables that configure runtime behavior.

For more detail on OpenMP itself, see the [OpenMP specification](#).

The OpenMP Debugging API (OMP_D) is an innovative new interface that allows third-party tools, such as debuggers, to extract the execution state of an OpenMP (OMP) runtime library, including live processes and core files. OMP_D was first added to the [OpenMP 5.0 specification](#), dated November 2018. See the [TotalView Platforms Guide](#) for current OMP_D platform support.

OMP_D allows the TotalView debugger, or other similar third-party tools, to extract information about OpenMP objects such as threads, parallel and task regions, control variables, internal control variables, parent/child thread relationships, and runtime call-stack boundaries.

TotalView provides an OpenMP view in the UI that displays OpenMP control variables, threads, and parallel and task regions. The Call Stack view can be filtered to hide internal OpenMP runtime frames and make it easier to peruse the stack and relevant **#pragma** directive frames. It also displays information on implicit and explicit task regions.

OMP_D Requirements

For the latest OMP_D compiler support, see the [Platforms Guide](#).

OpenMP Setup and Configuration

To take advantage of OMPD in TotalView, compile and link your program for OMPD support, then enable OpenMP debugging in TotalView and, optionally, enable stack filtering.

NOTE: Some compilers and OMPD libraries have bugs or other problems. See [Workarounds for Existing DWARF Debugging Problems](#) and [Compiling and Linking for OMPD Support](#) for detail.

Workarounds for Existing DWARF Debugging Problems

Set TotalView's `-compiler_vars` option

Many compilers have problems when generating DWARF debug information for OpenMP programs, especially for variables inside OMP constructs. The debug information is either completely missing, inaccurate, or is generated with the "artificial" attribute, which is intended to alert the debugger that the variable does not exist in the source code. The compilers also generate debug information for compiler-generated variables; these are also marked artificial and often outnumber the source code variables.

By default, TotalView does not automatically display artificial variables, so the source-code variables are not displayed. As a workaround, set the TotalView `-compiler_vars` option to display artificial variables. Unfortunately, this also causes the debugger to display the compiler-generated variables too, which can make it more difficult to see the source code variables.

To persist the `-compiler_vars` option across TotalView sessions, set the `TV::compiler_vars` state variable to true in your `tvdr` file located in the `.totalview` subdirectory in your home directory:

```
dset TV::compiler_vars -compiler_vars
```

For LLVM compilers, set the `LD_LIBRARY_PATH` environment variable

The `libompd.so` libraries in LLVM-Clang 17.0.6 and earlier, and in LLVM-Flang 17.0.1 and earlier, have an unnecessary dependency on `libomp.so`, causing TotalView to fail to load the library. Subsequent LLVM versions should address this problem.

The workaround is to set `LD_LIBRARY_PATH` to the directory path containing `libomp.so` in the LLVM installation before starting TotalView.

Compiling and Linking for OMPD Support

Compiling and linking a program for OMPD support varies by compiler. The following is a summary of compiling and linking rules:

- **Compiling with the `-fopenmp` option.**

Most compilers require using the `-fopenmp` option to enable compiling for OpenMP. Check the compiler documentation for details.

- **Linking with a special "lib-debug" version of the OpenMP and OMPD libraries.**

The AMD Clang and AMD Flang compilers require linking with a special `lib-debug` version of the OpenMP and OMPD libraries. How to link the `lib-debug` version differs, depending on the compiler and its version:

AMD Clang:

- *AMD Clang compiler drivers from ROCm 5.6 and newer:* Use the `-fopenmp-runtime-lib=lib-debug` compiler option to force the use of the library versions that support OMPD.
- *AMD Clang compiler drivers from ROCm 5.5:* The `-fopenmp-runtime-lib=lib-debug` is not supported; instead, use `RPATH` or `RUNPATH` to search for the directory that contains the debug libraries with the OMPD support, then add that to the link command. For example, for ROCm 6.0.0, the directory is `/opt/rocm-6.0.0/llvm/lib-debug`.

AMD Flang:

Use `RPATH` or `RUNPATH` as described for AMD Clang.

(While The newer AMD Flang compilers do support the `-fopenmp-runtime-lib=lib-debug` option, the compiler driver mishandles it.) Note that the `lib-debug` version of the OpenMP and OMPD libraries contain DWARF debug information. It is not known if these versions of the library were also built optimized.

- **HPE CCE**

No special compiler options are required.

The HPE CCE `cc` and `ftn` compilers do not require any special `RPATH` or compiler options for OMPD support. However, in versions up to and including CCE 18.0.0, bugs in the OMPD library result in incorrect OMPD results.

Enabling OpenMP Debugging

The OpenMP specification requires that the `OMP_DEBUG` environment variable be set to "enabled" before launching an OpenMP application to enable OMPD support in the OpenMP runtime library.

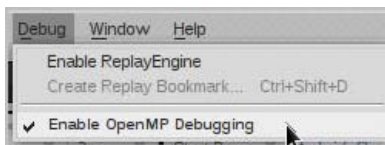
NOTE: **For MPI Applications:** MPI applications require that the environment variable `OMP_DEBUG` to be set *before* launching the application for debugging in TotalView. Setting an environment variable in an MPI application varies by MPI implementation and resource manager (for example, SLURM, Flux, Open MPI, or MPICH). For SLURM, for instance, the `srun` command supports an `--export` option, so enable OMPD support like this:

```
srun --export=OMP_DEBUG=enabled a.out
```

Enable OpenMP debugging for non-MPI programs:

For non-MPI programs launched by TotalView, you can enable OpenMP debugging and stack filtering using either the shell or the TotalView UI, CLI, or `-env` option.

In the UI, check the menu option **Debug > Enable OpenMP Debugging**.



In the shell before launching TotalView:

Set the environment variable `OMP_DEBUG`, for example, in `csh`:

```
setenv OMP_DEBUG enabled
```

When launching TotalView, using the `-env` option:

```
totalview -env OMP_DEBUG=enabled <your_program>
```

In the CLI from within TotalView:

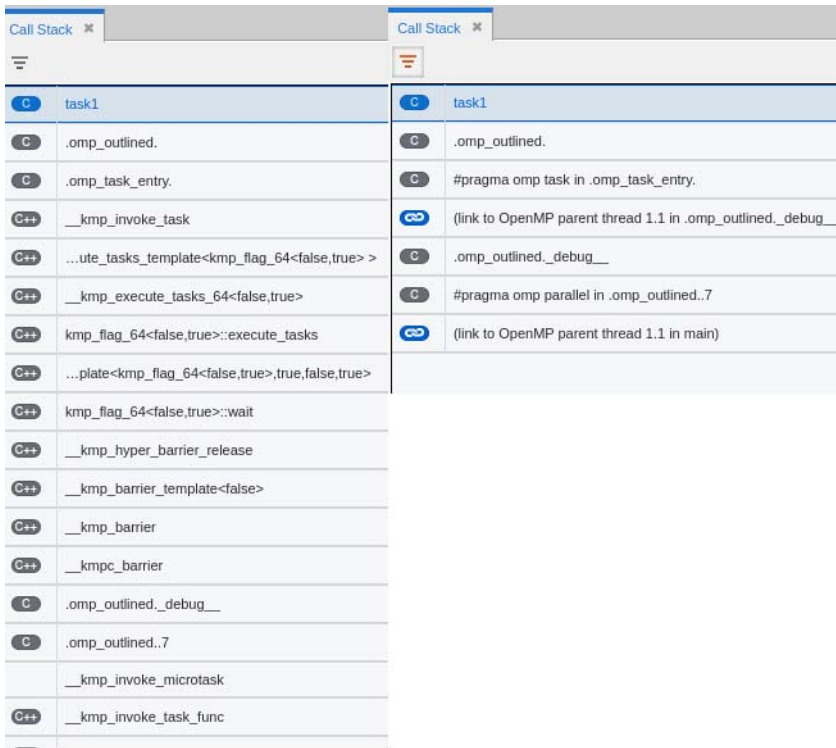
Use the boolean state variable `TV::openmp_debug_enabled`:

```
dset TV::openmp_debug_enabled true
```


Enabling Stack Filtering

Filtering the stack backtrace helps you focus on application stack frames rather than stack frames specific to the OpenMP runtime library, which can clutter the Call Stack view. [Figure 116](#) displays an unfiltered call stack on the left, compared to a filtered stack on the right.

Figure 116, Call Stack Filtering



Enable or disable stack trace filtering using one of the following methods:

- Click the transform button () in the Call Stack view to toggle the transform on or off, *or*
- Enter one of the following commands in the Command Line view:


```
dstacktransform enable
```

```
dstacktransform disable
```

 (This command can also be used to list, add, or delete stack transformations.) *or*
- Use a state variable to control the filtering of the stack:


```
dset TV::stack_trace_transform_enabled true
```

```
dset TV::stack_trace_transform_enabled false (default)
```

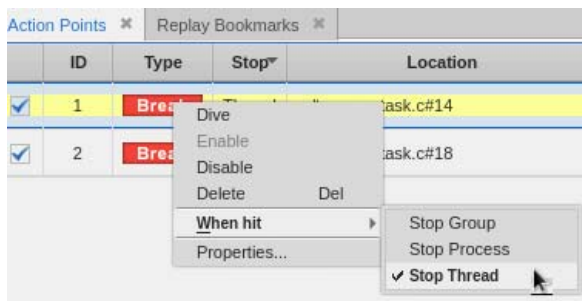
Any of these methods control *all* defined stack transforms, not just for OpenMP, but, for example, Python and user-defined transformations.

Running Your Program

After you load your OpenMP program, enable the OMPD library, set some breakpoints, and start your program, you can examine the OpenMP view and the Call Stack view to analyze the backtrace and see the state of your OpenMP threads.

Setting Breakpoints in an OpenMP Program

When setting breakpoints in an OpenMP program, you may want all OpenMP threads to run to a particular line, in which case, set the breakpoints to stop just the threads that hit the breakpoint rather than all threads in the process, using the context menu in the Action Points view:



The Call Stack

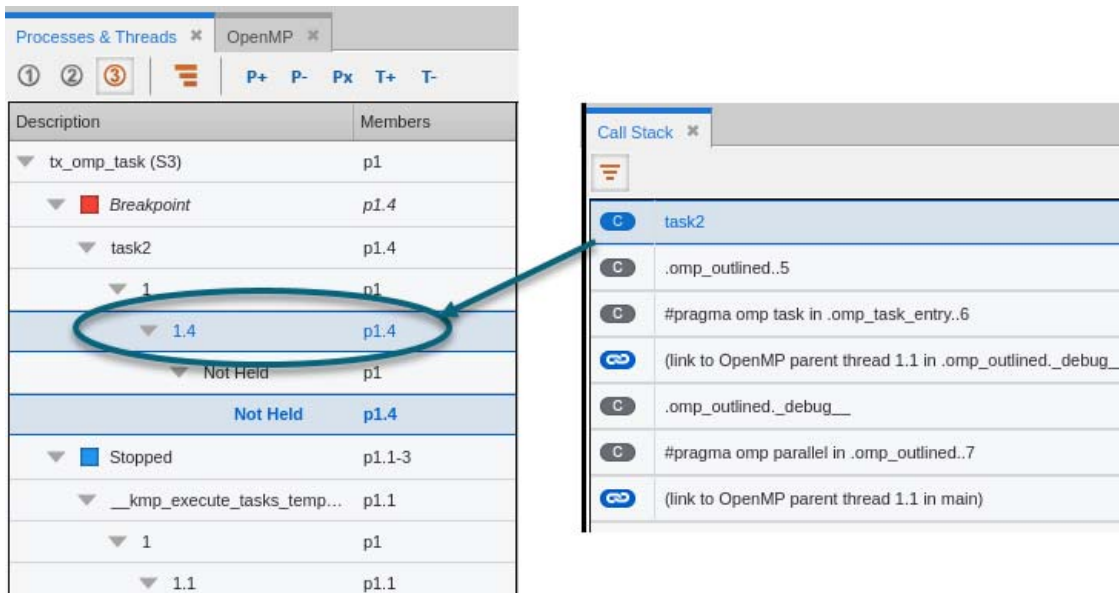
When stack filtering is turned on (see [Enabling Stack Filtering](#)), the Call Stack displays a transformed version of the stack, which includes filtering out, inserting, and annotating stack frames.

How OpenMP Stack Transformations Work in TotalView

When stack filtering is enabled for an OpenMP program that has OMPD support enabled, TotalView:

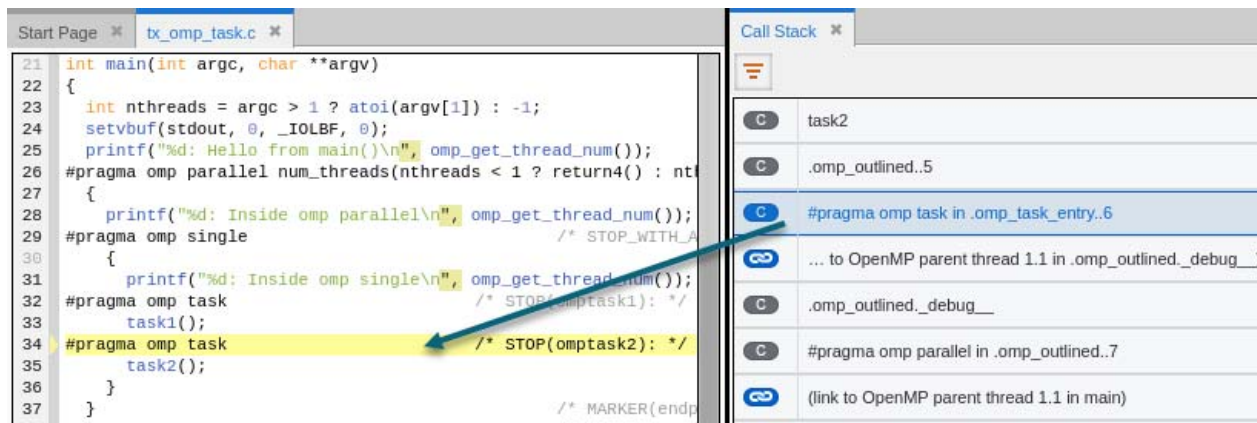
- Filters out OMP runtime frames.
- Annotates parallel and task region function names with **#pragma omp parallel** or **#pragma omp task** for C/C++ programs and **!\$omp parallel** or **!\$omp task** for Fortran programs.
- Inserts "parent thread link" frames, so that clicking on the frame focuses on the encountering thread's generating task frame. Parent thread link frames are inserted only if the parent thread is different than the current thread. If the generating task frame no longer exists in the parent frame, then the debugger focuses on frame 0.

For example, consider an OpenMP program with a single parallel region and multiple threads. When a breakpoint is hit and you focus on a thread in the Processes & Threads view, that thread's backtrace displays in the Call Stack, as usual:

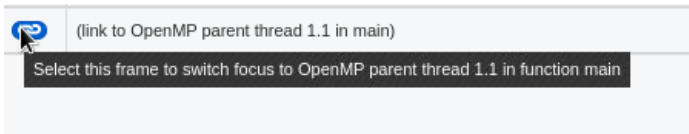


Observe the following:

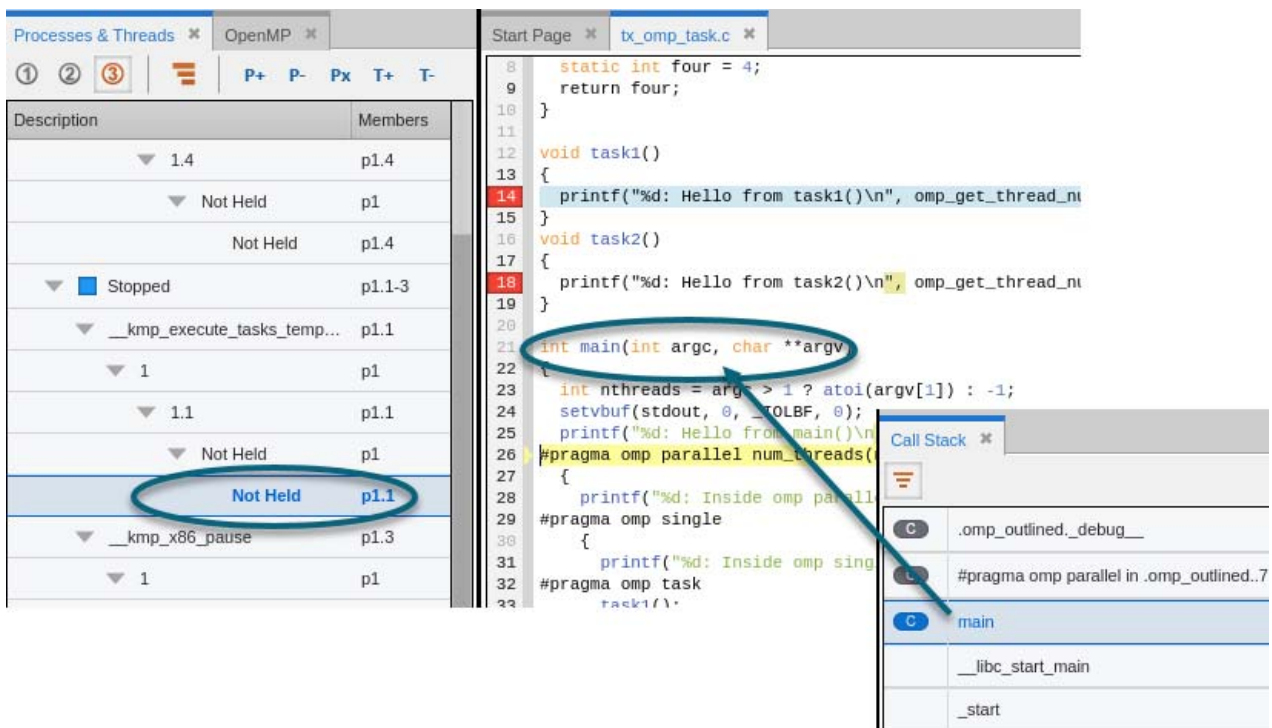
- The Call Stack is filtered, identified by the orange filter icon (☰). Selecting the icon toggles the state of stack filtering.
- The OpenMP runtime frames have been removed from the display.
- Selecting the **#pragma omp task** frame in the Call Stack automatically navigates to the location in the source code containing the **#pragma omp** directive. Selecting the frame will focus on a source line within the parallel or task region, although the exact location depends on the compiler:



- The stack backlink to the parent thread is identified by a link icon (🔗). In this example, the thread that has focus, or 1.4, was invoked by a parallel region in parent thread 1.1 in function **main**, identified by the popup tip when you hover a cursor over its entry in the Call Stack:



Clicking this link focuses the source pane on its parent thread's stack frame for function **main**, which in turn updates the Call Stack for the backtrace for thread 1.1:



This parent/child stack linking continues through the backtrace until the initial thread 1.1, is reached.

The OpenMP View

The OpenMP view displays all the OMPD information provided by the OpenMP Debugging Interface, also available via the `domp` CLI command.

- **OMPD Info Tab**

Information about the OMPD dynamic library (DLL) in use for the selected process.

Processes & Threads × OpenMP ×	
API Version	201811
DLL Version	LLVM OpenMP 5.0 Debugging Library implementing TR 62
DLL Name	/opt/rocm-6.0.0/llvm/lib-debug/libompd.so

Regions | Control Variables | **OMP Info** | Threads | ICVs

CLI command: `domp -ompd`

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

■ **Control Variables Tab**

The OMPD display control variable settings for a selected process. Changing the focus to a different process repopulates this list with that process’s control variables.

Processes & Threads × OpenMP ×	
Variable	Value
KMP_ABORT_DELAY	0
...APTIVE_LOCK_PROPS	'1,1024'
KMP_AFFINITY	'noverbose,warnings,disabled'
KMP_ALIGN_ALLOC	64
..._ALL_THREADPRIVATE	0
KMP_ATOMIC_MODE	2
KMP_A_DEBUG	0

Regions | **Control Variables** | OMPD Info | Threads | ICVs

CLI command: `domp -control_vars`

■ **ICVs Tab**

The internal control variables that maintain runtime state.

ICV Name	Value
▼ Process 11 (global / address space scope)	
affinity-format-var	"OMP: pid %P tid %i thread %n bound to OS proc set {%A}"
cancel-var	0
debug-var	1
display-affinity-var	0
max-task-priority-var	0
num-procs-var	2
stacksize-var	8388608
tool-libraries-var	""
tool-var	1
tool-verbose-init-var	""
▼ Thread 11.1 (thread scope)	
default-device-var	0
dyn-var	0
nthreads-var	"2"
thread-num-var	0
▼ Region 0 (task / implicit task scope)	
bind-var	0
final-task-var	0
implicit-task-var	0
max-active-levels-var	1
run-sched-var	"static,0"
thread-limit-var	2147483647
▶ Region 1 (task / implicit task scope)	

The ICV name is canonicalized and unqualified based on the ICV map extracted from the OMPD library. Any OMP 5.0 OMPD-specific ICV names, which all begin with `ompd-`, are either renamed to their OMP 5.2 name or discarded if the OMP 5.2 name is also defined.

CLI command: `domp -icvs`

■ **Regions Tab**

The nest of parallel and task regions for the entire share group of the selected process. Selecting different regions in the display focuses the source pane on the specific region.

Task Line	Task Function	
▼ / (Parallel Regions)		2:8[0-1.1-4]
tx_omp_task.c#26	.omp_outlined..7	2:8[0-1.1-4]
▼ / (Task Regions)		2:8[0-1.1-4]
▼ tx_omp_task.c#26	.omp_outlined..7	2:8[0-1.1-4]
tx_omp_task.c#32	.omp_task_entry.	2:2[0.2, 1.4]
tx_omp_task.c#34	.omp_task_entry..6	2:2[0-1.1]

CLI Command: `domp -threads -regions`

■ **Threads Tab**

Displays details on all the OpenMP threads in the application.

Thread ID	OMP Thread #	System ID	State/Region#	Wait Id/P	Flags	Task Function	Reentry Addr
▼ 1.1	0	0x7f831fa74740	work_parallel	0x0	-p-	.omp_task_entry..6	0x0000000000000000
			region 0		-p-	.omp_task_entry..6	0x0000000000000000
			region 1	1.1	ip-	.omp_outlined..7	0x00007ffe5135220
			region 2	1.1	i--	<unavailable>	0x00007ffe5135660
1.2	1	0x7f831e6ac780	wait_barrier	0x0	ip-	.omp_outlined..7	0x00007f831e6abca0
			region 0	1.2	ip-	.omp_outlined..7	0x00007f831e6abca0
			region 1	1.1	i--	<unavailable>	0x00007ffe5135660
▶ 1.3	2	0x7f831dea800	wait_barrier	0x0	ip-	.omp_outlined..7	0x00007f831dea9d20
▶ 1.4	3	0x7f831d6a8880	work_parallel	0x0	-p-	.omp_task_entry.	0x0000000000000000

Annotations in the image:

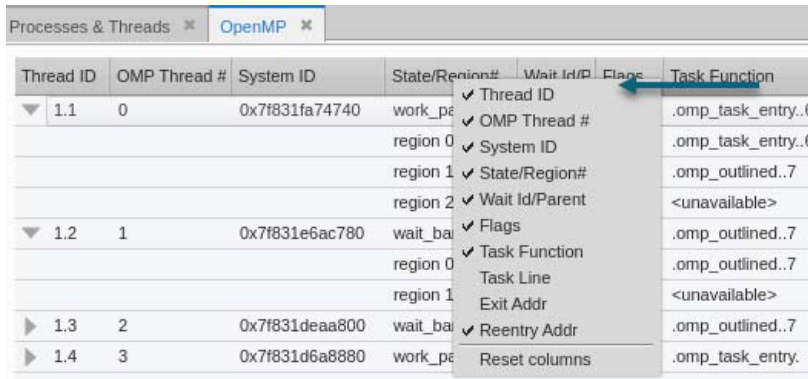
- innermost region: points to region 0 of thread 1.1
- outermost region: points to region 2 of thread 1.1
- the parent of region 1: points to region 1 of thread 1.2

Note that:

- A thread's first line at Region 0 identifies its current (innermost) region, displaying the basic OpenMP thread information.
Expand any top level thread to display the thread's nest of parallel and task regions.
- Once expanded, each line shows the enclosing regions' task flags and distance from the current region, identified as an integer in the State/Region# column; the last line identifies the outermost region. The regions may cross thread boundaries from a child thread to its parent thread.

In the example above, **Thread 1.2's region 1** has a parent of **1.1**, which created the region being executed.

- Right-clicking on the header shows or hides columns.



This setting is persisted between sessions.

CLI Command: `domp -threads`. This command has numerous options that are all included as columns within this Threads tab.

Table 11: OpenMP Threads Tab Column Description

Column	Description
Thread ID	The TotalView process and thread id or the MPI Rank and thread ID if this is an MPI program.
OMP Thread #	The thread's OpenMP thread number within its current team.
System ID	The thread's Kernel ID or user thread id.
State/Region #	The thread's OpenMP state for the top-level thread. This state is defined by the OpenMP standard. If the expander on the top-level thread is clicked, the thread's nest of parallel or task regions is displayed, and the column shows the index of the region.
Wait Id/Parent	The thread's OpenMP current wait ID or Parent ID. The Parent ID (which includes the debugger process and thread ID) is the ID of the encountering thread for the region.

Table 11: OpenMP Threads Tab Column Description

Column	Description
Flags	<p>OpenMP Flags. The following flags are reported:</p> <ul style="list-style-type: none"> ■ i: The task is an implicit task. An implicit task has no scheduling task. No corresponding routine in the OpenMP runtime reports this information. ■ p: The thread is in an active parallel region. This corresponds to the <code>omp_in_parallel()</code> predicate in the OpenMP runtime. ■ f: The task is a final task. This corresponds to the <code>omp_in_final()</code> in the OpenMP runtime. <p>When true, the flag is displayed by just its letter. If false, the flag is displayed as a hyphen ("-"). If the flag could not be fetched, a "?" is displayed.</p>
Task Function	The subroutine name for the task function for the implicit or explicit task region. If the task function cannot be determined, "<unavailable>" is displayed.
Task Line	The file name and line number of the task function.
Exit Addr	The thread's stack address corresponding to the location where control exited the OpenMP runtime to execute the task user code. The exit address is 0 for the root region.
Reentry Addr	The thread's stack address corresponding to the location where control reentered the runtime from the user task code. The reentry address is 0 for a leaf.

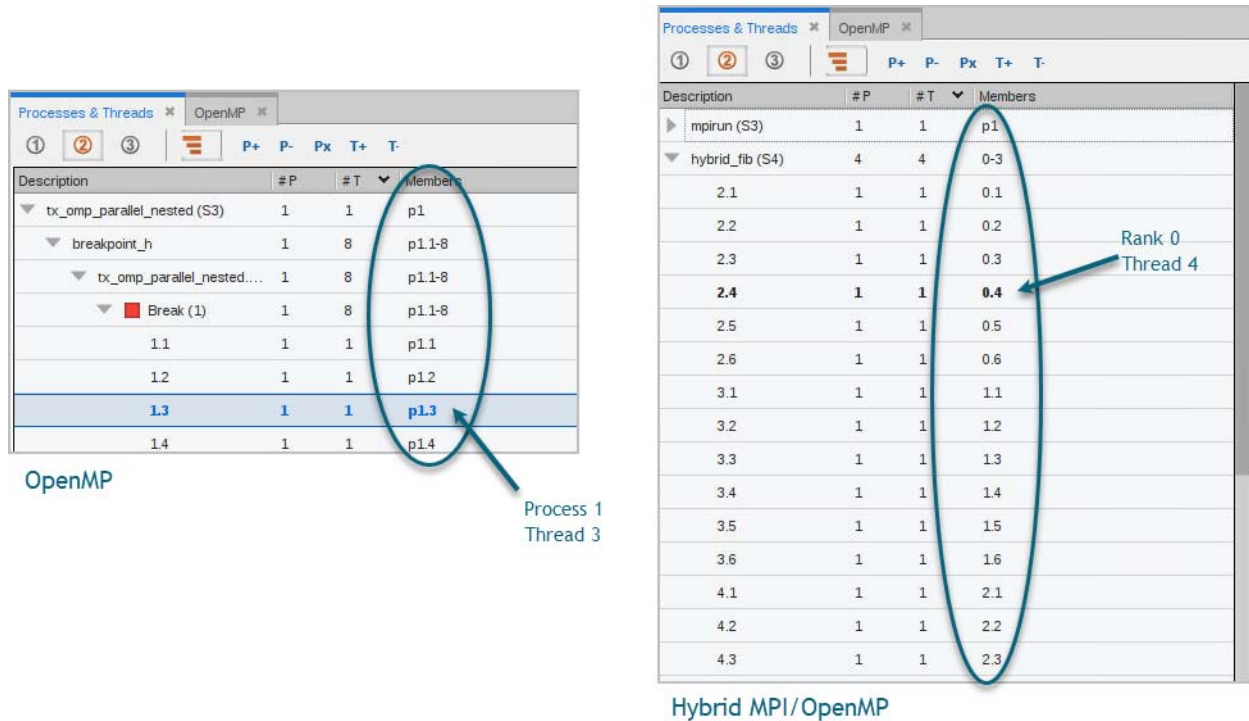
Hybrid Programming: Combining OpenMP with MPI

NOTE: If you plan to incorporate OMPD in an MPI program, some configuration is necessary, in particular, setting the environment variable `OMP_DEBUG`. See [Enabling OpenMP Debugging](#) for detail.

OpenMP shares memory between threads on a single node, while MPI can launch separate tasks and communicate between them across multiple nodes, but does not share memory *between* nodes. You can combine the use of these two technologies in a hybrid programming model to gain both shared memory and the ability to distribute tasks across multiple cores. In this case, each process is an MPI process, while the threads within those processes may be parallelized using OpenMP.

Consider this example. [Figure 117](#) compares the Threads tab in an OpenMP program (left) with one in a hybrid program (right).

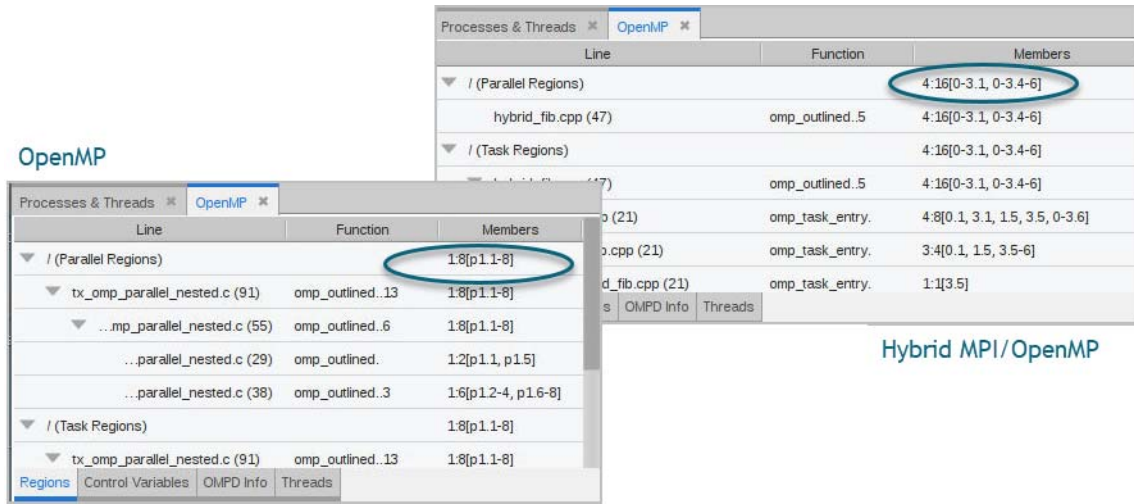
Figure 117, Hybrid programming example, Threads tab



The **Members** tab in the pure OpenMP program displays a single process, identified by the “p1.x” process ID, along with multiple threads. The same tab in the hybrid program displays the MPI rank rather than the process ID, followed by the thread ID.

The **Regions** tab of the OpenMP view also reflects the MPI rank under the Members column, shown in [Figure 118](#).

Figure 118, Hybrid programming example, OpenMP view's Regions tab



In this example, the hybrid MPI/OpenMP program (top right) has 4 MPI ranks with 16 threads.

Controlling fork, vfork, and execve Handling

- The `exec_handling` and `fork_handling` Command Options and State Variables
- Exec Handling
- Fork Handling
- Example

The `exec_handling` and `fork_handling` Command Options and State Variables

TotalView allows you to control how the debugger handles system calls to `execve()`, `fork()`, `vfork()`, and `clone()` (when used without the `CLONE_VM` flag).

- When calling `fork()`, `vfork()`, and `clone()`, choose to either attach or detach from the new child process.
- When calling `execve()`, choose either to continue the new process, halt it, or ask what action to take.

This behavior is controlled by two CLI state variables and two command options. Set the state variables to control the default behavior for TotalView. Use the command options when starting TotalView to control the behavior for a particular debugging session. The command options override the state variable settings.

Table 12: `exec_handling` and `fork_handling` Command Options and State Variables

Command Options	CLI State Variables
<code>-exec_handling exec-handling-list</code>	<code>TV::exec_handling exec-handling-list</code>
<code>-fork_handling fork-handling-list</code>	<code>TV::fork_handling fork-handling-list</code>

The lists `exec-handling-list` and `fork-handling-list` are Tcl lists of `regex` and `action` pairs. Each `regex` is matched against the process's name to find a matching action, which determines how to handle the exec or fork event.

RELATED TOPICS

The state variables	<code>TV::exec_handling exec-handling-list</code> and <code>TV::fork_handling fork-handling-list</code>
<code>totalview</code> command options	<code>-exec_handling exec-handling-list</code> and <code>-fork_handling fork-handling-list</code>
Setting breakpoints when using <code>fork()</code> and <code>execve()</code>	Setting Breakpoints When Using the <code>fork()/execve()</code> Functions
Linking with the dbfork library	Linking with the dbfork Library

Exec Handling

When a process being debugged execs a new executable, the debugger iterates over *exec-handling-list* to match the original process name (that is, the name of the process before it called exec) against each *regexp* in the list. When it finds a match, it uses the corresponding *action*, as follows:

Action	Description
halt	Stop the process
go	Continue the process
ask	Ask whether to stop the process

If a matching process name is not found in the *exec-handling-list*, the value of the **TV::parallel_stop** CLI state variable preference is used.

Fork Handling

When first launching or attaching to a process, the debugger iterates over *fork-handling-list* to match the process name against each *regexp* in the list. When it finds a match, it uses the corresponding *action* to determine how future fork system calls will be handled, as follows:

Action	Description
attach	Attach to the new child processes.
detach	Detach from the new child processes.

If a matching process name is not found in the *fork-handling-list* list, TotalView handles **fork()** based on whether the process was linked with the **dbfork** library and the setting of the **TV::dbfork** CLI state variable preference.

Example

It's important to properly construct the *exec-handling-list* and *fork-handling-list* list of pairs, so that the list is properly quoted for Tcl or the shell. Generally, enclose the list in curly braces in the CLI, and enclose it in single quotes in the shell.

Note that the regular expressions are not anchored, so you must use "^" and "\$" to match the beginning or end of the process name.

Calling exec:

This example configures TotalView to automatically continue the process (without asking) when bash calls exec, but to ask when other processes call exec, using the following **dset** CLI command or **totalview** command option:

```
dset TV::exec_handling {{{^bash$} go} {. ask}}
totalview -exec_handling '{{^bash$} go} {. ask}'
```

Above, the regexp is wrapped in an extra set of curly braces to make sure that Tcl does not process the "\$" as a variable reference.

Calling fork:

This example configures TotalView to attach to the child process when a process containing the name "tx_fork_exec" calls fork, but to detach from other forked processes, using the following **dset** CLI command or **totalview** command option:

```
dset TV::fork_handling {{tx_fork_exec attach} {. detach}}
totalview -fork_handling '{tx_fork_exec attach} {. detach}'
```

An example session:

```
% totalviewcli -verbosity errors \
  -exec_handling '{{^bash$} go} {. ask}' \
  -fork_handling '{tx_fork_exec attach} {. detach}' \
  -args \
    bash -c 'tx_fork_exec tx_hello'
dl.<> co
Parent done ....
Child is calling execve ...
Process bash<tx_fork_exec>.1 has exec'd /path/to/tx_hello.
  Do you want to stop it now?

: yes
dl.<> ST
1      (0)                Nonexistent [bash]
2      (20053)            Stopped      [bash<tx_fork_exec><tx_hello>.1]
  2.1  (20053/20053)     Stopped      PC=0x7f70517dd210
dl.<>
```

Group, Process, and Thread Control

Overview

When debugging a parallel program, you often need fine-grained control over the program's execution. TotalView organizes a program's processes and threads into groups, a design that provides you the ability to precisely select a focus for any execution control operation.

The focus of any debugging command determines the set of processes and threads that will be affected by that command, whether that's a single thread, a single process, a group of processes or threads, or multiple groups—and anything in between.

Select the focus either in the UI using its **Focus** control, or in the CLI using the **dfocus** command. Debugger commands determine the *arena* (the specific collection of thread(s) or process(es) within the target) by looking at the selected focus.

The arena includes a *thread of interest*, a *process of interest*, and a *group of interest*. It also includes a *width*, which provides a way to specify the set of processes and threads on which certain debugger commands will act. For example, the width determines whether a command operates on a single thread, a process, or an entire group. It may also include a *group specifier* to identify which group the command should act on.

Thread, Process, and Group of Interest

- **TOI**—The Thread of Interest is selected by the user. In the UI, the TOI is typically identified by selecting a particular thread within the Processes and Threads view. In the CLI, the TOI is the primary thread that will be affected by a command. For example, the **Step** command always steps the thread of interest, but it may optionally run other threads in the process and other processes in the group.
- **POI**—The Process of Interest is the process that contains the TOI
- **GOI**—The Group of Interest is specified by the arena and the POI/TOI

An understanding of groups, arenas, P/T sets, and width will help you to set a precise focus to effectively debug complex, parallel programs with any number of threads and processes.

This chapter discusses:

- Groups in TotalView
- Arenas and P/T Sets
- Stepping and Program Execution

Groups in TotalView

TotalView organizes all processes and threads into groups, which provides a mechanism by which you can precisely select the processes and threads on which a command will operate. Some groups contain (possibly multi-threaded) processes, while others contain just threads. The creation of these process or thread groups is largely automatic and occurs in the debugger without explicit input from the user. TotalView uses these groups for any operations that involve multiple threads or processes.

These automatically created groups offer a set of useful collections of threads and processes that can be selected as the group of interest (GOI) for certain debugger commands. You can select these either in the UI via the **Focus** menu, or, for finer control, in the CLI by providing a *P/T set* parameter to the **dfocus** or other commands. (More on P/T sets later, but a P/T set is simply a list of arenas.)

The GOI is the group that was specified in the focus and will therefore determine the set of processes and threads affected by the command. Most commands define a default group that makes sense for the command. For example, execution commands (**Go**, **Stop**, **Step**, etc.) default to the Control Group, and data-oriented or breakpoint commands default to the Share Group.

What Is a Group?

Groups are fundamental to multiprocess and multithread control in TotalView. Each process or thread group is defined by a unique numeric ID. A group is either a *process group* containing only processes, or a *thread group* containing only threads.

TotalView automatically defines several groups that are used in common operations, including: the *control group* (all processes in a program, which are normally controlled together); the *share group* (all processes in the program that share a single image); the *workers group* (all worker threads in a control group); the *lockstep group* (all threads that share the same PC in a share group); and the *all group* (all processes in the debug session).

Internally, a group is just a table of processes or threads. Groups that TotalView defines are automatically updated as new threads and processes are created or old threads and processes exit. The contents of these groups therefore vary during program execution. Users may also define custom groups. These have identical properties as the fundamental groups of TotalView, but are not automatically maintained by the debugger.

A group contains no information regarding the thread or process of interest, nor does it identify the level at which a command should operate (i.e., should it operate on only the thread, the entire process, or the group of processes?). Use an *arena* to specify the target, or focus, of a command. An arena in TotalView is a group *plus* a specific process and thread, along with an optional *width* indicator to define the subset that should be affected by a command. A P/T set is composed of a list of arena specifiers. For more information, see [Arenas and P/T Sets](#).

Types of Groups Created by TotalView

TotalView automatically organizes your processes and threads into the following predefined groups:

- **Process Groups:**

By default, a process group contains all threads in all processes that are members of the group.

- **Control Group:** All processes in a program. These processes can be local or remote. If your program uses processes that it did not create, TotalView places them in separate control groups by default. For example, a client/server program with two distinct executables that run independently will be placed into separate control groups. In contrast, processes created by `fork()/exec()` are in the same control group.

You don't have to accept these defaults. You can specify an existing control group for newly attached or created processes, and move existing processes to a different control group.

- **Share Group:** All processes in a control group that share the same image, i.e., the same executable filename and path. In most cases, parallel programs have more than one share group. Share groups, like control groups, can be local or remote.
- **All Group:** All processes in a debugging session.

- **Thread Groups:**

A thread group can contain a subset of the threads from any process.

- **Workers Group:** All worker threads in a control group. These threads can reside in more than one share group. TotalView automatically places all threads that are not manager threads into the workers group. Manager threads, i.e., those created by the `pthread` package to manage other threads, do not execute code and cannot normally be controlled individually. (Note that most modern systems do not use manager threads.)
- **Lockstep Group:** All threads in the share group that are at the same PC (program counter) as the thread of interest (TOI). This group is a subset of a workers group and will typically include threads from multiple processes.

How TotalView Creates Groups

TotalView places processes and threads into groups as your program creates them, except for the lockstep groups, which are created or changed whenever a process or thread hits an action point or is stopped for any reason. Here is a rundown on how these groups are created.

As soon as a program starts, TotalView creates the two process groups: a control group and a share group. It also creates a workers group, containing the thread in the `main()` routine. There is no lockstep group yet, since lockstep groups contain only stopped threads. As new threads are spawned and begin to run, they also are added to these three groups.

Let's consider a few scenarios common to parallel program debugging.

Groups Created When a Program Calls `fork()/exec()`

TotalView can automatically attach to child processes created when a process being debugged calls `fork()` or `vfork()`. Here's a typical flow:

1. TotalView is started on a program named "a.out". TotalView names the process "a.out" and automatically places it in Control Group 1 and Share Group 2.
2. During execution, process "a.out" makes a `fork()` system call to create a child process. If TotalView is configured to automatically attach to child processes, it names the process "a.out.1". The child process remains in Control Group 1 because of the parent/child relationship, and it also remains in Share Group 2 because it is executing the program named "a.out".
3. During execution, process "a.out.1" (the child process) makes an `execve()` system call on a program named "b.out". TotalView renames the child process to "a.out<b.out>.1". The child process remains in Control Group 1, but is placed in Share Group 3 because it is now executing a program named "b.out".
4. All the threads in processes "a.out" and "a.out<b.out>.1" are placed in Worker Group 4, because neither program creates manager threads, and both processes are members of the same Control Group.

RELATED TOPICS

Setting breakpoints when acquiring processes using `fork` or `exec` [Setting Breakpoints When Using the `fork\(\)/execve\(\)` Functions](#)

Use TotalView `TV::exec_handling` and `-exec_handling` options to control whether new processes are stopped or allowed to continue running [Exec Handling](#)

Choose to either attach or detach from the new child process. [Fork Handling](#)

Groups Created for MPI Programs

TotalView can automatically attach to MPI processes created by an MPIR starter program (such as, `mpirun`), for example, when you launch the MPI starter process under TotalView. Here's a typical flow:

1. TotalView is started on the MPI starter program named "mpirun". TotalView names the process "mpirun" and automatically places it in Control Group 1 and Share Group 2.
2. During execution, the "mpirun" process launches the MPI processes, and waits for the debugger to attach to the MPI processes before allowing them to execute the application code. The MPI application is an MPMD-style application, where some of the MPI processes run the executable "a.out", and others run the executable "b.out".
3. When TotalView detects that the "mpirun" process has launched the MPI processes, it automatically attaches to them. The MPI processes are placed in Control Group 1 with the MPI starter process because effectively the MPI starter process is the parent of the MPI processes. The MPI starter process remains in Share Group 2, but two new share groups are created for the MPI processes: Share Group 3 for the MPI processes executing "a.out" and Share Group 4 for the MPI processes executing "b.out".
4. All the threads in the MPI starter process and MPI processes are placed in Worker Group 5, because none of the processes create manager threads and all processes are members of the same Control Group.

NOTE: Note that if you have multiple share groups in your debugging session, you can set the focus to "Share group" in the Focus menu, and only the share group that contains the TOI will run, while the others will remain steady. See [Executing a Single Share Group](#) for an example.

Groups Created for CUDA Programs

The take home point for group creation when debugging CUDA programs is that CUDA threads are placed in *the same share group* as are their host Linux processes. Because CUDA threads and the host process are all in the same share group, you can create pending or sliding breakpoints on source lines and functions in the GPU code before the code is loaded onto the GPU. This organization allows support for a unified Source view display, where the breakpoint and source line information of the code running on the GPU is unified with the code running on the host CPU.

RELATED TOPICS

How groups are created when debugging CUDA programs

[The TotalView CUDA Debugging Model](#)

More on the unified Source view display

[Unified Source View and Breakpoint Display and Unified Source View Display](#)

Executing a Single Share Group

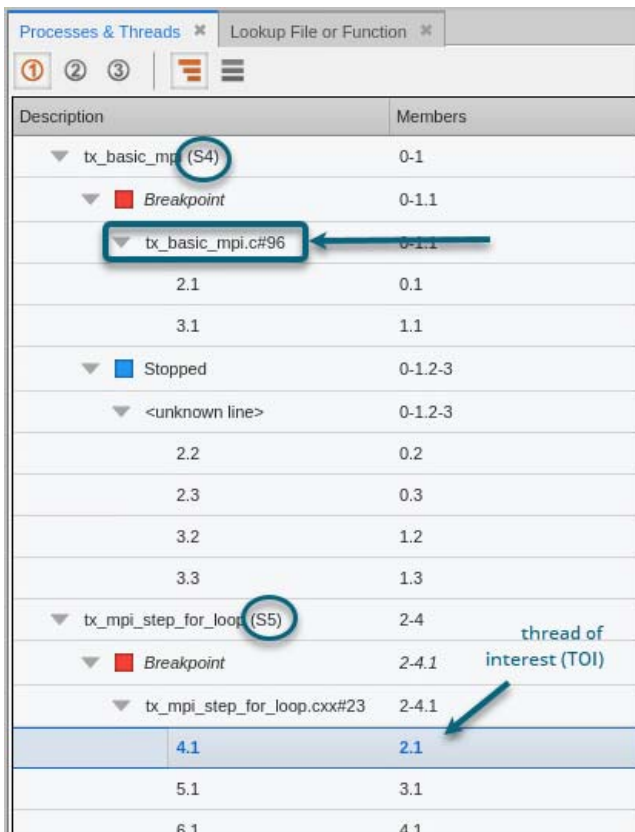
Some MPI implementations support a multiple programs/multiple data mode (MPMD). With this mode, you can launch separate executables with different MPI arguments, and TotalView will place each executable's jobs in their own share group.

In this mode, you can then use the Focus menu to isolate a single share group for execution so that all other share groups remain steady. The default behavior is to allow all processes in all share groups (the control group) to run.

NOTE: To view the separate share groups in the Processes & Threads view, be sure that **Share Group** is selected from the view's Configure panel.

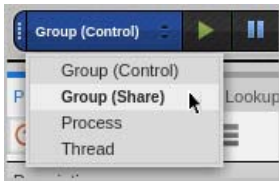
Consider this session with two share groups **S4** and **S5**.


Figure 119, Two share groups in a session



Note that some processes in the **S4** share group are stopped at line 96. Share group **S5** is stopped at breakpoint on line 23 and contains the thread of interest (TOI), **2.1**.

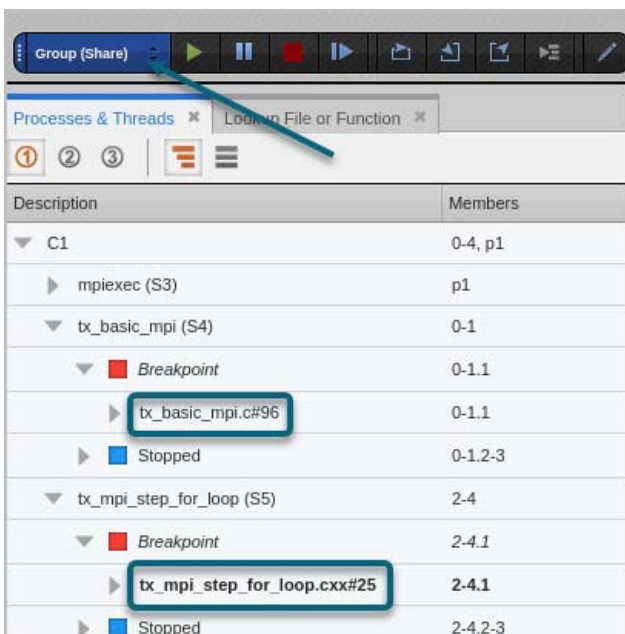
Choosing **Group (Share)** from the **Focus menu** will advance just the **S5** share group because it contains the thread of interest (TOI), **4.1** (which has a rank of 2.1).



With the focus on the share group, select **Go** from the toolbar (), then note that share group **S4** did not advance, but share group **S5** advanced to line 25.

NOTE: To ensure that advancing your program considers the focus selected under the Focus menu, use the **Go** command in the toolbar as this example shows. The **Go** commands available under the menu items **Group**, **Process**, or **Thread** operate only on the entire control group and are unrelated to the Focus menu.

Figure 120, Share group breakpoint hopping



NOTE: Because the executables in different share groups often need to interact with each other, be aware that holding processes in another share group can result in a deadlock situation; before taking advantage of this capability, be sure that the share groups are independently executing to avoid deadlock.

Single Stepping While Focused on a Share Group

When the focus is on a share group and you choose **Go** from the toolbar, all the processes in that share group are allowed to run, while those in any other share groups hold steady.

Stepping operations behave differently, however. If you want to single step a particular share group through an application, you may need to hold the processes in other share groups or they will be allowed to run freely while you try to move your TOI to a goal.

Holding a Process

To hold a process, focus on the process in the Processes & Threads view, and select **Process > Hold** from the menu. If you have multiple processes, rather than selecting each one individually and using the **Process > Hold** menu item, you can use the CLI to select all of them at once.

For example, this command holds all the processes in the share group containing process 2:

```
f gS2 dhold -process
```

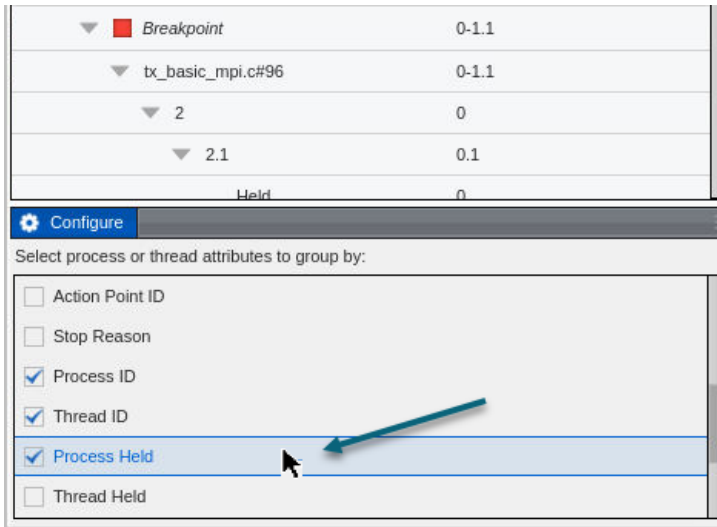
As a shortcut, you could use:

```
f gS2 HP
```

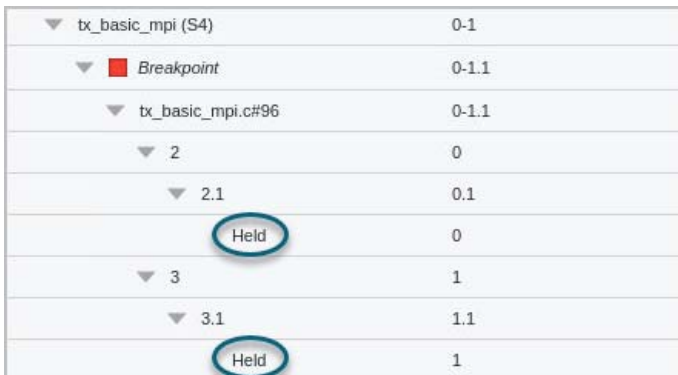
Viewing Held Processes

To verify that the processes are held, turn on the Process Held option in the Configure panel of the Processes & Threads view:

Figure 121, Process Held option in the Processes & Threads View



The view then identifies held or unheld processes:



You can also view the status in the CLI. This command displays the status of all groups:

`f g st`

The CLI displays an **H** for held processes:

```
d1.<> f g st
1 (29116) Running [/etnus/packages/mware/openmpi/openmpi/4.1.0/x86-64-1:
  1.1 (29116/0x7f9a0c064740) Running PC=0x7f9a0a5c9bb9
  1.2 (29116/0x7f9a05a57700) Running PC=0x7f9a0a5d6947
  1.3 (29116/0x7f9a04e48700) Running PC=0x7f9a0a5cbd1f
  1.4 (29116/0x7f99ffdec700) Running PC=0x7f9a0a5cbd1f
2 (29120@ubuntu1804-x8664) H Breakpoint [mpiexec<tx_basic_mpi>.0]
  2.1 (29120/0x7f45f3b6a740@ubuntu1804-x8664) Breakpoint PC=0x5607c0d75fa2, [/nfs.
  2.2 (29120/0x7f45f0c7f700@ubuntu1804-x8664) Stopped PC=0x7f45f3370bb9
  2.3 (29120/0x7f45e97f6700@ubuntu1804-x8664) Stopped PC=0x7f45f337d947
3 (29121@ubuntu1804-x8664) H Breakpoint [mpiexec<tx_basic_mpi>.1]
  3.1 (29121/0x7f28cb8c6740@ubuntu1804-x8664) Breakpoint PC=0x55f89569afa2, [/nfs.
  3.2 (29121/0x7f28cb0cbb9@ubuntu1804-x8664) Stopped PC=0x7f28cb0cbb9
```

Arenas and P/T Sets

Many debugger commands operate on a thread, process, or group based on the defined *arena*, i.e., the target, or focus, provided to a command. An arena in TotalView is comprised of a group, a specific process and thread, and a *width* indicator to define the subset that should be affected by a command.

Arenas are contained within P/T sets, which are lists that can contain any number of arenas.

Arena Specifiers in a P/T Set

A P/T set in the CLI is a Tcl list, usually enclosed in curly braces (`{ }`), and composed of a list of arena specifiers, or a P/T expression composed of arena specifiers and operators. An arena defines a collection of processes and threads that are the target of an action. Many CLI commands can act on one or more arenas. For example, the following **dfocus** command uses two arena specifiers to set the focus on two arenas: process 1 and process 2:

```
dfocus {p1 p2}
```

TotalView identifies one thread in the arena as the thread of interest (TOI). However, the collection defined by the arena actually includes all the processes and threads in the group associated with the arena, i.e., the group of interest (GOI).

An arena specifier includes a *width* and a TOI. (Widths are discussed in detail in [Process and Thread Width Specifiers in a P/T Set](#).) It may also include a group. In an arena, the TOI specifies a target thread, while the width identifies a subset of the collection of processes and threads to be affected by the command.

For a P/T set that contains multiple arenas, most commands iterate over each arena and act specifically on each. Some CLI output commands, however, combine arenas and act on them as a single target.

P/T Set and Arena Identifier Syntax

Create a P/T set in the CLI by either:

- Entering arena identifiers within braces (`{ }`) to create a Tcl list.
- Using Tcl commands that create and manipulate the list of arena identifiers.

P/T sets are then used as arguments to a command. If you're entering one element, you usually do not have to use the Tcl list syntax.

For example, the following list contains arena specifiers for process 2, thread 1, and process 3, thread 2:

```
{p2.1 p3.2}
```

If you do not explicitly specify a P/T set in the CLI, TotalView defines a target set for you. In the UI, the default is the process and thread in the Processes & Threads view. In the CLI, this default is indicated by the focus, which is shown in the default CLI prompt. (See [About the CLI Prompt](#).)

To save a P/T set definition for later use, assign the specifiers to a Tcl variable, for example:

```
set myset {g2.3 t3.1}
dfocus $myset dgo
```

The **dfocus** command returns the default focus, so you can save this value for later use, for example:

```
set save_set [dfocus]
```

Defining the Thread of Interest (TOI) in an Arena Specifier

The TOI is specified as **p.t**, where **p** is the TotalView process ID (DPID) and **t** is the TotalView thread ID (DTID). The **p.t** combination identifies the process of interest (POI) and thread of interest (TOI). The TOI is the primary thread acted on by a TotalView command. For example, while the **dstep** command always steps the TOI, it may also run the rest of the threads in the POI and step other processes in the group.

In addition to using numerical values, you can also use two special symbols:

- The less-than character (<) indicates the *lowest numbered worker thread* in a process, and is used instead of the DTID value. If, however, the arena explicitly names a thread group, the < symbol means the lowest numbered member of the thread group. This symbol lets TotalView select the first user thread, which is not necessarily thread 1.
- A dot (.) indicates the current set. Although you seldom use this symbol interactively, it can be useful in scripts.

Process and Thread Width Specifiers in a P/T Set

You can define a P/T set in two ways. If you're not manipulating groups, the format is as follows:

```
[width_specifier][dpid][.dtid]
```

[Group Specifiers in P/T Sets](#) extends this format to include groups. When using P/T sets, you can create sets with just width indicators or just group indicators, or both.

For example, **p2.3** indicates process 2, thread 3.

You can leave out parts of the P/T set if your entry is unambiguous. A missing width or DPID is filled in from the current focus. A missing DTID is always assumed to be <. For more information, see [Naming Incomplete Arenas](#).

The *width_specifier* is a lowercase letter that indicates which processes and threads are part of the arena and therefore the target of a command.

Table 13: Arena width specifiers

Width specifier	Target of command
t	Thread width Only the thread of interest (TOI).
p	Process width All threads in the group of interest (GOI) that are in the process of interest (POI). This may be a subset of the threads in the POI.
g	Group width All threads in the GOI, typically spanning multiple processes.
a	All processes and threads All threads known to TotalView.
d	Default width Depends on the default for each command. This is also the width to which the default focus is set. For example, the dstep command defaults to process width (run the process while stepping one thread), and the dwhere command defaults to thread width.

A useful way to think of the width is that it is a *restriction* on the rest of the arena specifier. The collection of threads on which to operate is the *arena restricted by the width*. A width of **a** includes everything. A width of **g** restricts it to just the group in the arena. A width of **p** restricts it further, to just the threads which are also members of the process and group in the arena. Finally, a width of **t** restricts the collection to just the TOI.

For example, consider how width impacts the **dstep** command. This command always requires a TOI, but the processes or threads on which this command acts differ:

- Step just the TOI during the step operation (thread-level single-step).
- Step the TOI and step all threads in the process that contain the TOI (process-level single-step).
- Step all processes in the group that have threads at the same PC as the TOI (group-level single-step).

For more information, see [Stepping and Program Execution](#).

Group Specifiers in P/T Sets

This section extends the arena specifier syntax to include groups, building on the discussion in [Process and Thread Width Specifiers in a P/T Set](#).

If you do not include a group specifier, the default is the control group that contains the POI. The CLI displays a target group in the focus string only if you set it to something other than the default group.

You most often use target group specifiers with the stepping commands, as they give these commands more control over what to step.

Add groups to an arena specifier using this format:

```
[width_specifier][group_specifier][dpid][.dtid]
```

If you enter only one element, TotalView determines other values based on the current focus. Identify a group using a letter, number, or name.

Identifying a Group Using a Letter

Use one of TotalView's predefined sets, each identified by a letter. For example, the following command sets the focus to the workers group:

```
dfocus W
```

Table 14 defines the group specifiers, which must be in uppercase:

Table 14: Arena group specifiers

Group specifier	Target of command
C	Control group All processes in the control group containing the process of interest.
D	Default control group All processes in the control group containing the process of interest. This is the same as the C group specifier except that it's not displayed with the arena.
S	Share group The set of processes in the control group that have the same executable as the process of interest.

Table 14: Arena group specifiers

Group specifier	Target of command
W	Workers group The set of all worker threads in the control group containing the process of interest, which typically contains threads from multiple processes.
L	Lockstep group A set that contains all threads in the share group that have the same PC as the arena's thread if interest. If you step these threads as a group, they proceed in lockstep.

Identifying a Group Using a Number

You can identify a group by the number TotalView assigns to it. The following example sets the focus to group 3:

```
dfocus 3/
```

The trailing slash marks this as a group number instead of a DPID. The slash character is optional if you're using a group letter specifier. However, you must use it as a separator when entering a numeric group ID and a **dpid.dtid** pair. For example, the following example identifies process 2 in group 3:

```
p3/2
```

Identifying a Group Using a Name

You can name a set that you define. Enter this name with slashes. The following example sets the focus to the set of threads contained in process 3 that are also contained in a group called **my_group**:

```
dfocus p/my_group/3
```

Arena Specifier Examples

Naming Incomplete Arenas

You can omit parts of the arena specifier if the meaning remains unambiguous. A missing width, group, or process ID will be assumed, based on the current focus. A missing thread ID is always assumed to be **<**. Some examples:

- If you omit the DTID, TotalView uses the default **<**, where **<** indicates the first worker thread in process 1:

```
d1.<
```

If, however, the arena explicitly names a thread group, **<** means the first thread in the thread group.

- If you don't use a width, TotalView uses the width from the current focus.
- If you don't use a DPID, TotalView uses the DPID from the current focus.
- If you set the focus to a list, there is no longer a default arena. This means that you must explicitly name a width and a DPID.

TotalView does not use the DTID from the current focus, since the DTID is a process-relative value.

- A dot before or after the number specifies a process or a thread. For example, **1.** is clearly a DPID, while **.7** is a DTID.

If you type a number without a dot, the CLI most often interprets the number as being a DPID.

- If the width is **t**, you can omit the dot. For instance, **t7** refers to thread 7.
- If you enter a width and don't specify a DPID or DTID, TotalView uses the DPID and DTID from the current focus.

If you use a letter as a group specifier, TotalView obtains the rest of the arena specifier from the default focus.

- You can use a group ID or tag followed by a **/**. TotalView obtains the rest of the arena from the default focus.

Focus merging can also influence how TotalView fills in missing specifiers. For more information, see [Merging Focuses](#).

Combining Arena and Group Specifiers

Table 15 defines the selected target when using arena and group specifiers to step a program:

Table 15: Combining Arena and Group Specifiers

Arena specifier combination	Target of command
aC	All processes and threads. (The group specifier is not relevant when you use the all "a" width specifier.)
aS	
aW	
aL	
gC	All processes in the thread of interest's (TOI)'s control group.
gS	All processes in the TOI's share group.
gW	All worker threads in the control group that contains the TOI.
gL	All threads in the share group that are currently at the same PC as the TOI.

Table 15: Combining Arena and Group Specifiers

Arena specifier combination	Target of command
pC pS	All threads in the process of interest (POI).
pW	All worker threads in the POI.
pL	All threads in the POI that are currently at the same PC as the TOI.
tC tS tW tL	Just the TOI. The t specifier overrides the group specifier, so all of these specifiers resolve to the current thread.

Here are additional examples that add DPID and DTID numbers to the raw specifier combinations from [Table 15](#):

pW3

All worker threads in process 3.

gW3

All worker threads in the control group that contains process 3. The difference between this and **pW3** is that **pW3** restricts the focus to just process 3.

gL3.2

All threads in the same *share* group as process 3 that are executing at the same PC as thread 2 in process 3.

3

Specifies process 3. The arena width, POI, and TOI are inherited from the enclosing P/T set, so the exact meaning of this specifier depends on the previous context.

While the slash is unnecessary because no group is indicated, it is syntactically correct.

g3.2/3

The **3.2** group ID is the name of the lockstep group for thread 3.2. This group includes all threads in the process 3 share group that are executing at the same PC as thread 2.

p3/3

Sets the process to process 3. The group of interest (GOI) is set to group 3. If group 3 is a process group, most commands ignore the group setting. If group 3 is a thread group, most commands act on all threads in process 3 that are also in group 3.

When you set the process using an explicit group, you might not be including all the threads you expect to be included. This is because commands must look at the TOI, process of interest (POI), and GOI.

NOTE: It is redundant to specify a thread width with an explicit group ID as this width means that the focus is on one thread.

In the following examples, the first argument to the **dfocus** command defines a temporary P/T set that the CLI command (the last term) operates on. The **dstatus** command lists information about processes and threads. These examples assume that the global focus was **d1.<** initially.

dfocus g dstatus

Displays the status of all threads in the control group.

dfocus gW dstatus

Displays the status of all worker threads in the control group.

dfocus p dstatus

Displays the status of all threads in the current focus process. The width is process, and the (default) group is the control 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.

The following example shows how the prompt changes as you change the focus. In particular, notice how the prompt changes when you use the **C** and the **D** group specifiers.

```
d1.<> f C
dC1.<
dC1.<> f D
d1.<
d1.<>
```

Two of these lines end with the less-than symbol (<). These lines are not prompts. Instead, they are the value returned by TotalView when it executes the **dfocus** command.

Naming Lists with Inconsistent Widths

You can create lists that contain more than one width specifier. This can be very useful, but it can be confusing. Consider the following:

```
{p2 t7 g3.4}
```

This list is quite explicit: all of process 2, thread 7, and all processes in the same group as process 3, thread 4. However, how should TotalView use this set of processes, groups, and threads?

In most cases, TotalView does what you would expect: it iterates over the list and acts on each arena separately. If TotalView cannot interpret an inconsistent focus, it prints an error message.

Some commands work differently. Some use each arena's width to determine the number of threads on which it acts. This is exactly what the **dgo** command does. In contrast, the **dwhere** command creates a call graph for process-level arenas, and the **dstep** command runs all threads in the arena while stepping the thread of interest (TOI). TotalView may wait for threads in multiple processes for group-level arenas. The command description in the *TotalView Reference Guide* points out anything that you need to watch out for.

Merging Focuses

When you specify more than one focus for a command, the CLI merges them. The following example combines **dfocus** and **dstatus** commands. The focus indicated by the prompt—called the *outer* focus—determines the output of **dstatus**.

Note that **f** is a CLI alias for **dfocus**.

```
t1.<> f d
d1.<
d1.<> f tL dstatus
1:      37258   Breakpoint [omp_prog]
  1.1: 37258.1 Breakpoint PC=0x1000acd0,
      [./omp_prog.f#35]
d1.<> f tL f p dstatus
1:      37258   Breakpoint [omp_prog]
  1.1: 37258.1 Breakpoint PC=0x1000acd0,
      [./omp_prog.f#35]
  1.3: 37258.3 Breakpoint PC=0x1000acd0,
      [./omp_prog.f#35]
d1.<> f tL f p f D dstatus
1:      37258   Breakpoint [omp_prog]
  1.1: 37258.1 Breakpoint PC=0x1000acd0,
      [./omp_prog.f#35]
  1.2: 37258.2 Stopped    PC=0xffffffffffffffff
  1.3: 37258.3 Breakpoint PC=0x1000acd0,
      [./omp_prog.f#35]
d1.<> f tL f p f D f L dstatus
1:      37258   Breakpoint [omp_prog]
  1.1: 37258.1 Breakpoint PC=0x1000acd0,
      [./omp_prog.f#35]
  1.3: 37258.3 Breakpoint PC=0x1000acd0,
      [./omp_prog.f#35]
```

Stringing multiple focuses together might not produce the most readable result. In this case, it shows how one **dfocus** command can modify the arena on which another command operates.

Using P/T Set Operators

At times, you do not want all of one type of group or process to be in the focus set. In this case, use the following three operators to manage your P/T sets:

|

Creates a union; that is, all members of two sets.

-

Creates a difference; that is, all members of the first set that are not also members of the second set.

&

Creates an intersection; that is, all members of the first set that are also members of the second set.

For example, the following creates a union of two P/T sets:

```
p3 | L2
```

You can combine these operations, for example:

```
p2 | p3 & L2
```

This statement creates an intersection between p3 and L2, and then creates a union between p2 and the results of the intersection operation. You can directly specify the order by using parentheses; for example:

```
p2 | (p3 & pL2)
```

Typically, these three operators are used with the following P/T set functions:

breakpoint(*ptset*)

Returns a list of all threads that are stopped at a breakpoint.

comm(*process*, "*comm_name*")

Returns a list containing the first thread in each process associated within a communicator within the named process. While *process* is a P/T set it is not expanded into a list of threads.

error(*ptset*)

Returns a list of all threads stopped due to an error.

existent(*ptset*)

Returns a list of all threads.

held(*ptset*)

Returns a list of all threads that are held.

nonexistent(*ptset*)

Returns a list of all processes that have exited or which, while loaded, have not yet been created.

running(*ptset*)

Returns a list of all running threads.

stopped(*ptset*)

Returns a list of all stopped threads.

unheld(*ptset*)

Returns a list of all threads that are not held.

watchpoint(*ptset*)

Returns a list of all threads that are stopped at a watchpoint.

The way in which you specify the P/T set argument is the same as the way that you specify a P/T set for the **dfocus** command. For example, **watchpoint(L)** returns all threads in the current lockstep group that are stopped at a watchpoint. The only operator that differs is **comm**, whose argument is a process.

The dot operator (**.**), which indicates the current set, can be helpful when you are editing an existing set.

The following examples clarify how you use these operators and functions. The P/T set **a** (all) is the argument to these operators.

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

Shows information about all threads that are stopped at breakpoints or watchpoints. The **a** argument is the standard width specifier for **all**.

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

Shows information about all stopped threads that are not stopped at breakpoints.

f {. | breakpoint(a)} dstatus

Shows information about all threads in the current set, as well as all threads stopped at a breakpoint.

f {g.3 - p6} duntil 577

Runs thread 3 along with all other processes in the group to line 577. However, it does not run anything in process 6.

f {(\$PTSET) & p123}

Uses just process 123 in the current P/T set.

Setting and Creating Custom Groups

This section presents a series of examples that set and create groups. To create a custom group, use the command **dgroups**.

You can use the following methods to indicate that thread 3 in process 2 is a worker thread:

```
dset WGROUP(2.3) $WGROUP(2)
```

Assigns the group ID of the thread group of worker threads associated with process 2 to the **WGROUP** variable. (Assigning a nonzero value to **WGROUP** indicates that this is a worker group.)

```
dset WGROUP(2.3) 1
```

This is a simpler way of doing the same thing as the previous example.

```
dfocus 2.3 dworker true
```

Adds the threads in the indicated focus to a workers group.

```
dset CGROUP(2) $CGROUP(1)
```

```
dgroups -add -g $CGROUP(1) 2
```

```
dfocus 1 dgroups -add 2
```

These three commands insert process 2 into the same control group as process 1.

```
dggroups -add -g $WGROU(2) 2.3
```

Adds process 2, thread 3 to the workers group associated with process 2.

```
dfocus tW2.3 dggroups -add
```

This is a simpler way of doing the same thing as the previous example.

Following are some additional examples:

```
dfocus g1 dggroups -add -new thread
```

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

```
set mygroup [dggroups -add -new thread $GROUP($SGROUP(2))]
dggroups -remove -g $mygroup 2.3
dfocus g$mygroup/2 dgo
```

The first command creates a new group that contains all the threads from the process 2 share group; the second removes thread 2.3; and the third runs the remaining threads.

RELATED TOPICS

The **dfocus** command [dfocus](#) in "CLI Commands" in the *TotalView Reference Guide*

The **dgroup** command [group](#) in "CLI Commands" in the *TotalView Reference Guide*

The **dset** command [dset](#) in "CLI Commands" in the *TotalView Reference Guide*

Using the g Specifier: An Extended Example

The meaning of the **g** width specifier is sometimes not clear when it is coupled with a group scope specifier. Why have a **g** specifier when you have four other group specifiers? Stated in another way, isn't something like **gL** redundant?

The simplest answer, and the reason you most often use the **g** specifier, is that it forces the group when the default focus indicates something different from what you want it to be.

The following example shows this. The first step sets a breakpoint in a multi-threaded OMP program and executes the program until it hits the breakpoint.

```
d1.<> dbreak 35
1
d1.<> dcont
Thread 1.1 has appeared
Created process 1/37258, named "omp_prog"
Thread 1.1 has exited
Thread 1.1 has appeared
Thread 1.2 has appeared
Thread 1.3 has appeared
Thread 1.1 hit breakpoint 1 at line 35 in ".breakpoint_here"
```

The default focus is **d1.<**, which means that the CLI is at its default width, the process of interest (POI) is 1, and the thread of interest (TOI) is the lowest numbered nonmanager thread. Because the default width for the **dstatus** command is process, the CLI displays the status of all processes. Typing **dfocus p dstatus** produces the same output.

```
d1.<> dstatus
1:      37258  Breakpoint  [omp_prog]
  1.1: 37258.1 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
  1.2: 37258.2 Stopped    PC=0xfffffffffffffffffff
  1.3: 37258.3 Stopped    PC=0xd042c944
d1.<> dfocus p dstatus
1:      37258  Breakpoint  [omp_prog]
  1.1: 37258.1 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
  1.2: 37258.2 Stopped    PC=0xfffffffffffffffffff
  1.3: 37258.3 Stopped    PC=0xd042c944
```

The CLI displays the following when you ask for the status of the lockstep group. (The rest of this example uses the **f** abbreviation for **dfocus**, and **st** for **dstatus**.)

```
d1.<> f L st
1:      37258  Breakpoint  [omp_prog]
  1.1: 37258.1 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
```

This command displays the status of the threads in thread, which is the 1.1 lockstep group since this thread is the TOI. The **f L focus** command narrows the set so that the display only includes the threads in the process that are at the same PC as the TOI.

By default, the **dstatus** command displays information at process width. This means that you don't need to type **f pL dstatus**.

The **duntil** command runs thread 1.3 to the same line as thread 1.1. The **dstatus** command then displays the status of all the threads in the process:

```
d1.<> f t1.3 duntil 35
35@>      write(*,*)"i= ",i,
          "thread= ",omp_get_thread_num()
d1.<> f p dstatus
1:      37258  Breakpoint  [omp_prog]
  1.1: 37258.1 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
  1.2: 37258.2 Stopped    PC=0xfffffffffffffffffff
  1.3: 37258.3 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
```

As expected, the CLI adds a thread to the lockstep group:

```
d1.<> f L dstatus
1:      37258  Breakpoint  [omp_prog]
  1.1: 37258.1 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
  1.3: 37258.3 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
```

The next set of commands begins by narrowing the width of the default focus to thread width—notice that the prompt changes—and then displays the contents of the lockstep group:

```
d1.<> f t
t1.<> f L dstatus
1:      37258   Breakpoint  [omp_prog]
  1.1: 37258.1 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
```

Although the lockstep group of the TOI has two threads, the current focus has only one thread, and that thread is, of course, part of the lockstep group. Consequently, the lockstep group *in the current focus* is just the one thread, even though this thread's lockstep group has two threads.

If you ask for a wider width (**p** or **g**) with **L**, the CLI displays more threads from the lockstep group of thread 1.1. as follows:

```
t1.<> f pL dstatus
1:      37258   Breakpoint  [omp_prog]
  1.1: 37258.1 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
  1.3: 37258.3 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
t1.<> f gL dstatus
1:      37258   Breakpoint  [omp_prog]
  1.1: 37258.1 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
  1.3: 37258.3 Breakpoint  PC=0x1000acd0,
        [./omp_prog.f#35]
```

If the TOI is 1.1, **L** refers to group number 1.1, which is the lockstep group of thread 1.1.

Because this example only contains one process, the **pL** and **gL** specifiers produce the same result when used with the **dstatus** command. If, however, there were additional processes in the group, you only see them when you use the **gL** specifier.

Changing the P/T Using the dfocus Command

You can change the focus on which a command acts using the **dfocus** command. If the CLI executes the **dfocus** command without a subcommand, it changes the default P/T set. For example, if the default focus is process 1, the following command changes the default focus to process 2:

```
dfocus p2
```

After TotalView executes this command, all commands that follow focus on process 2.

If you begin a command with **dfocus**, TotalView changes the target only for the subcommand that follows. After the subcommand executes, TotalView restores the *former* default. The following example shows both of these ways to use the **dfocus** command. Assume that the current focus is process 1, thread 1. The following commands change the default focus to group 2 and then step the threads in this group twice:

```
dfocus g2
dstep
```

```
dstep
```

In contrast, if the current focus is process 1, thread 1, the following commands step group 2 and then step process 1, thread 1:

```
dfocus g2 dstep  
dstep
```

Some commands operate only at the process level; that is, you cannot apply them to a single thread (or group of threads) in the process, but must apply them to all or to none.

Stepping and Program Execution

Use TotalView execution commands to:

- Execute one source line or machine instruction at a time, including stepping *into* any subroutines; for example, **Process > Step** in the UI and **dstep** in the CLI.

```
CLI: dstep
```

- Execute one source line, *stepping over* subroutine calls; for example, **Process > Next** in the UI and **dnext** in the CLI.

```
CLI: dnext
```

- Run to a selected line, which acts like a temporary breakpoint; for example, **Process > Run To**.

```
CLI: duntil
```

- Run until a function call returns; for example, **Process > Out**.

```
CLI: dout
```

- Advance the program by single assembler instructions.

```
CLI: dstepi
```

- Advance the program by single assembler instructions, *stepping over* subroutine calls.

```
CLI: dnexti
```

In all cases, execution commands operate on the thread of interest (TOI). In the CLI, the TOI is the thread that TotalView uses to determine the scope of the execution operation. In the UI, the TOI is the thread selected in the Processes and Threads view.

In this section:

- [Individual Execution Commands](#)
- [Executing at Group Width](#)
- [Executing at Process Width](#)
- [Executing at Thread Width](#)

- Synchronizing Processes and Threads

RELATED TOPICS

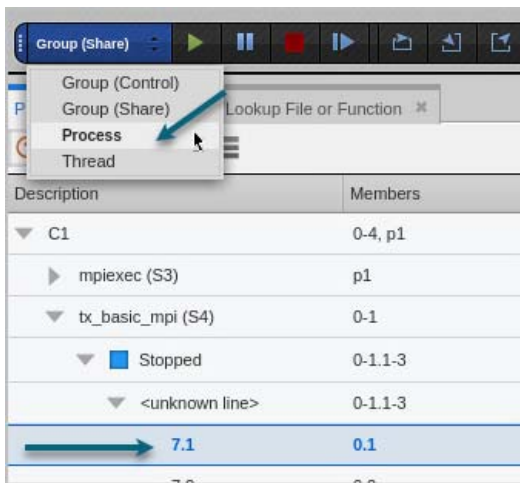
Stepping examples

CLI Stepping Examples

Individual Execution Commands

In the UI, the focus of any execution command is the thread selected in the Processes and Threads view, and the width control selected from the **Focus** menu. In [Figure 122](#), the thread of interest (TOI) is 7.1, and the width is process; TotalView will advance all threads in the process that contains the TOI, or process 7.

Figure 122, The Focus menu in the UI



The term *goal* identifies the location at which a given command will stop executing. For example, for a **Step** command, the goal is the “next line,” even if it’s in a different subroutine, while for a **Run to** command, the goal is the selected line. A **Go** command has no defined goal.

[Table 16](#) defines the default action each execution command takes. Execution commands all default to process width if no other width is provided.

NOTE: [Table 16](#) includes commands available in the UI. Some other execution commands are available only in the CLI, in particular **drun** and **drrun** used after **dload** if you have a starter program. See “CLI Commands” in the *TotalView Reference Guide* for more information.

Table 16: Execution Commands' Default Actions

Command	Default action
Go	Continues all processes and threads in the current focus. See dgo .
Halt	Stops all processes and threads in the current focus. See dhalt .
Kill	Kills all target processes in the current focus. If, however, you kill the primary process for a control group, all processes in the control group are killed. Process-level only. This command cannot be used at thread-width. See dkill .
Restart	Kills all processes in the current control group, then restarts the process that was in the current focus. This command is relevant only to a group. To restart a process, use drerun . See dkill .
Next	Steps one source line, stepping over any subroutine calls. See dnext .
Step	Steps the thread of interest one source line while allowing other threads in the process to run freely. If a statement in a source line call a subroutine, this command steps into the function. See dstep .
Out	Runs a thread until it returns from either of the current subroutine or one or more nested subroutines. When using the default process width, all threads in the process that are not running to this goal run free. See dout .
Run To	Runs the process until a selected location is reached, usually a line in the Source view. If this command is applied to a group, its behavior differs depending on the width. See duntil .

Executing at Group Width

Executing at group width depends on whether the group of interest (GOI) is a process group or a thread group.

- *Process group*—TotalView examines the group and identifies which of its processes has a thread stopped at the same location as the thread of interest (TOI) (i.e. a *matching* process). TotalView runs these matching processes until one of their threads arrives at the goal. At that point, TotalView stops the thread's process. The command finishes when it has stopped all matching processes.

- *Thread group*—TotalView runs all processes in the control group. However, as each thread arrives at the goal, TotalView stops only that thread; the rest of the threads in the same process continue executing. The command finishes when all threads in the GOI arrive at the goal. When the command finishes, TotalView stops all processes in the control group.

TotalView doesn't wait for threads that are not in the same share group as the TOI, since they are executing different code and can never arrive at the goal.

Executing at Process Width

Executing at process width (which is the default) depends on whether the group of interest (GOI) is a process group or a thread group.

- *Process group*—TotalView runs all threads in the process, and execution continues until the thread of interest (TOI) arrives at its goal, which can be the next statement, the next instruction, and so on. Only when the TOI reaches the goal does TotalView stop the other threads in the process.
- *Thread group*—TotalView lets all threads in the GOI run. As each member of the GOI arrives at the goal, TotalView stops it; the rest of the threads continue executing. The command finishes when all members of the GOI arrive at the goal. At that point, TotalView stops the whole process.

Executing at Thread Width

Executing at thread width runs only that thread. No other threads run. In contrast, process width runs all threads in the process that are allowed to run while the thread of interest (TOI) is advanced.

Executing a thread isn't the same as advancing a thread's process, because a process can have more than one thread.

NOTE: Thread-stepping is not implemented on Sun platforms. On SGI platforms, thread-stepping is not available with pthread programs. If, however, your program's parallelism is based on SGI's sprocs, thread-stepping is available.

Thread-level, single-step operations can fail to complete if the TOI needs to synchronize with a thread that isn't running. For example, if the TOI requires a lock that another held thread owns, and steps over a call that tries to acquire the lock, the primary thread can't continue successfully. You must allow the other thread to run in order to release the lock. In this case, you should use process-width stepping instead.

Synchronizing Processes and Threads

When you are running a multi-process or multi-threaded program, it's frequently useful to synchronize execution to the same place. You can do this manually using a **Hold** command, or automatically by setting a barrier point. You can also use the **Run To** command (or **duntil** in the CLI) to run to a specified line in the Source view, which can synchronize processes and threads so you can step them together.

Holding and Releasing Processes and Threads

You can synchronize execution manually using a **hold** command, or automatically by setting a barrier point.

When a process or a thread is held, it ignores any command to resume executing. For example, if you placed a hold on a single process in a control group that contained three processes, selecting **Group > Go** would resume executing just the other two processes in the group. The held process would ignore the Go command.

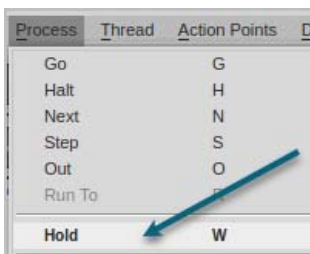
Manually holding and releasing processes and threads is useful when:

- You need to run a subset of the processes and threads. You can manually hold all but those you want to run.
- A process or thread is held at a barrier point and you want to run it without first running all the other processes or threads in the group to that barrier. In this case, you release the process or the thread manually and then run it.

See [Barrier Points](#) for more information on manually holding and releasing barrier breakpoints.

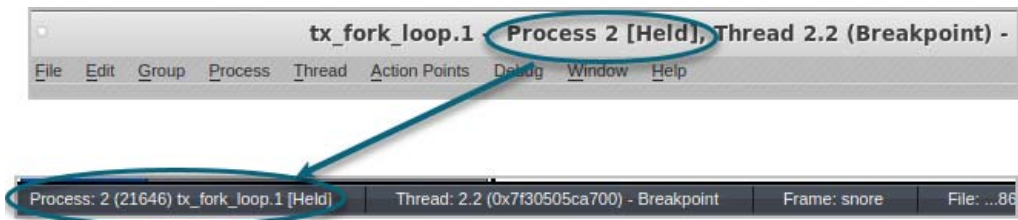
Holding a Process

To hold or release a hold on a process, select the process in the Processes & Threads view, then toggle the menu item **Process > Hold**:



When TotalView is holding a process, the Processes & Threads view displays Stopped, and the title and status bar display “[Held]” for that process.

If a process or a thread is running when you set or release a hold, TotalView stops the process or thread and then holds it. TotalView lets you hold and release processes independently from threads.



If you hold a process, then choose **Process > Go**, TotalView launches a warning popup that you first need to unhold the process before running it:



Holding a Thread

NOTE: The ability to hold a thread will be added to the UI in an upcoming release.

(**CLI only**)). To hold or release a thread, use the CLI:

```
dhold -thread
dunhold -thread
```

Setting the focus changes the scope.

Using Run To and duntil

The **Run To** command in the UI, and the **duntil** command in the CLI, run to a selected line in the Source view code, helping to synchronize processes and threads so you can step them together. Using this command at process width or group width differs.

At process width:

- If the thread of interest (TOI) is already at the goal location, TotalView steps the TOI past the line before the process runs. This lets you use the **Run To** command repeatedly in loops.
- If any other thread in the process is already at the goal, TotalView temporarily holds it while other threads in the process run. After all threads in the process reach the goal, TotalView stops the process. This lets you synchronize the threads in the process of interest (POI) at a source line.

At group width:

- TotalView identifies all processes in the control group that already have a thread stopped at the goal. These are the *matching* processes. TotalView then runs only non-matching processes. Whenever a thread arrives at the goal, TotalView stops its process. The command finishes when it has stopped all processes in the control group. This lets you synchronize at least one thread from all processes in preparation for group-stepping them.

RELATED TOPICS

Setting barrierpoints

[Barrier Points](#)Using **duntil** to run to a selected line[duntil](#)

CLI Stepping Examples

The following are examples that use the CLI stepping commands:

- **Step a single thread**

While the thread runs, no other threads run (except kernel manager threads).

Example: `dfocus t dstep`

- **Step a single thread while the process runs**

A single thread runs into or through a critical region.

Example: `dfocus p dstep`

- **Step one thread in each process in the group**

While one thread in each process in the share group runs to a goal, the rest of the threads run freely.

Example: `dfocus g dstep`

- **Step all worker threads in the process while nonworker threads run**

Worker threads run through a parallel region in lockstep.

Example: `dfocus pW dstep`

- **Step all workers in the share group**

All processes in the share group participate. The nonworker threads run.

Example: `dfocus gW dstep`

- **Step all threads that are at the same PC as the TOI**

TotalView selects threads from one process or the entire share group. This differs from the previous two items in that TotalView uses the set of threads that are in lockstep with the thread of interest (TOI) rather than using the workers group.

Example: `dfocus L dstep`

In the following examples, the default focus is set to **d1.<**

dstep

Steps the TOI while running all other threads in the process.

dfocus W dnext

Runs the TOI and all other worker threads in the process to the next statement. Other threads in the process run freely.

dfocus W duntil 37

Runs all worker threads in the process to line 37.

dfocus L dnext

Runs the TOI and all other stopped threads at the same PC to the next statement. Other threads in the process run freely. Threads that encounter a temporary breakpoint in the course of running to the next statement usually join the lockstep group.

dfocus gW duntil 37

Runs all worker threads in the share group to line 37. Other threads in the control group run freely.

UNW 37

Performs the same action as the previous command: runs all worker threads in the share group to line 37. This example uses the predefined **UNW** alias instead of the individual commands. That is, **UNW** is an alias for **dfocus gW duntil**.

SL

Finds all threads in the share group that are at the same PC as the TOI and steps them all in one statement. This command is the built-in alias for **dfocus gL dstep**.

sl

Finds all threads in the current process that are at the same PC as the TOI, and steps them all in one statement. This command is the built-in alias for **dfocus L dstep**.

RELATED TOPICS

[Execution commands](#)

[Stepping and Program Execution](#)

Execution Commands Using the “all” Arena Specifier

These examples illustrate using commands with the **a** “all” arena specifier.

f aC dgo

Runs everything. If you're using the **dgo** command, everything after the **a** is ignored: **a/aPizza/17.2**, **ac**, **aS**, and **aL** do the same thing. TotalView runs everything.

f aL dstep

Runs only the threads in the thread of interest (TOI)'s lockstep group. All other threads run freely until the stepping process for these lockstep threads completes.

f aC duntil

While everything runs, TotalView must wait until something reaches a goal. It really isn't obvious what this focus is. Since **C** is a process group, you might guess that all processes run until at least one thread in every participating process arrives at a goal. The reality is that since this goal must reside in the current share group, this command completes as soon as all processes in the TOI share group have at least one thread at the goal. Processes in other control groups run freely until this happens.

The TOI determines the goal. If there are other control groups, they do not participate in the goal.

f aS duntil

This command does the same thing as the **f aC duntil** command because the goals for **f aC duntil** and **f aS duntil** are the same, and the processes that are in this scope are identical.

Although more than one share group can exist in a control group, these other share groups do not participate in the goal.

f aL duntil

Although everything will run, it is not clear what should occur. **L** is a thread group, so you might expect that the **duntil** command will wait until all threads in all lockstep groups arrive at the goal. Instead, TotalView defines the set of threads that it allows to run to a goal as just those threads in the TOI's lockstep group. Although there are other lockstep groups, these lockstep groups do not participate in the goal. So, while the TOI's lockstep threads are progressing towards their goal, all threads that were previously stopped run freely.

f aW duntil

Everything runs. TotalView waits until all members of the TOI workers group arrive at the goal.

Scalability in HPC Computing Environments

TotalView provides features and performance enhancements for scalable debugging in today's HPC computing environments, and no special configuration or action is necessary on your part to take advantage of TotalView's scalability abilities.

This chapter details TotalView's features and configurations related to scalability:

- **Scalability Configuration Settings.** Depending on your needs, you might want to set specific configuration variables that enable scalable debugging operations.
- **MRNet Configuration Settings.** TotalView uses MRNet, a tree-based overlay network, for scalable communication. TotalView is preconfigured for scalability, but in some situations you may want to change MRNet's configuration.

Other areas of TotalView that can help support scalability are:

- **The Processes and Threads View.** This view aggregates program state so that it can display quickly and is easy to understand.
- **`dstatus` and `dwhere` command options.** These options provide aggregated views of various process and thread properties. (See the **`-group_by`** option in the *TotalView Reference Guide* entries for these commands.)
- **Compressed process/thread list.** The **`ptlist`** compactly displays the set of processes and threads that have been aggregated together.

RELATED TOPICS

Compressed List Syntax (**`ptlist`**)

"Compressed List Syntax (`ptlist`)" in the **`dstatus`** entry of the *TotalView Reference Guide*

`dstatus -group_by` and **`dwhere-group_by`** options

`dstatus` and **`dwhere`** in the *TotalView Reference Guide*

Configuring TotalView for Scalability

You can configure TotalView in various ways to take advantage of features that improve startup performance and scalability for large-scale parallel jobs.

Disable User-Thread Debugging

Disabling user-thread debugging can help improve debugger performance. User thread debugging is an area of the debugger that has not yet been parallelized, and can therefore slow down job launch and attach time. However, disabling user thread debugging also disables support for displaying thread local storage (e.g., via the `__thread` compiler keyword).

To configure TotalView with these settings, create a TotalView startup file in `<totalviewInstallDir>/<PLATFORM>/lib/.tvdrc` and add the following lines:

```
# If TLS is not required, disable user threads for faster launch
# and attach times
dset -set_as_default TV::user_threads false
```

Tune Dynamic Library Load Processing

When a target process calls `dlopen()`, a **dlopen** event is generated and must be handled by TotalView. Because **dlopen** event handling can affect debugger performance for a variety of reasons, especially if the application loads many shared libraries or the debugger is controlling many processes, TotalView provides ways to configure **dlopen** for better performance and scalability in HPC computing environments:

- **Filtering dlopen events** to avoid stopping a process for each event.
- **Handling dlopen events in parallel**, reducing client/server communication overhead to fetch library information.

Note that both this option and MRNet must be enabled for TotalView to fetch libraries in parallel.

Filtering dlopen Events

Typically, TotalView processes all dynamic shared libraries as they are loaded, looking for locations where breakpoints need to be planted and processing debug symbols in order to give developers the most information about their application. This processing takes time and can greatly slow down the startup of the debugger. This extra work may be unnecessary if no breakpoints need to be planted in the shared libraries.

Filtering **dlopen** events may be particularly beneficial when using Open MPI or other highly dynamic runtime libraries.

TotalView provides two **dlopen** state variables and related command line options to defer planting breakpoints in dynamically loaded libraries until the process stops for some other reason. Deferring **dlopen** event processing allows the debugger to handle all dynamically loaded shared libraries at the same time, which is much more efficient than handling them serially.

Use these state variables to configure a dynamic library processing mode of either Fast, Medium, or Slow.

- In **Fast mode**, the process never stops for a **dlopen** event, not even for "null" **dlopen** events. Using this option can result in significant performance gains, but may be impractical for some applications because any breakpoints in a dynamic library won't be enabled until later in program execution.
- In **Medium mode**, some libraries can be specified to either immediately reevaluate or defer evaluation of breakpoint specifications, rather than all or none. This allows for a more fine grained approach in processing library symbols and setting breakpoints in order to stop early in program execution.
- In **Slow mode**, every **dlopen** event results in the immediate reevaluation of breakpoint specifications.

For detail, see "[Filtering dlopen Events](#)" in the *TotalView Reference Guide*.

Handling dlopen Events in Parallel

TotalView's default behavior is to handle *dlopened* libraries serially, creating multiple, single-cast client-server communications. This can degrade performance, depending on the number of libraries a process *dlopes*, and the number of processes in the job.

Instead, you can configure the debugger to support handling events in parallel using the state variable and command line option **TV::dlopen_read_libraries_in_parallel** and **-dlopen_read_libraries_in_parallel**. MRNet must also be enabled for this to work.

For detail, see [Handling dlopen Events in Parallel](#) in the *TotalView Reference Guide*.

MRNet

MRNet stands for “Multicast/Reduction Network.” MRNet uses a tree-based front-end to back-end communication model to significantly improve the efficiency of data multicast and aggregation for front-end tools running on massively parallel systems.

The following description is from the MRNet web site (<http://www.paradyn.org/mrnet/>):

MRNet is a software overlay network that provides efficient multicast and reduction communications for parallel and distributed tools and systems. MRNet uses a tree of processes between the tool's front-end and back-ends to improve group communication performance. These internal processes are also used to distribute many important tool activities, reducing data analysis time and keeping tool front-end loads manageable.

MRNet-based tool components communicate across logical channels called streams. At MRNet internal processes, filters are bound to these streams to synchronize and aggregate dataflows. Using filters, MRNet can efficiently compute averages, sums, and other more complex aggregations and analyses on tool data. MRNet also supports facilities that allow tool developers to dynamically load new tool-specific filters into the system.

Rogue Wave’s use of MRNet is part of a larger strategy to improve the scalability of TotalView as high-end computers grow into very high process and thread counts.

TotalView supports MRNet on Linux x86_64, Linux ARM64, Linux PowerLE clusters, and Cray XT/XE/XK/XC.

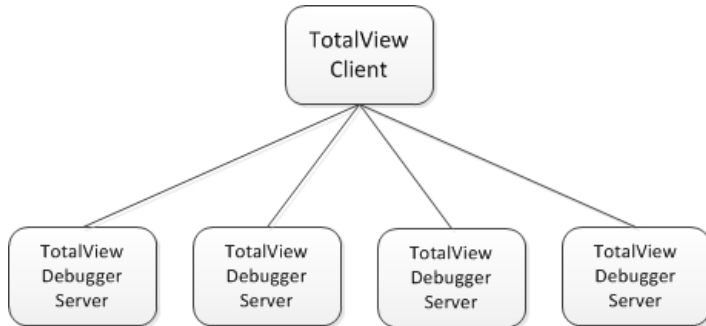
TotalView Infrastructure Models

Starting with Classic TotalView 8.11.0, the TotalView debugger supported two infrastructure models that control the way the debugger organizes its TotalView debugger server processes when debugging a parallel job involving multiple compute nodes. As of Classic TotalView 8.15, TotalView uses the tree-based infrastructure described below by default.

The first model uses a “flat vector” of TotalView debugger server processes. The TotalView debugger has always supported this model, and still does. Under the flat vector model, the debugger server processes have a direct (usually socket) connection to the TotalView front-end client. This model works well at low process scales, but begins to degrade as the target application scales beyond a few thousand nodes or processes. This is the default infrastructure model.

Figure 123 shows the TotalView client connected to four TotalView debugger servers (`tvdsvr`). In this example, four separate socket channels directly connect the client to the debugger servers.

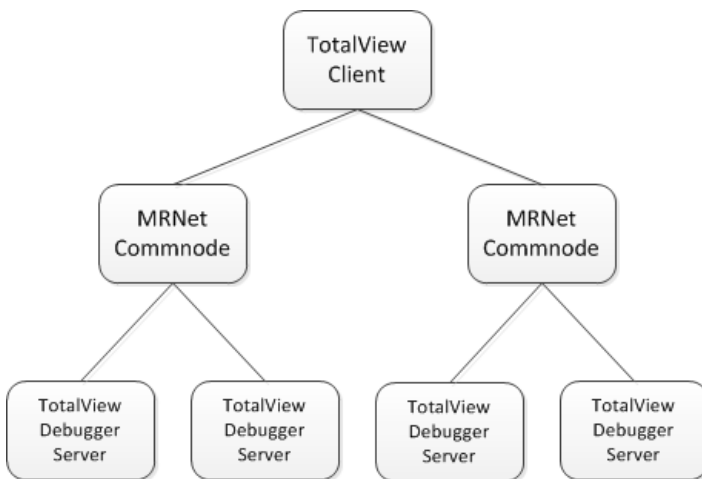
Figure 123, Flat Vector of Servers Infrastructure Model



The second model uses MRNet to form a tree of debugger server and MRNet communication processes connected to the TotalView front-end client, which forms the root of the tree. MRNet supports building many different shapes of trees, but note that the shape of the tree (for example, depth and fan-out) can greatly affect the performance of the debugger. The following sections describe how to control the shape of the MRNet tree in TotalView.

Figure 124 shows an MRNet tree in which the TotalView client is connected to four TotalView debugger servers through two MRNet commnode processes using a tree fan-out value of 2.

Figure 124, MRNet Infrastructure Model



Using MRNet with TotalView

TotalView is already preconfigured for maximum scalability, so no further customization is necessary. This section is for advanced users and describes TotalView options and state variables related to the use of MRNet with TotalView, as follows:

- [General Use](#)
- [Using MRNet on Cray Computers](#)

Please refer to the TotalView documentation for a general description of how options and state variables can be used with TotalView.

General Use

This section discusses basic configuration options of MRNet with TotalView. If you are working on a Cray computer, look at the section specific to that system as well.

Disabling MRNet Before Startup

By default, TotalView uses the MRNet infrastructure on the platforms where it is supported (see the *TotalView Platforms Guide* for specifics). On platforms where MRNet is not supported, TotalView uses its standard vector-of-servers infrastructure.

If for some reason you do not want to use the MRNet infrastructure to debug an MPI job, you must first disable MRNet in TotalView before launching the MPI job. MRNet can be disabled by:

- Starting TotalView with the **-nomrnet** option:

```
prompt> totalview -nomrnet
```

- With TotalView running, use the command line interface (CLI) to set the **TV::mrnet_enabled** state variable:

```
prompt> dset TV::mrnet_enabled false
```

MRNet Server Launch String

Option: `-mrnet_server_launch_string string`

State variable: `TV::mrnet_server_launch_string string`

Default string: `%B/tvdsvr%K -working_directory %D -set_pw %P -verbosity %V %F`

The server launch string defines configuration options when launching a debugging server. TotalView has a default string it uses when launching a server using the vector-of-servers architecture, and an option and state variable that allow you to modify the default string. The MRNet usage of TotalView also has a default launch string and corresponding option and state variables.

The MRNet launch string differs from the standard launch string in two ways: it does not contain a remote shell command expansion (e.g., `rsh` or `ssh`), and it has no `-callback` option.

TotalView always appends the following string to the expanded MRNet launch string:

```
-mrnet_launch node_id
```

where `node_id` is an integer that specifies the server's TotalView node ID within the job. If `node_id` is 0, the server assigns itself a node ID equal to its MRNet rank plus 1.

Controlling the Shape of an MRNet Tree

The shape of the MRNet tree calculated by TotalView can be controlled through a collection of options and state variables. Given the list of hosts, which is typically extracted from the MPIR proctable, TotalView calculates an MRNet topology string to create various shapes of trees.

These are the basic controls:

- **Tree fan-out:** specifies the maximum number of children a node can have. If the number of leaves in the tree is not a power of the fan-out, some of the tree nodes will have fewer children.
- **Tree depth:** specifies the maximum depth of the tree (that is, the number of levels below the root). If the number of leaves is not greater than the square of the tree depth value, a shallower tree is built.
- **Extra root node:** Whether to allocate an extra communications node below the root.
- **Create a “super bushy” tree:** Create one debugger server process per MPI process rather than the default of creating one debugger server process per node, to overcome a CUDA limitation.

MRNet Tree Fan-Out

Option: `-mrnet_fanout integer`

State variable: `TV::mrnet_fanout integer`

Default value: 32

If you change the default value, the new value must be greater than or equal to **2** and less than or equal to **32768**.

MRNet Tree Depth

Option: `-mrnet_levels` *integer*

State variable: `TV::mrnet_levels` *integer*

Default value: `2`

The MRNet tree depth can be specified in terms of the number of levels below the root. If you change the default value, the new value must be greater than or equal to `-2` and less than or equal to `32`.

- If the tree depth is `0`, the MRNet tree fan-out value is used, and TotalView attempts to honor the fan-out value near the bottom of the tree (the leaves).
- If the tree depth is set to a value that is **greater than 0** (which includes the default value of `2`), the fan-out value is ignored and a balanced tree is built with at most the specified number of levels.
- If the tree depth is `-1`, the fan-out value is used, and TotalView attempts to honor the fan-out value near the top (the root) of the tree, rather than near the bottom of the tree (the leaves).
- If the tree depth is `-2`, TotalView builds a tree similar to the one created when the tree depth is `-1`, except that the tree is unbalanced from side-to-side.

As an example, consider a tree with a root node and eight leaf nodes. If the fan-out value is `4` and the tree depth value is `0`, a tree that is “bushy” near the leaves is built because TotalView honors fan-out at the leaf end of the tree.

```

root:1 => n1:2 n5:2 ;
n1:2 => n1:1 n2:1 n3:1 n4:1 ;
n5:2 => n5:1 n6:1 n7:1 n8:1 ;

```

However, for the same tree when the tree depth setting is `-1`, a tree that is “bushy” near the root is built because TotalView honors fan-out at the root end of the tree.

```

root:1 => n1:2 n3:2 n5:2 n7:2 ;
n1:2 => n1:1 n2:1 ;
n3:2 => n3:1 n4:1 ;
n5:2 => n5:1 n6:1 ;
n7:2 => n7:1 n8:1 ;

```

Allocate an Extra Root Node

Option: `-mrnet_extra_root` *boolean*

State variable: `TV::mrnet_extra_root` *boolean*

Default value: `false`

For example, for a tree with a root and eight leaf nodes, using a fan-out value of `4`, a tree depth value of `0`, and requesting an extra root node, the following topology string will be calculated:

```

root:3 => root:1 ;
root:1 => n1:2 n5:2 ;

```



```
n1:2 => n1:1 n2:1 n3:1 n4:1 ;
n5:2 => n5:1 n6:1 n7:1 n8:1 ;
```

Create a “Super Bushy” Tree

Option: `-mrnet_super_bushy`

State variable: `TV::mrnet_super_bushy`

Default value: `false`

Set this option to **true** if you are debugging an MPI job in which more than one CUDA process is running on a node. This option addresses the CUDA debug API limitation that allows a debugger process (such as the `tvdsvr`) to debug at most one target process using a GPU.

Path to MRNet Components

Option: `-mrnet_commnode_path path-to-mrnet_commnode`

State variable: `TV::mrnet_commnode_path path-to-mrnet_commnode`

Default value: `tv-installation-root/platform/bin/mrnet_commnode`

In a TotalView distribution, this is a path to a shell script that sets environment variables and execs the proper executable for the platform.

Path to the MRNet shared library directory

Option: `-mrnet_filterlib_dir path-to-mrnet-shlib-directory`

State variable: `TV::mrnet_filterlib_dir path-to-mrnet-shlib-directory`

Default value: `tv-installation-root/platform/shlib/mrnet/obj`

The TotalView server tree filters library `libservertree_filters.so.1` and the MRNet `libxplat.so` and `libmrnet.so` libraries are stored in this directory.

Performance Notes

Perforce has conducted performance tests on some specific systems, and based on this testing, we’re providing a few tips. These tips should be considered as guidelines. The only way to know how performance is affected by different tree configurations on your system is by trying out alternatives with your own jobs.

- In general, higher fan-outs seem to perform better than deeper trees. Specifically, trees deeper than two levels consistently performed worse than a two-level tree.
- In our testing, a one-level tree failed due to resource shortages at around 512 nodes, so this is not a viable option at higher scales.

MRNet and ssh/rsh

Controlling MRNet's Use of rsh vs ssh

When MRNet is used as the infrastructure in a Linux cluster front-end node, MRNet's built-in support is used to instantiate the tree of debugger servers and communications processes. Tree instantiation is based on a remote shell startup mechanism. By default, MRNet uses `ssh` as the remote shell program, but some environments require that `rsh` be used instead. TotalView controls the remote shell used by MRNet using the `TV::xplat_rsh` state variable or the `-xplat_rsh` TotalView command option to set this state variable. If this variable isn't explicitly set and the `XPLAT_RSH` environment variable is not set or is empty, TotalView uses the value of `TV::launch_command` when instantiating an MRNet tree.

On Cray XT, XE, and XK systems, MRNet uses the ALPS Tool Helper library to instantiate the tree, which does not require the use of a separate remote shell program.

Tips on Using ssh/rsh with MRNet

The use of `rsh` / `ssh` differs in every system environment, therefore you should consult your system's documentation to know whether `rsh` or `ssh` should be used for your system. The `rsh` and `ssh` man pages are also a useful resource. Regardless, we offer the following tips as a guideline for how to configure `rsh` and `ssh`:

- Configure `rsh` or `ssh` to allow accessing the remote nodes without a password. `rsh` typically uses a file named `$/HOME/.rhosts` (see `man 5 rhost` on a Linux system). `ssh` typically uses a pair of private/public keys stored in files under your `$/HOME/.ssh` directory (see `man 1 ssh` on a Linux system).
- Disable X11 forwarding in `ssh` in your `$/HOME/.ssh/config` file (see `man 5 ssh_config` on a Linux system).
- Set `StrictHostKeyChecking` to `no` in `ssh` in your `$/HOME/.ssh/config` file (see `man 5 ssh_config` on a Linux system). If the `ssh` host keys change for a remote host, you may need to delete the lines for the host from the `$/HOME/.ssh/known_hosts` file, or remove the file.

Using MRNet on Cray Computers

The following sections describe the options and state variables that control the configuration and use of MRNet on Cray. Please refer to the *TotalView Reference Guide* for a general description of how `options` and `state variables` can be used with TotalView.

For more information on Cray, see Debugging Cray XT/XE/XK/XC Applications on page 540.

Is Cray XT Flag

State variable: `TV::is_cray_xt` *boolean*

Default value: Set to `true` if TotalView is running on Linux-x86_64 or Linux-ARM64 (aarch64) and `/proc/cray_xt/nid` exists; otherwise, set to `false`.

Note that some Cray front-end (eloin) nodes do not have a `/proc/cray_xt/nid` file, in which case a job must be submitted to start TotalView on a Cray XT/XE/XK/XC node, or `tvconnect` must be used in your batch job. (For detail on `tvconnect`, see Reverse Connections on page 491.)

Is Cray CTI Flag

State variable: `TV::is_cray_cti` *boolean*

Default value: Set to `true` if TotalView is running on Linux-x86_64 or Linux-ARM64 (aarch64) and `/opt/cray/pe/cti/` exists; otherwise, set to `false`. TotalView uses the CTI (Cray Tools Interface) library to deploy debugger processes on the node where your application is running.

Cray XT MRNet Server Launch String

Option: `-cray_xt_mrnet_server_launch_string` *string*

State variable: `TV::cray_xt_mrnet_server_launch_string` *string*

Default value: `/var/spool/alps/%A/toolhelper%A/tvdsvr%K \`
`-working_directory %D -set_pw %P -verbosity %V %F`

Analogous to the standard MRNet server launch string, the Cray XT MRNet server launch string is used when MRNet launches the TotalView debugger servers on Cray when using the ATH (ALPS Tool Helper) library. TotalView expands the launch string using the normal launch string expansion rules.

Cray XT MRNet Transfer File List

Option: `-cray_xt_mrnet_xfer_file_list` *stringlist*

State variable: `TV::cray_xt_mrnet_xfer_file_list` *stringlist*

Default value:

The default value is calculated at TotalView startup time, as follows. The following is used as a "base" list of files needed by TotalView on the Cray compute nodes when MRNet and the Cray ATH libraries are in use.

```
TVROOT/bin/mrnet_commnode_main_cray_xt
TVROOT/bin/tvdsvr_mrnet
TVROOT/bin/tvdsvrmain_mrnet
TVROOT/shlib/mpa/obj_cray_xt/libmpattr.so.1
TVROOT/shlib/unwind/obj/libunwind-*.so.8
TVROOT/shlib/mrnet/obj_cray_xt/libmrnet.so
TVROOT/shlib/mrnet/obj_cray_xt/libxplat.so
TVROOT/shlib/mrnet/obj_cray_xt/libservertree_filters.so.1
TVROOT/shlib/mrnet/obj_cray_xt/libtvwrapalps.so.1
/lib64/libthread_db.so.1
```

Note that the name of the "libunwind-*.so.8" library depends on the platform, and will be either "libunwind-x86_64.so.8" for x86_64 or "libunwind-aarch64.so.8" for ARM64.

On the x86_64 platform, TotalView also stages the libraries required to support ReplayEngine, which include:

```
/usr/bin/ld
/usr/bin/objcopy
TVROOT/lib/libundodb_debugger_x64.so
TVROOT/lib/undodb_a_x64.o
TVROOT/lib/undodb_infiniband_preload_x64.so
TVROOT/lib/undodb_a_x32.o
TVROOT/lib/undodb_infiniband_preload_x32.so
```

The above list is then passed to the shell script named "cray_sysdso_deps.sh" to calculate the system shared libraries needed by the executables and shared libraries on the base list. The actual list of system libraries can vary from system to system, but typically consists of the following files:

```
/lib64/libgcc_s.so.1
/usr/lib64/libbfd-<version>.so
/usr/lib64/libstdc++.so.6
```

The version of `libbfd`, which is needed by `ld` and `objcopy`, varies from system to system.

The default value is a space-separated string-list of file names that are transferred (staged) to the compute nodes. These files are the shell script, executable and shared library files required to run the MRNet commnode and TotalView debugger server processes on the compute nodes. When instantiating the MRNet tree on Cray, the ALPS Tool Helper library is used to broadcast these files into the compute nodes' ramdisk under the `/var/spool/alps/apid` directory. `TVROOT` is the path to the platform-specific files in the TotalView installation.

Note that most up-to-date Cray systems support the debugger using the Cray Tools Interface (CTI) library, however TotalView attempts to support older legacy Cray systems that do not have CTI available by using the ALPS Tool Helper (ATH) library.

Cray CTI MRNet Transfer File List

Option: `-cray_cti_mrnet_xfer_file_list` *stringlist*

State variable: `TV::cray_cti_mrnet_xfer_file_list` *stringlist*

Default value:

The default value is calculated at TotalView startup time, as follows. The following is used the "base" list of files needed by TotalView on the Cray compute nodes when MRNet and the Cray CTI libraries are in use.

```
TVROOT/bin/mrnet_commnod_main_cray_cti
TVROOT/bin/tvdsvrmain_mrnet
TVROOT/shlib/mpa/obj_cray_xt/libmpattr.so.1
TVROOT/shlib/unwind/obj/libunwind-*.so.8
TVROOT/shlib/mrnet/obj_cray_cti/libmrnet.so
TVROOT/shlib/mrnet/obj_cray_cti/libxplat.so
TVROOT/shlib/mrnet/obj_cray_cti/libservertree_filters.so.1
TVROOT/shlib/mrnet/obj_cray_cti/libtvwrapcti.so.1
```

```
/lib64/libthread_db.so.1
```

Note that the name of the "libunwind-*.so.8" library depends on the platform, and will be either "libunwind-x86_64.so.8" for x86_64 or "libunwind-aarch64.so.8" for ARM64.

On the x86_64 platform, TotalView also stages the libraries required to support ReplayEngine, which include:

```
/usr/bin/ld  
/usr/bin/objcopy  
TVROOT/lib/libundodb_debugger_x64.so  
TVROOT/lib/undodb_a_x64.o  
TVROOT/lib/undodb_infiniband_preload_x64.so
```

Note that CTI does not support staging 32-bit ELF files, therefore they are not included in the above list. Shared library dependencies are calculated by CTI itself, therefore CTI takes care of staging any additional required shared library dependencies.

PART IV Accessing TotalView Remotely

Remotely debug your programs:

- **TotalView Remote Connections**

Use your TotalView UI to directly connect to another server running TotalView. This option allows you to seamlessly run remote debugging sessions without requiring an external client, and provides you access to all of TotalView's debugging tools as if you were running TotalView on a local machine.

- **TotalView Remote Display**

Use the TotalView Remote Display client to run TotalView on another system.

TotalView Remote Connections

- [About Remote Connections](#)
- [Configuring a Remote Connection](#)
- [Debugging on a Remote Connection](#)

From the TotalView UI running on your machine, set up a remote session on another server running TotalView. This feature allows you to debug programs using the TotalView UI on your local machine, with access to all the tools available within a regular TotalView debugging session on a remote machine.

This feature works in two ways:

- **For TotalView's regular supported platforms**, this feature does *not* require the installation and use of a separate client; rather, it is a built-in feature of the regular TotalView.
- **For platforms on which the TotalView debugger is not supported** — for instance, Windows — a TotalView Remote Client installer is provided.

About Remote Connections

The ability to set up a remote connection is available in the TotalView UI on all supported platforms.

The TotalView Remote Client makes it possible to run TotalView even on platforms that do not support the TotalView debugger. For example, Windows users can install the client and then remotely connect to the TotalView debugger on a supported platform.

Setting up a remote connection requires that either the TotalView Remote Client or TotalView be running on the local machine, that TotalView be installed on the remote machine, and that you enter any required connection and authorization information.

Before establishing a remote connection:

- For platforms that do not support the TotalView debugger, download and install the TotalView Remote Client .
Note: If you are using the TotalView debugger on your local machine and want to run a remote session on another machine, there is no need to install an external client, as this functionality is built into TotalView.
- Ensure that your environment for running TotalView is set up properly on the remote server.
- Ensure that the version of TotalView running on both the remote and local server is the same.

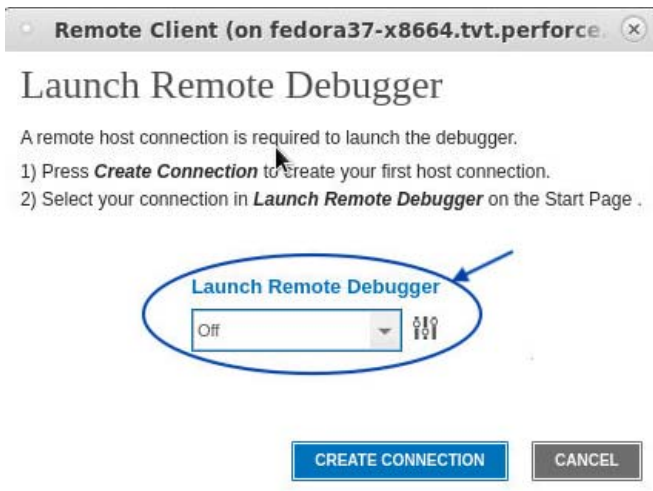
NOTE: The version of TotalView running on the local and remote server must be the same.

Once the remote connection is established, you can debug programs on the remote server in the same way you work on your local machine, with all the tools available to you.

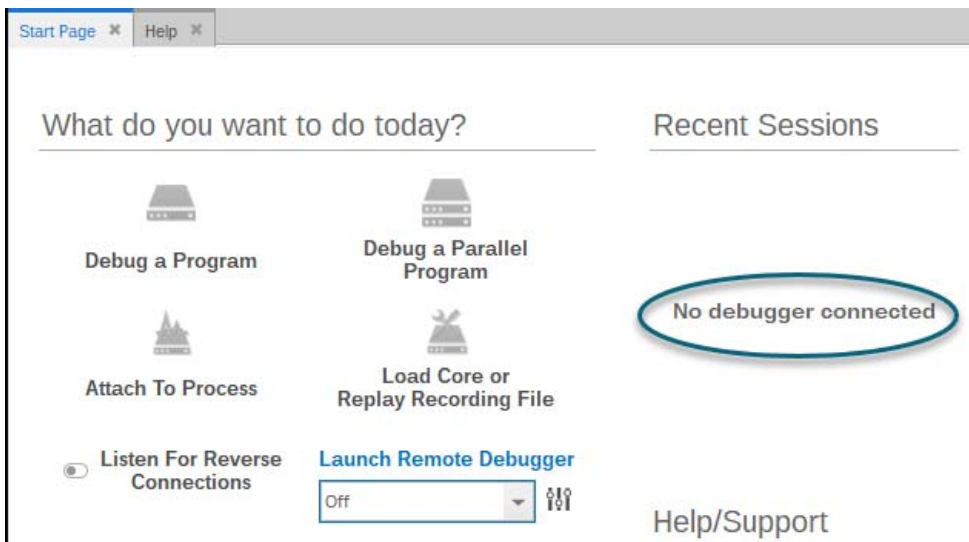
Connecting Remotely From the TotalView Remote Client


Download the TotalView Remote Client from TotalView's download page at <https://totalview.io/downloads>, and follow the prompts to install it.

Start the TotalView Remote Client. For systems that have only the TotalView Remote Client and do not have the TotalView debugger itself installed, and on those for which no remote connections have yet been created, a popup opens prompting you to create a connection.



Note that, because the TotalView debugger is not installed, the Start Page displays a message “No debugger connected.”



To start a TotalView debugging session, select the Configure () icon under **Launch Remote Debugger**.

The Remote Connections preferences page opens.

NOTE: Preferences other than Remote Connections are disabled until you have configured and launched a connection to the TotalView debugger.

Because no connection yet exists, the dropdown “Select a configuration to edit” is set to “Off.” You must create a new remote connection before proceeding. Select **create a new configuration**.

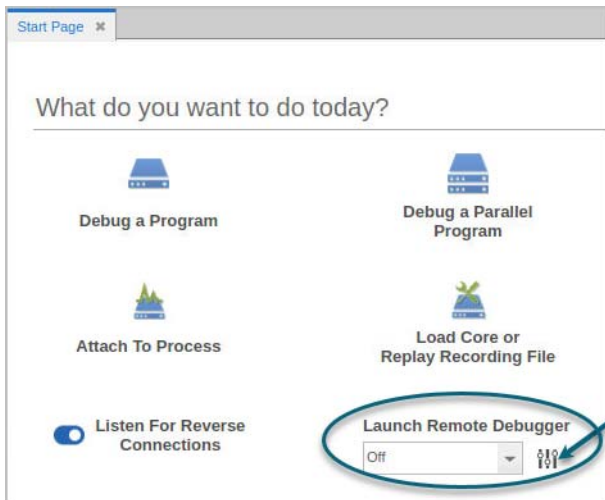
Provide the necessary information to reach the host and the debugger.




See [Configuring a Remote Connection](#).

Connecting Remotely From a TotalView Debugger Installation

When the TotalView debugger is installed, the remote connection feature is available from the Start Page before you open a debug session.



It is turned off by default. To set up or open a remote connection, select the Configure () icon under **Launch Remote Debugger**, to launch the Remote Connections preferences page. See [Configuring a Remote Connection](#).

Configuring a Remote Connection

Configure remote connections on the Remote Connections preferences page, launched when you select the Configure icon on the Start Page.

Figure 125, Remote Connection Preferences page

The screenshot shows a window titled "Preferences (on fedora37-x8664.tvt.perforce.com)" with a sidebar on the left containing icons for DISPLAY, ACTION POINTS, SEARCH PATH, PARALLEL, REMOTE CONNECTIONS (highlighted), TOOL BAR, and LABS. The main content area is titled "Remote Connections" and contains the following fields:

- Connection Name:** A text input field with "New Connection" and a "REQUIRED" label.
- Remote Host(s):** A text input field with "user@host1, user@host2" and a "REQUIRED" label.
- Private Key File:** A text input field with "/my/path/my_key.pem" and a "BROWSE..." button.
- TotalView Remote Installation Directory:** A text input field with "/opt/toolworks/totalview.2020.3.0/bin".
- Remote Command(s):** A text input field with "module load totalview Or source /path/to/remote/script".
- Remote TotalView Arguments:** A text input field with "remote totalview arguments".

At the bottom right, there are three buttons: "OK", "APPLY", and "CANCEL".

Existing configurations display in the “Select a configuration to edit” dropdown. To create a new connection, select **create a new configuration**. Enter the required information:

- **Connection Name**

A descriptive name for this connection.

- **Remote Host(s)**

The user and remote host to connect to, either a network path or an IP address. If your network has a gateway machine, use a comma-delineated string to include intermediate hosts. For example:

```
user@myServer.myNetwork.com, user@anotherServer.myNetwork.com
```

You can enter multiple hosts, but only the first two support entering a password; you'll need to access a third or subsequent host without a password.

NOTE: If you're using some customized settings, including an alias for your host name or a non-standard port for your SSH daemon, add this information to your local SSH configuration file.

- **Private Key File**

Provide an optional Private Key File used as part of the SSH connection to connect to your remote system.

- **TotalView Remote Installation Directory**

The directory on the remote host where TotalView is installed. Identify the installation's top-level directory, for example, `/opt/totalview.2024`.

- **Remote Command(s)**

Optional shell commands to execute before TotalView is run on the remote server. Shell commands must follow the syntax of your SSH login shell. Some examples:

- **Add TotalView to your path.** This avoids having to specify the TotalView remote installation directory:

```
csh: setenv PATH /opt/toolworks/totalview.2024.1/bin:$PATH
```

```
bash: export PATH=/opt/toolworks/totalview.2024.1/bin:$PATH
```

- **Load the TotalView module and openmpi module** if they are configured for your system. This also avoids having to specify the TotalView remote installation directory:

```
csh or bash: module load totalview ; module load openmpi
```

- **Source a script to set your environment:**

```
csh or bash: source /path/to/my/script
```

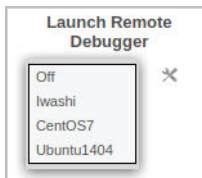
■ Remote TotalView Arguments

Enter program arguments. For instance, you might enter TotalView arguments, the program to debug, and arguments to your target program. For example:

```
/bin/hello -no_user_threads -a "Say Hi"
```

Where `-no_user_threads` is an argument for TotalView, and "Say Hi" is an argument for the target program `hello`.

Once configured, click **OK**. You are returned to the Start Page where your figured connections will now be available in the Remote Connections dropdown:



To Modify or Delete a Connection

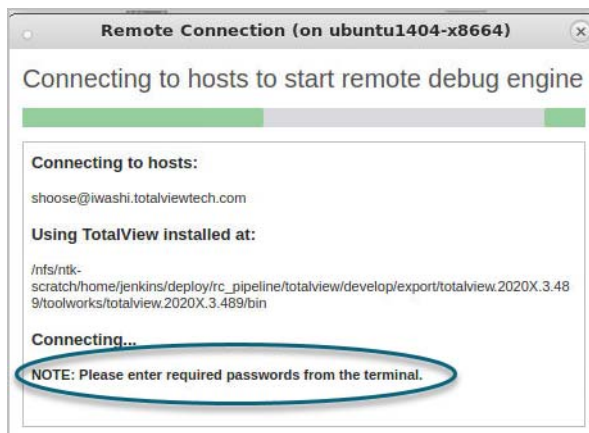
To edit a configuration, select it in the "Select a configuration to edit" dropdown box, make your changes, then select **OK**.

To delete a configuration, select **Delete Configuration**.

Debugging on a Remote Connection

NOTE: If you need to enter a password to connect to a remote server, you must start TotalView from a shell.

From the Start Page, select your remote connection from the Remote Connections dropdown. A popup launches:

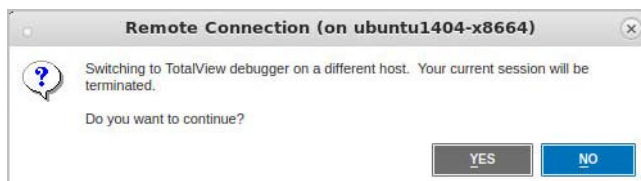


NOTE: The version of TotalView on both the local and remote servers must be the same. If not, TotalView will fail to connect.

If the remote host requires a password, the dialog prompts you to enter it in the shell from which you launched TotalView.

Running a remote connection will *end* your existing session; any local variables or open files will close.

If you are in the middle of an active session with files open, TotalView prompts you to ensure your choice.



NOTE: Opening a remote connection will close your existing session.

The remote session opens. Choose an existing session or create a new session. To create a new session, you will need to manually enter the filename and path under the File Name field.

NOTE: Manually enter the filename and path to the program you plan to debug. Browsing on the remote server via the UI is not yet supported.

Debug a program in the same way you would if you were running TotalView on your local machine. All debugging tools are available to you.

Identifying a Remote Debugging Session

When on a remote connection, two visual cues identify that this is a remote session:

- The title bar displays the server's name.
- The status bar displays the descriptive name for your remote connection configuration.

For example, the debugging session below was initiated on ubuntu1404-x8664, and is running TotalView on the server iwashi.totalviewtech.com, based on a connection named "Iwashi:"



TotalView Remote Display

Using the TotalView Remote Display client, you can start and then view TotalView as it executes on another system, so that TotalView need not be installed on your local machine.

- [Remote Display Supported Platforms](#)
- [Remote Display Components](#)
- [Installing the Client](#)
- [Client Session Basics](#)
- [Advanced Options](#)
- [Naming Intermediate Hosts](#)
- [Submitting a Job to a Batch Queuing System](#)
- [Setting Up Your Systems and Security](#)
- [Session Profile Management](#)
- [Batch Scripts](#)

Remote Display Supported Platforms

Remote Display is currently bundled into all TotalView releases.

Supported platforms include:

- Linux x86-64
- Microsoft Windows
- Apple macOS Intel

No license is needed to run the Client, but TotalView running on any supported operating system must be a licensed version of TotalView 8.6 or greater.

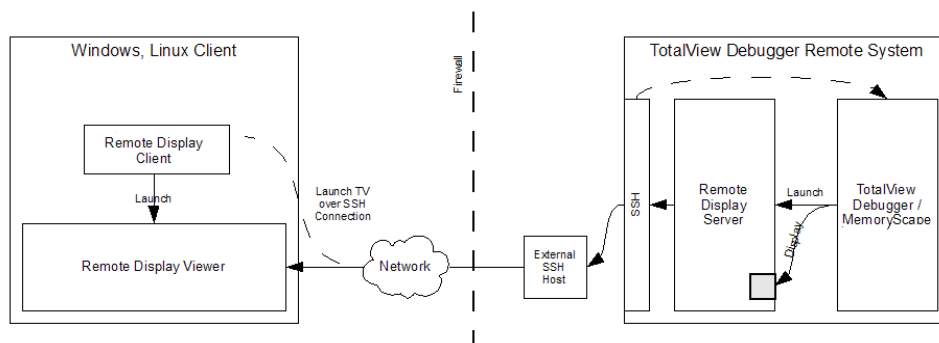
Remote Display Components

TotalView Remote Display has three components:

- The Client is a window running on a Remote Display supported platform (See [Remote Display Supported Platforms](#)).
- The Server is invisible, managing the movement of information between the Viewer, the remote host, and the Client. The Server can run on all systems that TotalView supports. For example, you can run the Client on a Windows system and set up a Viewer environment on an IBM RS/6000 machine.
- The Viewer is a window that appears on the Client system. All interactions between this window and the system running TotalView are handled by the Server.

Figure 126 shows how these components interact.

Figure 126, Remote Display Components



In this figure, the two large boxes represent the computer upon which you execute the Client and the remote system upon which TotalView runs. Notice where the Client, Viewer, and Server are located. The small box labeled External SSH Host is the gateway machine inside your network. The Client may be either inside or outside your firewall. This figure also shows that the Server is created by TotalView or MemoryScope as it is contained within these programs and is created after the Client sends a message to TotalView or MemoryScope.

TotalView and the X Window system must be installed on the remote server machine containing the `rgb` and `font` files in order for the remote display server to start correctly. The bastion nodes (if any) between the remote client machine and remote server machine do not require TotalView or X Window file access.

Installing the Client

The files used to install the client are in these locations:

- Remote Display Client files for each supported platform are in the **remote_display** subdirectory in your TotalView installation directory.
- Alternatively, request a Remote Display Client from TotalView's download page at <https://totalview.io/downloads>.

Because Remote Display is built into TotalView, you don't need a separate license for it. Remote Display works with your product's license. If you have received an evaluation license, you can use Remote Display on another system.

Installing on Linux

The Linux Client installer is **RDC_installer_<release_number>-linux-x86-64.run**.

Linux Requirements:

- The xterm application must be installed on both the RDC client and RDC server host.
- The RDC server host must have a window manager installed. The RDC looks for **icewm**, **fvwm**, **twm** and **mwm**. You can override the window manager in use by providing the executable name of your window manager on the RDC's Advanced Options dialog.

Installing on Microsoft Windows

The Windows Client installer is **RDC_Installer.<release_number>-win.exe**.

Windows Requirements:

- The xterm application must be installed on the RDC server host.
- The RDC server host must have a window manager installed. The RDC looks for **icewm**, **fvwm**, **twm** and **mwm**. You can override the window manager in use by providing the executable name of your window manager on the RDC's Advanced Options dialog.

Installing on macOS

The macOS installer is **RDC_installer_<release_number>-macos.dmg**.

macOS Requirements:

- The xterm application must be installed on both the RDC client and RDC server host.
- The RDC server host must have a window manager installed. The RDC looks for **icewm**, **fvwm**, **twm** and **mwm**. You can override the window manager in use by providing the executable name of your window manager on the RDC's Advanced Options dialog.
- Catalina (macOS 10.14) and newer users must install the free Real VNC viewer from <https://www.realvnc.com/en/connect/download/viewer/macos/>.

Client Session Basics

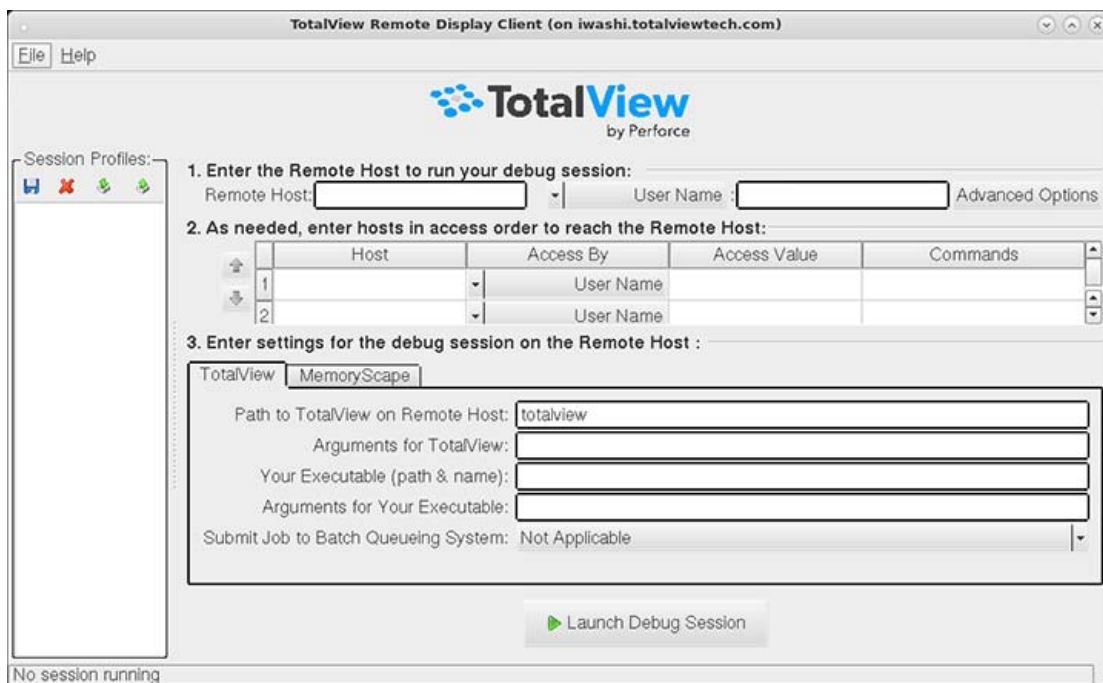
The TotalView Remote Display Client is simple to use. Just enter the required information, and the Client does the rest.

On Linux, invoke the Client with the following:

```
remote_display_client.sh
```

On Windows, either click the desktop icon or use the TVT Remote Display item in the start menu to launch the remote display dialog. On macOS, run the TVRemoteDisplayClient application from the Applications area.

Figure 127, Remote Display Client Window



The Client window displays similarly on Linux, Windows, or macOS.

Here are the basic steps:

1. Enter the Remote Host

- Remote Host:** The name of the machine upon which TotalView will execute. While the Client can execute only on specified systems (see [Remote Display Supported Platforms](#)), the remote system can be any system upon which you are licensed to run TotalView.

- **User Name** dropdown: Your user name, a public key file, or other **ssh** options.
2. **(Optional) As needed, enter hosts in access order...**(depending on your network).
If the Client system cannot directly access the remote host, specify the path. For more information, see [Naming Intermediate Hosts](#).
3. **Enter settings for the debug session on the Remote Host**
Settings required to start TotalView on the remote host. (The TotalView and MemoryScape tabs are identical.)
- **Path to TotalView on the Remote Host:** The directory on the remote host in which TotalView resides, using either an absolute or relative path. “Relative” means relative to your home directory.
 - **(Optional) Your Executable:** Either a complete or relative pathname to the program being debugged. If you leave this empty, TotalView begins executing as if you had just typed **totalview** on the remote host.
 - **Other options:**
You can add any command-line options for TotalView or your program.
TotalView options are described in the “*TotalView Debugger Command Syntax*” chapter of the *Classic TotalView Reference Guide*.
For arguments to your program, enter them in the same way as you would using the **-a** command-line option.

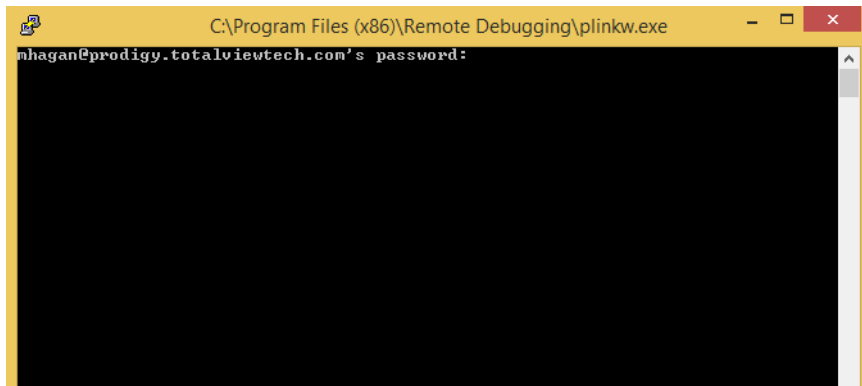
Additional options include:

- **Advanced Options:** Press the Advanced Options button to customize client/server interaction and server execution, [Advanced Options](#).
- **Submit job to batch queuing system:** You can submit jobs to the PBS Pro and LoadLeveler batch queuing systems, [Submitting a Job to a Batch Queuing System](#).

Launching the Remote Session

Next, press the **Launch Debug Session** button, which launches a password dialog box.

Figure 128, Asking for Password



Depending on how you have connected, you may be prompted twice for your password: first when Remote Display is searching ports on a remote system and another when accessing the remote host. You can often simplify logging in by using a public key file.

After entering the remote host password, a window opens on the local Client system containing TotalView as well as an xterm running on the remote host where you can enter operating system and other commands. If you do not add an executable name, TotalView displays its **File > New Debugging Session** dialog box. If you do enter a name, TotalView displays its **Process > Startup Parameters** dialog box.

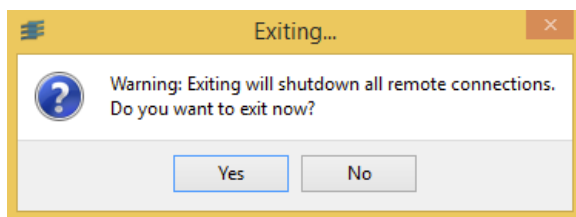
Closing the Remote Session

To close the session:

- From the Client, terminate the Viewer and Server by pressing the **End Debug Session** button. (The **Launch Debug Session** button changes to this button after you launch the session.)
- Click **Close** on the Viewer's window to remove the Viewer Window. This does not end the debugging session, so then select the Client's **End Debug Session** button. Using these two steps to end the session may be useful when many windows are running on your desktop, and the Viewer has obscured the Client.

Closing all Remote Sessions and the Client

To close all remote connections and shut down the Client window, select **File > Exit**.



Working on the Remote Host

After launching a remote session, the Client starts the Remote Display Server on the remote host where it creates a virtual window. The Server then sends the virtual window to the Viewer window running on your system. The Viewer is just another window running on the Client's system. You can interact with the Viewer window in the same way you interact with any window that runs directly on your system.

Behind the scenes, your interactions are sent to the Server, and the Server interacts with the virtual window running on the remote host. Changes made by this interaction are sent to the Viewer on your system. Performance depends on the load on the remote host and network latency.

If you are running the Client on a Windows system, these are the icons available:

Figure 129, Remote Display Client commands on Windows



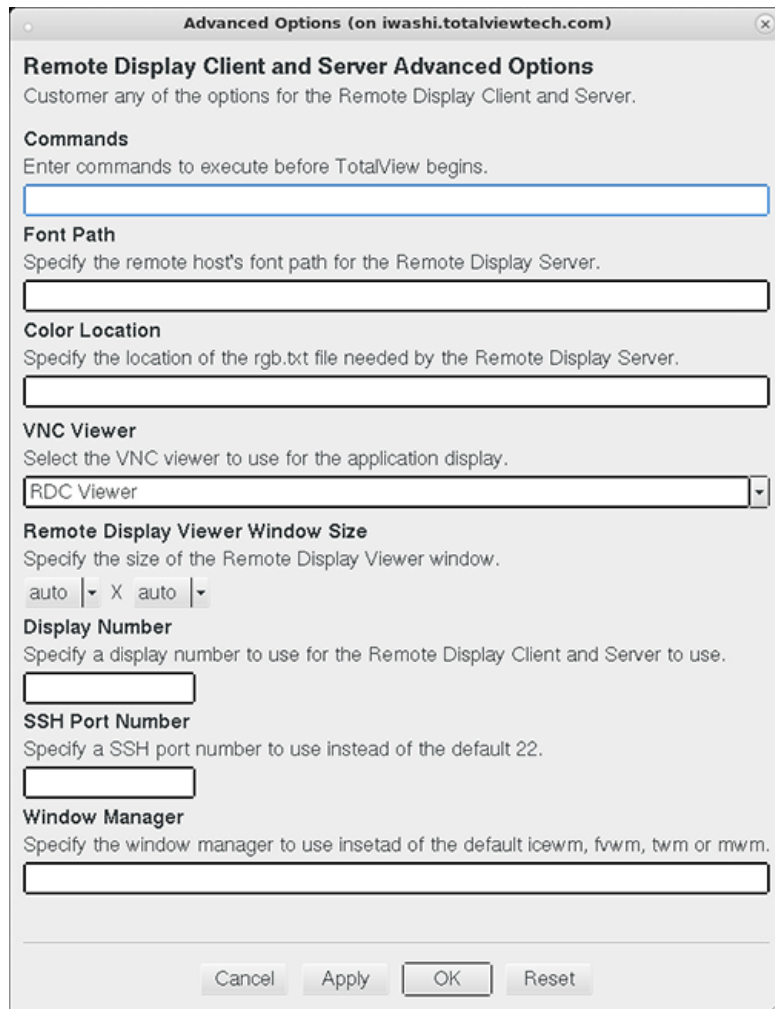
From left to right, the commands associated with these icons are:

- Connection options
- Connection information
- Full Screen - this does not change the size of the Viewer window
- Request screen refresh
- Send Ctrl-Alt-Del
- Send Ctrl-Esc
- Send Ctrl key press and release
- Send Alt key press and release
- Disconnect

Advanced Options

The Advanced Options window in [Figure 130](#) is used to customize Remote Display Client and Server interaction and to direct the Server and Remote Display Viewer execution.

Figure 130, Advanced Options Window



The screenshot shows a dialog box titled "Advanced Options (on iwashi.totalviewtech.com)". The main heading is "Remote Display Client and Server Advanced Options" with a subtitle "Customize any of the options for the Remote Display Client and Server." The dialog is organized into several sections, each with a title and a brief instruction:

- Commands:** "Enter commands to execute before TotalView begins." Below this is a single-line text input field.
- Font Path:** "Specify the remote host's font path for the Remote Display Server." Below this is a single-line text input field.
- Color Location:** "Specify the location of the rgb.txt file needed by the Remote Display Server." Below this is a single-line text input field.
- VNC Viewer:** "Select the VNC viewer to use for the application display." Below this is a dropdown menu with "RDC Viewer" selected.
- Remote Display Viewer Window Size:** "Specify the size of the Remote Display Viewer window." Below this is a size selector with "auto" on the left, "X" in the middle, and "auto" on the right, each with a small dropdown arrow.
- Display Number:** "Specify a display number to use for the Remote Display Client and Server to use." Below this is a single-line text input field.
- SSH Port Number:** "Specify a SSH port number to use instead of the default 22." Below this is a single-line text input field.
- Window Manager:** "Specify the window manager to use instead of the default icewm, fwm, twm or mwm." Below this is a single-line text input field.

At the bottom of the dialog, there are four buttons: "Cancel", "Apply", "OK", and "Reset".

Options are:

- **Commands:** Enter commands to execute before TotalView begins. For example, you can set an environment variable or change a directory location.

- **Font Path:** Specify the remote host's font path, needed by the Remote Display Server. Remote Display checks the obvious places for the font path, but on some architectures, the paths are not obvious.
- **Color Location:** Specify the location of the **rgb.txt** file needed by the Remote Display Server. Remote Display checks the obvious places for the location, but on some architectures, its location is not obvious. Providing the correct location may improve the startup time.
- **VNC Viewer:** Select the VNC viewer to use for application display.
- **Remote Display Viewer Window Size:** The default size of the Remote Display Viewer is dynamically computed, taking into account the size of the device on which the Remote Display Client is running. You can override this by selecting a custom size, which will be saved with the profile.
- **Display Number:** Specify a display number for Remote Display to use when the Client and Server connect. The Remote Display Client determines a free display number when connecting to the Server, requiring two password entries in some instances. Specifying the display number overrides the Remote Display Client determining a free number, and collisions may occur.
- **ssh Port Number:** On most systems, **ssh** uses port 22 when connecting, but in rare instances another port is used. This field allows you to override the default.
- **Window Manager:** Specify the name of the window manager. The path of the window manager you provide must be named in your **PATH** environment variable. The Server looks for (in order) the following window managers on the remote host: **icewm**, **fvwm**, **twm**, and **mwm**. Specifying a window manager may improve the startup time.

The buttons at the bottom are:

- **Cancel:** Closes the window without saving changes.
- **Apply:** Saves the changes with the profile, leaving the window open.
- **OK:** Closes the window and saves the changes with the profile.
- **Reset:** Reverts back to the previously saved values.

Naming Intermediate Hosts


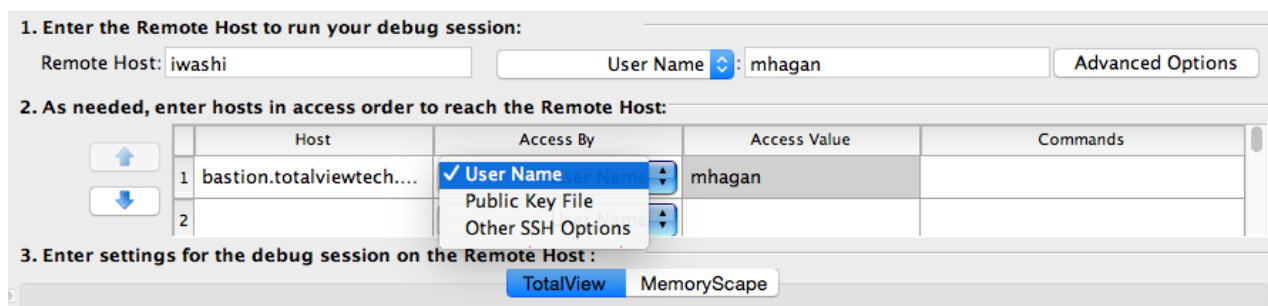
If the Client system does not have direct access to the remote host, you must specify the path, or paths, along with how you will access the host. You can enter multiple hosts; the order in which you enter them determines the order Remote Display uses to reach your remote host. Use the arrow buttons on the left () to change the order.

Figure 131, Access By Options



1. Enter the Remote Host to run your debug session:
 Remote Host: iwashi User Name: mhagan Advanced Options

2. As needed, enter hosts in access order to reach the Remote Host:

	Host	Access By	Access Value	Commands
1	bastion.totalviewtech...	User Name	mhagan	
2				

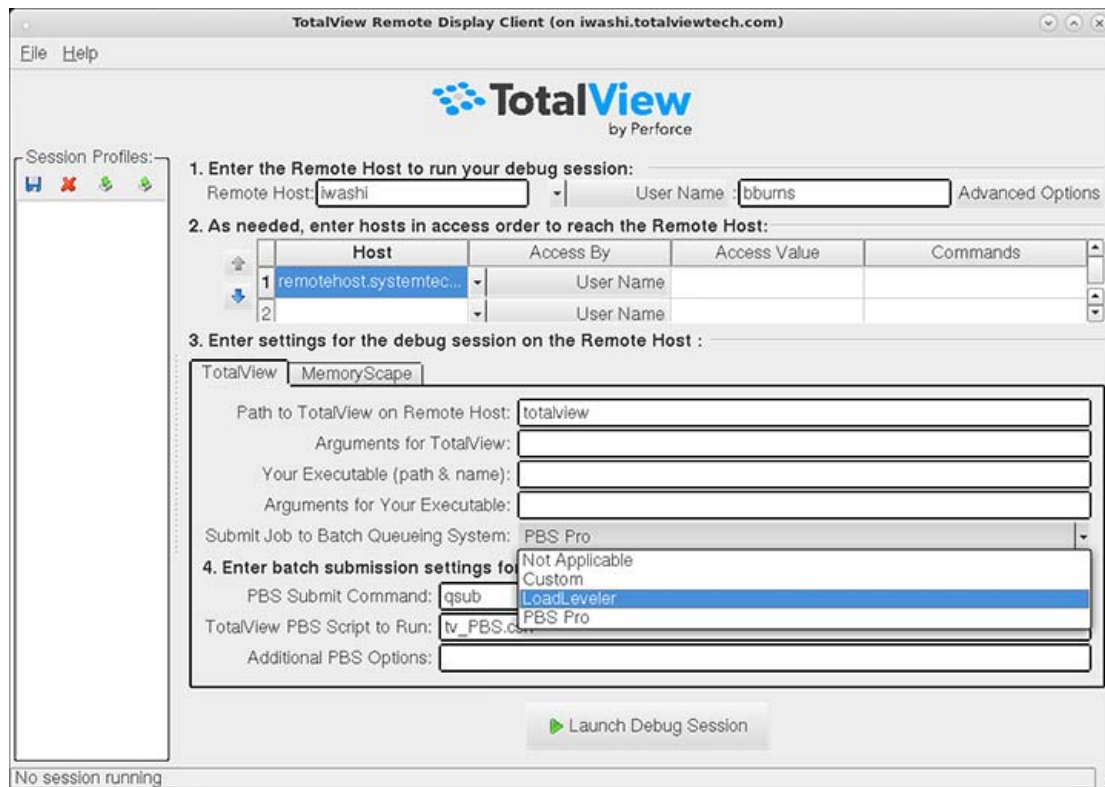
3. Enter settings for the debug session on the Remote Host :
 TotalView MemoryScape

- **Host:** The route the Client should take to access the remote host. For instance, this can be a network path or an IP address. If your network has a gateway machine, you would name it here in addition to other systems in the path to the remote host.
- **Access By/Access Value:** The most common access method is by a user name, the default. If this is incorrect for your environment, use the dropdown menu to select the correct method:
 - **User Name**, i.e. the name you enter into a shell command such as ssh to log in to the host machine. Enter this in the **Access Value** field.
 - **Public Key File**, the file that contains access information, entered into the **Access Value** field.
 - **Other SSH Options**, the ssh arguments needed to access the intermediate host. These are the same arguments you normally add to the ssh command.
- **Commands:** Commands (in a comma-separated list) to execute when connected to the remote host, before connecting to the next host.

Submitting a Job to a Batch Queuing System

TotalView Remote Display can submit jobs to the PBS Pro and LoadLeveler batch queuing systems.

Figure 132, Remote Display Window: Showing Batch Options



1. Select a batch system from the **Submit job to Batch Queuing System** dropdown list, either **PBS Pro** or **LoadLeveler**.

The default values are **qsub** for PBS Pro and **lsubmit** for LoadLeveler.

The **Script to Run** field is populated with the default scripts for either system: **tv_PBS.csh** for PBS Pro and **tv_LoadLeveler.csh** for LoadLeveler. These scripts were installed with TotalView, but can of course be changed if your system requires it. For more information, see [Batch Scripts](#).

2. (Optional) Select additional PBS or LoadLeveler options in the **Additional Options** field.

Any other required command-line options to either PBS or LoadLeveler. Options entered override those in the batch script.

3. Launch by pressing the **Launch Debug Session** button.

Behind the scenes, a job is submitted that will launch the Server and the Viewer when it reaches the head of the batch queue.

Setting Up Your Systems and Security

In order to maintain a secure environment, Remote Display uses SSH. The Remote Display Server, which runs on the remote host, allows only RFB (Remote Frame Buffer) connections from and to the remote host. No incoming access to the Server is allowed, and the Server can connect back to the Viewer only over an established SSH connection. In addition, only one Viewer connection is allowed to the Server.

As Remote Display connects to systems, a password is required. If you are allowed to use keyless ssh, you can simplify the connection process. Check with your system administrator to confirm that this kind of connection is allowed and the ssh documentation for how to generate and store key information.

Requirements for the Client to connect to the remote host:

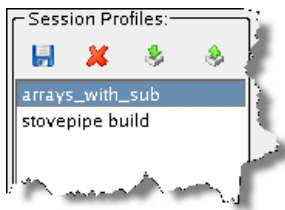
- Install TotalView on any remote systems where TotalView will be run to debug programs.
- If you use an `LM_LICENSE_FILE` environment variable to identify where your license is located, ensure that this variable is read in on the remote host. This is performed automatically if the variable's definition is contained within one of the files read by the shell when Remote Display logs in.
- ssh must be available on all non-Windows systems being accessed.
- X Windows must be available on the remote system.

Session Profile Management

The Client saves your information into a profile based on the name entered in the remote host area. You can restore these settings by clicking on the profile's name in the Session Profiles area.

Figure 133 shows two saved profiles.


Figure 133, Session Profiles

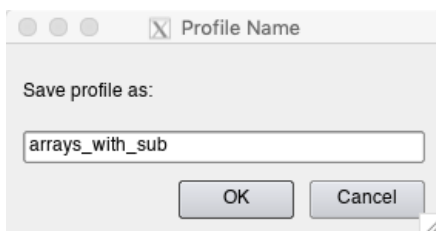


When you select a profile, the Client populates the right window with that profile's values.

If you edit the data in a text field, the Client automatically updates the profile information. If this is not what you want, click the **Create** icon to display a dialog box into which you can enter a new session profile name. The Client writes this existing data into a new profile instead of saving it to the original profile.

Saving a Profile

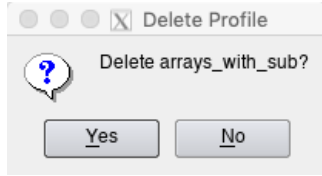
To save a profile, click the save button () or select **File > Profile > Save**, then provide a profile name in the Profile Name popup.




This command saves the profile information currently displayed in the Client window to a name you provide, placing it in the **Session Profiles** area. You do not need to save changes to the current profile as the Client automatically saves them.

Deleting a Profile

To delete a profile, click the delete button () or select **File > Profile > Delete**. This command deletes the currently selected profile and requires a confirmation.



Sharing Profiles

To import a profile, click the import button () or select **File > Profile > Import**, and then browse to the profile to import. After you import a file, it remains in your Client profile until you delete it.

To export a profile, click the export button () or select **File > Profile > Export**, browse to a directory where you want to export it, and then name the profile.

Batch Scripts

The actions that occur when you select PBS Pro or LoadLeveler within the **Submit job to Batch Queueing System** are defined in two files: **tv_PBS.csh** and **tv_LoadLever.csh**. If the actions defined in these scripts are not correct for your environment, you can either change one of these scripts or add a new script, which is the recommended procedure.

Place the script you create into *installation_dir/totalview_version/batch*. For example, you could place a new script file called **Run_Large.csh** into the *installation_dir/toolworks/totalview.2020.0/batch* directory.

tv_PBS.csh Script

Here are the contents of the **tv_PBS.csh** script file:

```
#!/bin/csh -f
#
# Script to submit using PBS
#
# These are passed to batch scheduler::
#
# account to be charged
##PBS -A VEN012
#
# pass users environment to the job
##PBS -V
#
# name of the job
#PBS -N TotalView
#
# input and output are combined to standard
##PBS -o PBSPro_out.txt
##PBS -e PBSPro_err.txt
#
##PBS -l feature=xt3
#
#PBS -l walltime=1:00:00,nodes=2:ppn=1
#
#
# Do not remove the following:
TV_COMMAND
exit
#
# end of execution script
#
```

You can uncomment or change any line and add commands to this script. The only lines you cannot change are:

```
TV_COMMAND
exit
```

tv_LoadLeveler.csh Script

Here are the contents of the **tv_Loadleveler.csh** script file:

```
#!/bin/csh -f
# @ job_type = bluegene
#@ output = tv.out.%(jobid).%(stepid)
#@ error = tv.job.err.%(jobid).%(stepid)
#@ queue
TV_COMMAND
```

You can uncomment or change any line and add commands to this script. The only line you cannot change is:

```
TV_COMMAND
```

PART V GPU Debugging

This part introduces the TotalView CUDA and AMD debuggers for debugging GPUs.

About GPU Debugging

The TotalView CUDA Debugger

- [NVIDIA CUDA Debugging Overview](#)
Introduces the CUDA debugger, including features, requirements, installation and drivers.
- [CUDA Debugging Model and Unified Display](#)
Explores setting and viewing action points in CUDA code.
- [CUDA Debugging Tutorial](#)
Discusses how to build and debug a simple CUDA program, including compiling, controlling execution, and analyzing data.
- [CUDA Problems and Limitations](#)
Issues related to limitations in the NVIDIA environment.
- [Sample CUDA Program](#)
Compilable sample CUDA program.

The TotalView AMD Debugger

- [AMD ROCm Debugging Overview](#)
Introduces the AMD ROCm debugger, including features, requirements, installation, and drivers.

- **AMD ROCm Debugging Model and Unified Display**

Explores setting and viewing action points in HIP code.

- **AMD ROCm Debugging Tutorial**

Discusses how to build and debug a simple HIP program, including compiling, controlling execution, and analyzing data.

- **AMD ROCm Problems and Limitations**

Issues related to limitations in the AMD ROCm environment.

- **Sample HIP Program**

Compilable sample HIP program.

Debugging CUDA Programs

- NVIDIA CUDA Debugging Overview
 - Installing the CUDA SDK Tool Chain
 - Directive-Based Accelerator Programming Languages
- CUDA Debugging Model and Unified Display
 - The TotalView CUDA Debugging Model
 - Pending and Sliding Breakpoints
 - Unified Source View and Breakpoint Display
- CUDA Debugging Tutorial
 - Compiling for Debugging
 - Starting a TotalView CUDA Session
 - Controlling Execution
 - Displaying CUDA Program Elements
 - GPU Core Dump Support
 - GPU Error Reporting
 - Displaying Device Information
- CUDA Problems and Limitations
 - Hangs or Initialization Failures
 - CUDA and ReplayEngine
- Sample CUDA Program

NVIDIA CUDA Debugging Overview

The TotalView CUDA debugger is an integrated debugging tool capable of simultaneously debugging CUDA code that is running on the host system and the NVIDIA® GPU. CUDA support is an extension to the standard version TotalView, and is capable of debugging 64-bit CUDA programs. Debugging 32-bit CUDA programs is currently not supported.

Supported major features:

- Debug CUDA application running directly on GPU hardware
- Set breakpoints, pause execution, and single step in GPU code
- View GPU variables in PTX registers, local, parameter, global, or shared memory
- Access runtime variables, such as threadIdx, blockIdx, blockDim, etc.
- Debug multiple GPU devices per process
- Support for the CUDA MemoryChecker
- Debug remote, distributed and clustered systems
- Support for directive-based programming languages
- Support for host debugging features

Requirements:

The CUDA SDK and a host distribution supported by NVIDIA. For SDK versions and supported NVIDIA GPUs, see the [TotalView Supported Platforms Guide](#).

Installing the CUDA SDK Tool Chain

Before you can debug a CUDA program, you must download and install the CUDA SDK software from NVIDIA using the following steps:

- Visit the NVIDIA CUDA Zone download page:
<https://developer.nvidia.com/cuda-downloads>
- Select Linux as your operating system
- Download and install the CUDA SDK Toolkit for your Linux distribution (64-bit)

By default, the CUDA SDK Toolkit is installed under `/usr/local/cuda/`. The `nvcc` compiler driver is installed in `/usr/local/cuda/bin`, and the CUDA 64-bit runtime libraries are installed in `/usr/local/cuda/lib64`.

You may wish to:

- Add `/usr/local/cuda/bin` to your `PATH` environment variable.
- Add `/usr/local/cuda/lib64` to your `LD_LIBRARY_PATH` environment variable.

Directive-Based Accelerator Programming Languages

Converting C or Fortran code into CUDA code can take some time and effort. To simplify this process, a number of directive-based accelerator programming languages have emerged. These languages work by placing compiler directives in the user's code. Instead of writing CUDA code, the user can write standard C or Fortran code, and the compiler converts it to CUDA at compile time.

TotalView currently supports Cray's OpenMP Accelerator Directives and Cray's OpenACC Directives. TotalView uses the normal CUDA Debugging Model when debugging programs that have been compiled using these directives.

CUDA Debugging Model and Unified Display

- [The TotalView CUDA Debugging Model](#)
- [Pending and Sliding Breakpoints](#)
- [Unified Source View and Breakpoint Display](#)

Debugging CUDA programs presents some challenges when it comes to setting action points. When the host process starts, the CUDA threads don't yet exist and so are not visible to the debugger for setting breakpoints. (This is also true of any libraries that are dynamically loaded using **dlopen** and against which the code was not originally linked.)

To address this issue, TotalView allows setting a breakpoint *on any line* in the Source view, whether or not it can identify executable code for that line. The breakpoint becomes either a *pending* breakpoint or a *sliding* breakpoint until the CUDA code is loaded at runtime.

The Source view provides a unified display that includes line number symbols and breakpoints that span the host executable, host shared libraries, and the CUDA ELF images loaded into the CUDA threads. This design allows you to easily set breakpoints and view line number information for the host and GPU code at the same time. This is made possible by the way CUDA threads are grouped, discussed in the section [The TotalView CUDA Debugging Model](#).

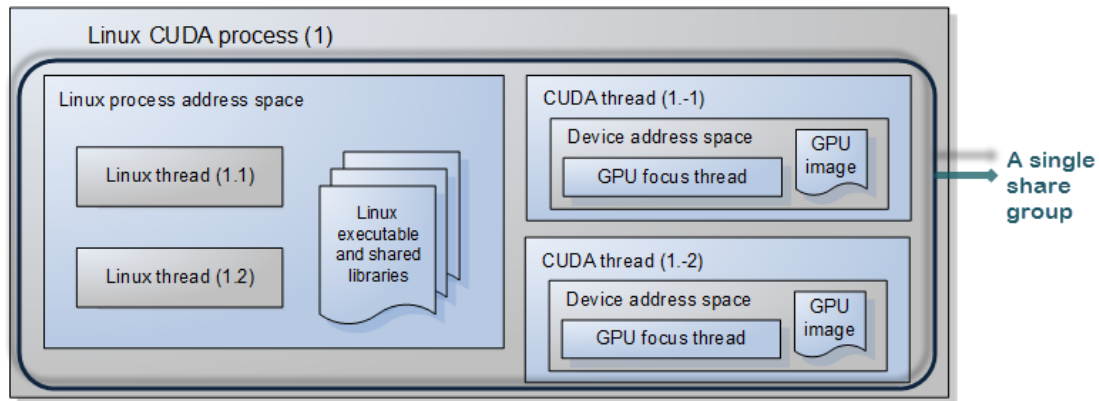
The TotalView CUDA Debugging Model

The address space of the Linux CPU process and the address spaces of the CUDA threads are placed into the same share group. Breakpoints are created and evaluated within the share group, and apply to all of the image files (executable, shared libraries, and CUDA ELF images) in the share group.

That means that a breakpoint can apply to both the CPU and GPU code. This allows setting breakpoints on source lines in the host code that are then planted in the CUDA images at the same location once the CUDA kernel starts.

Consider a Linux process consisting of two Linux pthreads and two CUDA threads. (A CUDA thread is a CUDA context loaded onto a GPU device.) [Figure 134](#) illustrates how TotalView would group the Linux and CUDA threads.

Figure 134, TotalView CUDA debugging model



The Linux host CUDA process

A Linux host CUDA process consists of:

- A Linux process address space, containing a Linux executable and a list of Linux shared libraries.
- A collection of Linux threads, where a Linux thread:
 - Is assigned a positive debugger thread ID.
 - Shares the Linux process address space with other Linux threads.
- A collection of CUDA threads, where a CUDA thread:
 - Is assigned a negative debugger thread ID.
 - Has its own address space, separate from the Linux process address space, and separate from the address spaces of other CUDA threads.
 - Has a "GPU focus thread", which is focused on a specific hardware thread (also known as a core or "lane" in CUDA lingo).

The above TotalView CUDA debugging model is reflected in the TotalView user interface and command line interface. In addition, CUDA-specific CLI commands allow you to inspect CUDA threads, change the focus, and display their status. See the **dcuda** entry in the *TotalView Reference Guide* for more information.

Pending and Sliding Breakpoints

Because CUDA threads and the host process are all in the same share group, you can create pending or sliding breakpoints on source lines and functions in the GPU code before the code is loaded onto the GPU. If TotalView can't locate code associated with a particular line in the source view, you can still plant a breakpoint there, if you know that there will be code there once the CUDA kernel loads.

Pending and sliding breakpoints are not specific to CUDA and are discussed in more detail in [Setting Source-Level Breakpoints](#).

RELATED TOPICS

Sliding breakpoints	Sliding Breakpoints
Pending breakpoints	Pending Breakpoints
Pending evalpoints	Creating a Pending Evalpoint
How the unified Source view displays breakpoints in dynamically-loaded code	Unified Source View and Breakpoint Display
Using dactions to display pending and mixed breakpoint detail before and after CUDA code has loaded.	"Examples of Actions Points in Both Host and Dynamically Loaded Code" in the dactions entry in the <i>TotalView Reference Guide</i>

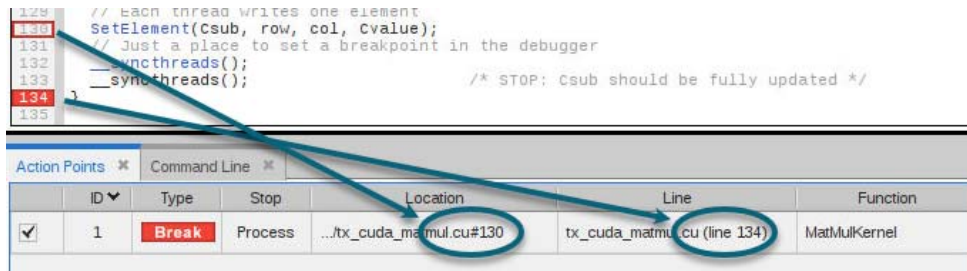
Unified Source View and Breakpoint Display

Because CUDA threads are in the same share group as are their host Linux processes, the Source view can visibly display a unified view of lines and breakpoints set in both the host code and the CUDA code. TotalView determines the equivalence of host and CUDA source files by comparing the base name and directory path of each file in the share group; if they are equal, the line number information is unified in the Source view.

A unified display is not specific to CUDA but is particularly suited to debugging CUDA programs. It is discussed in more detail in [The Source View](#).

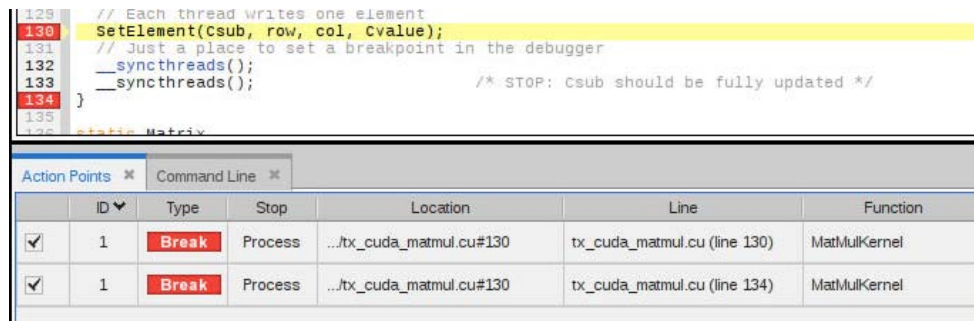
This is particularly visible when breakpoints are set. For example, [Figure 135](#) shows source code before the CUDA thread has launched. A breakpoint has been set at line 130 which slid to line 134 in the host code.

Figure 135, Source view before CUDA kernel launch



After CUDA kernel launch, Figure 136 shows that TotalView has read the line number information for the CUDA image and the slid breakpoint now displays according to the full breakpoint expression in the Action Points tab.

Figure 136, Source view after CUDA kernel launch



Notice also that the source-line breakpoints for the CUDA code have been unified with the CPU code. For example, lines 132 and 133 appeared with no bold before runtime, but after the CUDA threads have launched, TotalView is able to identify line symbol information there, so the line numbers now appear bold.

RELATED TOPICS

More on the unified Source view display

[Unified Source View Display](#)

The CUDA share group model

[The TotalView CUDA Debugging Model](#)

Using **dactions** to display pending and mixed breakpoint detail before and after CUDA code has loaded.

“Examples of Actions Points in Both Host and Dynamically Loaded Code” in the **dactions** entry in the *TotalView Reference Guide*

CUDA Debugging Tutorial

- [Compiling for Debugging](#)
- [Starting a TotalView CUDA Session](#)
- [Controlling Execution](#)
- [Displaying CUDA Program Elements](#)
- [The GPU Status View](#)
- [Enabling CUDA Memory Checker Feature](#)
- [GPU Core Dump Support](#)
- [GPU Error Reporting](#)

Compiling for Debugging

When compiling an NVIDIA CUDA program for debugging, it is necessary to pass the **-g -G** options to the **nvcc** compiler driver. These options disable most compiler optimization and include symbolic debugging information in the driver executable file, making it possible to debug the application. For example, to compile the sample CUDA program named **tx_cuda_matmul.cu** for debugging, use the following commands to compile and execute the application:

```
% /usr/local/bin/nvcc -g -G -c tx_cuda_matmul.cu -o tx_cuda_matmul.o
% /usr/local/bin/nvcc -g -G -Xlinker=-R/usr/local/cuda/lib64 \
tx_cuda_matmul.o -o tx_cuda_matmul
% ./tx_cuda_matmul
A:
[ 0][ 0] 0.000000
...output deleted for brevity...
[ 1][ 1] 131.000000
%
```

Access the source code for this CUDA program `tx_cuda_matmul.cu` program at [Sample CUDA Program](#).

Compiling for Fermi

To compile for Fermi, use the following compiler option:

```
-gencode arch=compute_20,code=sm_20
```

Compiling for Fermi and Tesla

To compile for both Fermi and Tesla GPUs, use the following compiler options:

```
-gencode arch=compute_20,code=sm_20 -gencode arch=compute_10,code=sm_10
```

See the NVIDIA documentation for complete instructions on compiling your CUDA code.

Compiling for Kepler

To compile for Kepler GPUs, use the following compiler options:

```
-gencode arch=compute_35,code=sm_35
```

See the NVIDIA documentation for complete instructions on compiling your CUDA code.

Compiling for Pascal

To compile for Pascal GPUs, use the following compiler options:

```
-gencode arch=compute_60,code=sm_60
```

See the NVIDIA documentation for complete instructions on compiling your CUDA code.

Compiling for Volta

To compile for Volta GPUs, use the following compiler options:

```
-gencode arch=compute_70,code=sm_70
```

See the NVIDIA documentation for complete instructions on compiling your CUDA code.

Starting a TotalView CUDA Session

A standard TotalView installation supports debugging CUDA applications running on both the host and GPU processors. TotalView dynamically detects a CUDA install on your system. To start the TotalView GUI or CLI, provide the name of your CUDA host executable to the **totalview** or **totalviewcli** command. For example, to start the TotalView GUI on the sample program, use the following command:

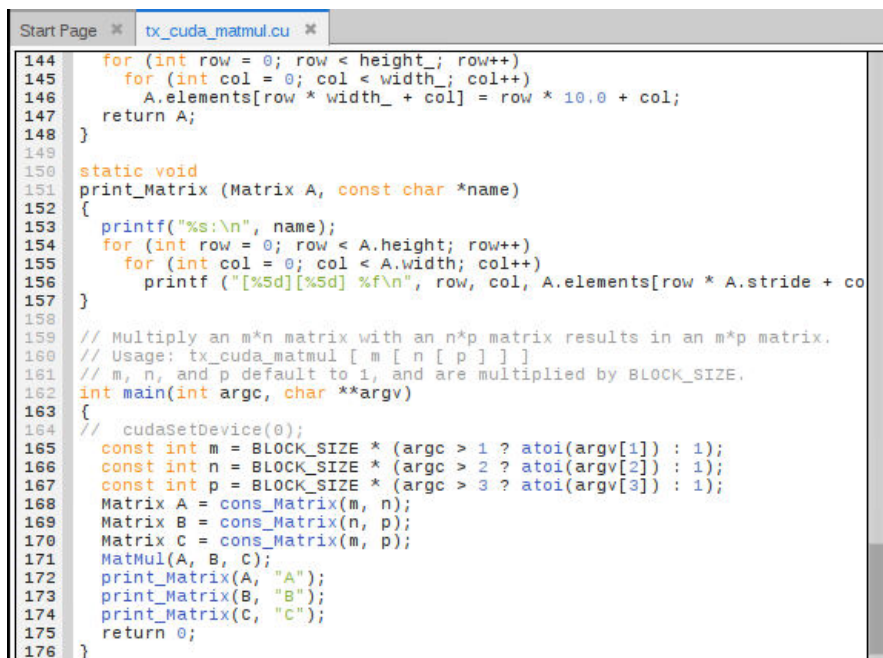
```
% totalview tx_cuda_matmul
```

If TotalView successfully loads the CUDA debugging library, it prints to the log the current CUDA debugger API version and the NVIDIA driver version:

```
CUDA library loaded: Current DLL API version is "8.0.128"; NVIDIA driver version
384.125
...
```

After reading the symbol table information for the CUDA host executable, TotalView opens the Source view focused on main in the host code, as shown in Figure 137.

Figure 137, Source view opened on CUDA host code



```

Start Page x tx_cuda_matmul.cu x
144     for (int row = 0; row < height_; row++)
145         for (int col = 0; col < width_; col++)
146             A.elements[row * width_ + col] = row * 10.0 + col;
147     return A;
148 }
149
150 static void
151 print_Matrix (Matrix A, const char *name)
152 {
153     printf("%s:\n", name);
154     for (int row = 0; row < A.height; row++)
155         for (int col = 0; col < A.width; col++)
156             printf ("%5d] [%5d] %f\n", row, col, A.elements[row * A.stride + co
157 }
158
159 // Multiply an m*n matrix with an n*p matrix results in an m*p matrix.
160 // Usage: tx_cuda_matmul [ m [ n [ p ] ] ]
161 // m, n, and p default to 1, and are multiplied by BLOCK_SIZE.
162 int main(int argc, char **argv)
163 {
164     // cudaSetDevice(0);
165     const int m = BLOCK_SIZE * (argc > 1 ? atoi(argv[1]) : 1);
166     const int n = BLOCK_SIZE * (argc > 2 ? atoi(argv[2]) : 1);
167     const int p = BLOCK_SIZE * (argc > 3 ? atoi(argv[3]) : 1);
168     Matrix A = cons_Matrix(m, n);
169     Matrix B = cons_Matrix(n, p);
170     Matrix C = cons_Matrix(m, p);
171     MatMul(A, B, C);
172     print_Matrix(A, "A");
173     print_Matrix(B, "B");
174     print_Matrix(C, "C");
175     return 0;
176 }

```

You can debug the CUDA host code using the normal TotalView commands and procedures.

Controlling Execution

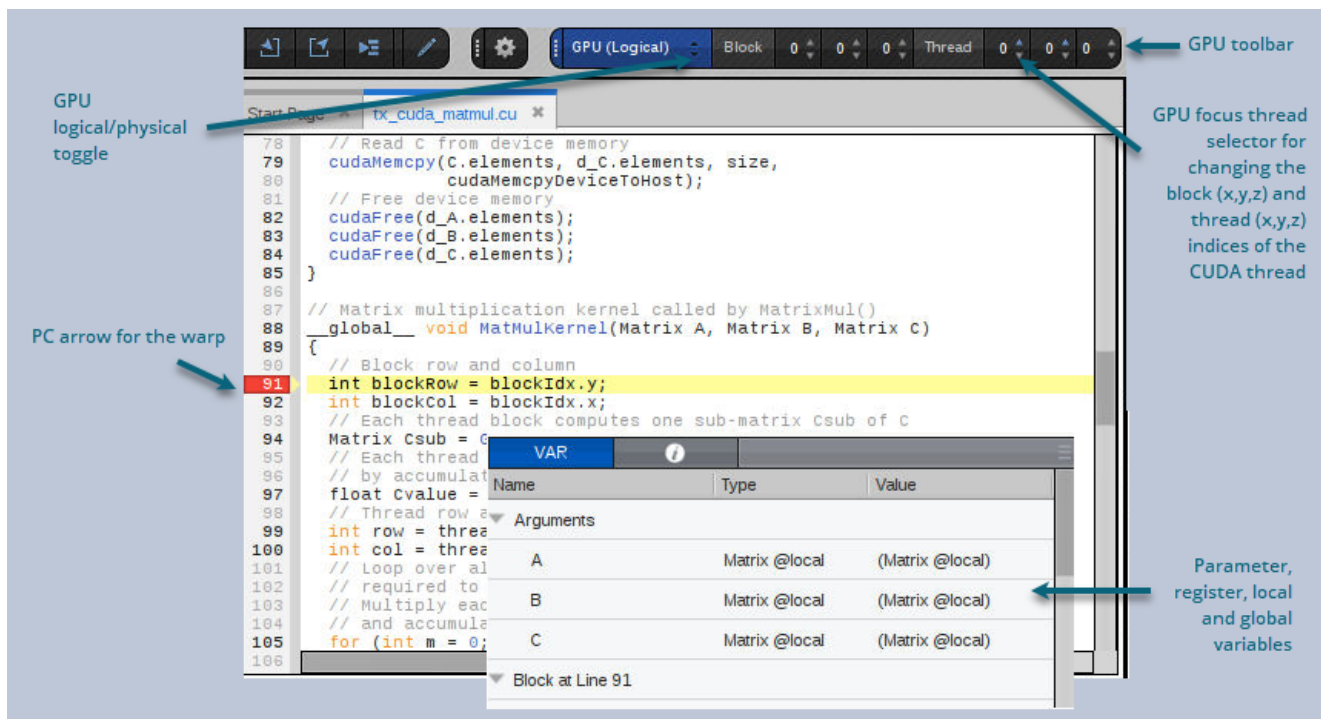
Set breakpoints in CUDA code before you start the process. If you start the process without setting any breakpoints, there are no prompts to set them afterward.

Note that breakpoints set in CUDA code will *slide* to the next host (CPU) line in the source file, but once the program is running and the CUDA code is loaded, TotalView recalculates the breakpoint expression and plants a breakpoint at the proper location in the CUDA code. (See [Sliding Breakpoints](#).)

Viewing GPU Threads

Once the CUDA kernel starts executing, it will hit the breakpoint planted in the GPU code, as shown in Figure 138.

Figure 138, CUDA thread stopped at a breakpoint, focused on GPU thread <<<(0,0,0),(0,0,0)>>>



The logical coordinates of the GPU focus threads are displayed in the GPU toolbar. You can use the GPU focus thread selector to change the GPU focus thread. When you change the GPU focus thread, the logical coordinates displayed also change, and the Call Stack and Source view are updated to reflect the state of the new GPU focus thread.

The yellow PC highlighted line in the Source view shows the execution location of the GPU focus thread. The GPU hardware threads, also known as "lanes," execute in parallel so multiple lanes may have the same PC value. The lanes may be part of the same warp (up to 32 maximum threads that are scheduled concurrently), or in different warps.

The Local Variables view shows the parameter, register and local variables for the function in the selected stack frame. The variables for the selected GPU kernel code or inlined function expansion are shown.

The Call Stack shows the stack backtrace and inlined functions:

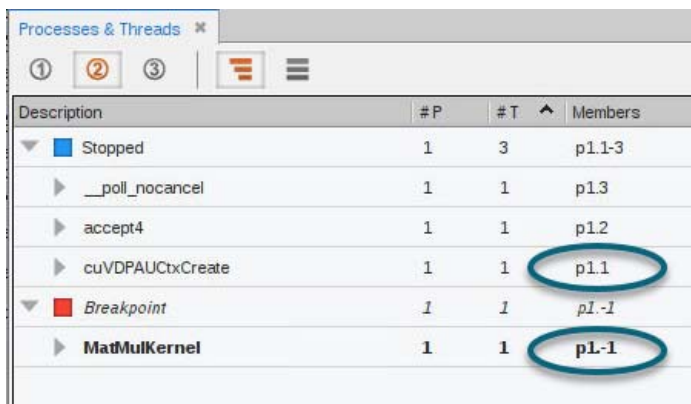


Each stack frame in the stack backtrace represents either the PC location of GPU kernel code, or the expansion of an inlined function. Inlined functions can be nested. The "return PC" of an inlined function is the address of the first instruction following the inline expansion, which is normally within the function containing the inlined-function expansion.

CUDA Thread IDs and Coordinate Spaces

TotalView gives host threads a positive debugger thread ID and CUDA threads a negative thread ID. In this example, the initial host thread in process "1" is labeled "1.1" and the CUDA thread is labeled "1.-1".

Figure 139, CUDA Thread IDs



In TotalView, a "CUDA thread" is a CUDA kernel invocation consisting of registers and memory, as well as a "GPU focus thread".

Use the "GPU focus selector" on the GPU toolbar to change the physical coordinates of the GPU focus thread:

Figure 140, GPU Focus Selector on the GPU logical space toolbar



GPU Toolbars

Two GPU toolbars display the two coordinate spaces. One is the logical coordinate space that is in CUDA terms grid and block indices: $\langle\langle\langle Bx, By, Bz \rangle, \langle Tx, Ty, Tz \rangle\rangle\rangle$. The other is the physical coordinate space that is in hardware terms the device number, streaming multiprocessor (SM) number on the device, warp (WP) number on the SM, and lane (LN) number on the warp.

Any given thread has both a thread index in this 4D physical coordinate space, and a different thread index in the 6D logical coordinate space. These indices are shown in the two GPU toolbars.

Figure 141, GPU logical and physical toolbars



To view a CUDA host thread, select a thread with a positive thread ID in the Process and Threads view. To view a CUDA GPU thread, select a thread with a negative thread ID, then use the GPU thread selector on the logical toolbar to focus on a specific GPU thread. There is one GPU focus thread per CUDA thread, and changing the GPU focus thread affects all windows displaying information for a CUDA thread and all command line interface commands targeting a CUDA thread. In other words, changing the GPU focus thread can change data displayed for a CUDA thread and affect other commands, such as single-stepping.

Note that in all cases, when you select a thread, TotalView automatically switches the Source pane, Call Stack, Data View and Action Points view to match the selected thread.

Single-Stepping GPU Code

TotalView allows you to single-step GPU code just like normal host code, but note that a single-step operation steps the entire warp associated with the GPU focus thread. So, when focused on a CUDA thread, a single-step operation advances all of the GPU hardware threads in the same warp as the GPU focus thread.

To advance the execution of more than one warp, you may either:

- set a breakpoint and continue the process
- select a line number in the source pane and select "Run To".

Execution of more than one warp also happens when single-stepping a **__syncthreads()** thread barrier call. Any source-level single-stepping operation runs all of the GPU hardware threads to the location following the thread barrier call.

Single-stepping an inlined function (nested or not) in GPU code behaves the same as single-stepping a non-inlined function. You can:

- step into an inlined function,
- step over an inlined function,
- run to a location inside an inlined function,
- single-step within an inlined function, and
- return out of an inlined function.

Halting a Running Application

You can temporarily halt a running application at any time by selecting "Halt", which halts the host and CUDA threads. This can be useful if you suspect the kernel might be hung or stuck in an infinite loop. You can resume execution at any time by selecting "Go" or by selecting one of the single-stepping buttons.

Displaying CUDA Program Elements

GPU Assembler Display

Due to limitations imposed by NVIDIA, assembler display is not supported. All GPU instructions are currently displayed as 32-bit hexadecimal words.

GPU Variable and Data Display

TotalView can display variables and data from a CUDA thread.

Add an expression from the Call Stack to the Data View to display parameter, register, local, and shared variables, as shown in [Figure 142](#). The variables are contained within the lexical blocks in which they are defined. The type of the variable determines its storage kind (register, or local, shared, constant or global memory). The address is a PTX register name or an offset within the storage kind.

Figure 142, The Data View displaying a parameter



The identifier `@local` is a TotalView built-in type storage qualifier that tells the debugger the storage kind of "A" is local storage. The debugger uses the storage qualifier to determine how to locate `A` in device memory. The supported type storage qualifiers are shown in Table 17.

Table 17: Supported Type Storage Qualifiers

Storage Qualifier	Meaning
<code>@code</code>	An offset within executable code storage
<code>@constant</code>	An offset within constant storage
<code>@generic</code>	An offset within generic storage
<code>@frame</code>	An offset within frame storage
<code>@global</code>	An offset within global storage
<code>@local</code>	An offset within local storage
<code>@parameter</code>	An offset within parameter storage
<code>@iparam</code>	Input parameter
<code>@oparam</code>	Output parameter
<code>@shared</code>	An offset within shared storage
<code>@surface</code>	An offset within surface storage
<code>@texsampler</code>	An offset within texture sampler storage
<code>@texture</code>	An offset within texture storage

Table 17: Supported Type Storage Qualifiers

Storage Qualifier	Meaning
<code>@rtvar</code>	Built-in runtime variables (see CUDA Built-In Runtime Variables)
<code>@register</code>	A PTX register name (see PTX Registers)
<code>@sregister</code>	A PTX special register name (see PTX Registers)
<code>@managed_global</code>	Statically allocated managed variable. See Managed Memory Variables .

The type storage qualifier is a necessary part of the type for correct addressing in the debugger. When you edit a type or a type cast, make sure that you specify the correct type storage qualifier for the address offset.

Managed Memory Variables

About Managed Memory

The CUDA Unified Memory component defines a managed memory space that allows all GPUs and hosts to “see a single coherent memory image with a common address space,” as described in the NVIDIA documentation “[Unified Memory Programming](#).”

Allocating a variable in managed memory avoids explicit memory transfers between host and GPUs, as any allocation created in the managed memory space is automatically migrated between the host and GPU.

A managed memory variable is marked with a “`__managed__`” memory space specifier.

How TotalView Displays Managed Variables

To make it easier to recognize and work with managed variables, TotalView annotates their address with the term “Managed”, and, for statically allocated variables, adds the `@managed_global` type qualifier.

Statically Allocated Managed Variables

For example, consider this statically allocated managed variable, declared with the `__managed__` keyword:

```
__device__ __managed__ int mv_int_initialized=10;
```

TotalView decorates the type with **@managed_global** and adds "(Managed)" to its address. Here, note that the managed variable is identified in these ways, while the regular global is not:

Name	Type	Thread ID	Value	Address
mv_int_initialized	int @managed_global	1.1	0x0000000a (10)	0x7fd07c000000 (Managed)
global_int_uninitialized	int	1.1	0x00000000 (0)	0x0068256c

Dynamically Allocated Managed Variables

Managed memory can be dynamically allocated using the **cudaMallocManaged()** function, for example:

```
cudaMallocManaged((void**)&(elm->name), sizeof(char) * (strlen("hello") + 1) );
```

Here, the Data View shows that the variable **elem** points into managed memory. That is, **elem** is a pointer and its value points into managed memory; note that the pointer's value is annotated with "(Managed)".

Note that one of its members, **name**, also points into managed memory.

Name	Type	Thread ID	Value
elem	struct DataElement @generic * ...	1-1	0x7f59b4200000 (Managed) -> struct DataElement ...
*(elem)	@generic struct DataElement	1-1	(@generic struct DataElement)
name	\$string @generic *	1-1	0x7f59b4200000 (Managed) -> "hello"
value	int	1-1	0x0000000a (10)

CUDA Built-In Runtime Variables

TotalView allows access to the CUDA built-in runtime variables, which are handled by TotalView like any other variables, except that you cannot change their values.

The supported CUDA built-in runtime variables are as follows:

- `struct dim3_16 threadIdx;`
- `struct dim3_16 blockIdx;`
- `struct dim3_16 blockDim;`
- `struct dim3_16 gridDim;`
- `int warpSize;`

The types of the built-in variables are defined as follows:

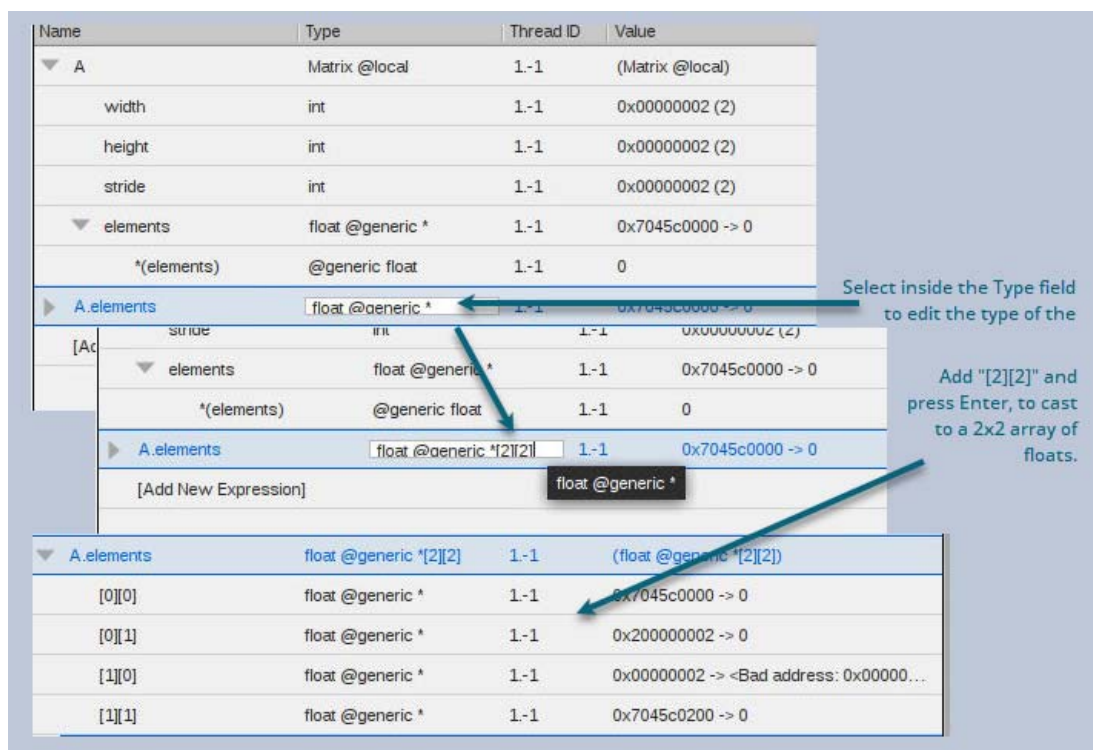
- `struct dim3_16 { unsigned short x, y, z; };`
- `struct dim2_16 { unsigned short x, y; };`

You can dive on the name of a runtime variable in the Data View, which creates a new expression. Built-in variables can also be used in the TotalView expression system.

Type Casting

The Data View allows you to edit the types of variables. This is useful for viewing an address as a different type. For example, [Figure 143](#) shows the result of casting a float in generic storage to a 2x2 array of floats in generic storage.

Figure 143, Casting to a 2x2 array of float in local storage



You can determine the storage kind of a variable by diving on the variable to create a new expression in the Data View in the graphical user interface (GUI), or by using the **dwhat** command in the command line interface (CLI).

Using the CLI to Cast

Here are some examples of using the CLI to determine variable types and to perform type casts.

When you are using the CLI and want to operate on a CUDA thread, you must first focus on the CUDA thread. The GPU focus thread in the CLI is the same as in the GUI:

```
d1.<> dfocus .-1
d1.-1
d1.-1>
```

The **dwhat** command prints the type and address offset or PTX register name of a variable. The **dwhat** command prints additional lines that have been omitted here for clarity:

```
d1.-1> dwhat A
In thread 1.-1:
Name: A; Type: @parameter const Matrix; Size: 24 bytes; Addr: 0x00000010
...
d1.-1> dwhat blockRow
In thread 1.-1:
Name: blockRow; Type: @register int; Size: 4 bytes; Addr: %r2
...
d1.-1> dwhat Csub
In thread 1.-1:
Name: Csub; Type: @local Matrix; Size: 24 bytes; Addr: 0x00000060
...
d1.-1>
```

You can use **dprint** in the CLI to cast and print an address offset as a particular type. Note that the CLI is a Tcl interpreter, so we wrap the expression argument to **dprint** in curly braces {} for Tcl to treat it as a literal string to pass into the debugger. For example, below we take the address of "A", which is at 0x10 in parameter storage. Then, we can cast 0x10 to a "pointer to a Matrix in parameter storage", as follows:

```
d1.-1> dprint {&A}
&A = 0x00000010 -> (Matrix const @parameter)
d1.-1> dprint {*(@parameter Matrix*)0x10}
*(@parameter Matrix*)0x10 = {
width = 0x00000002 (2)
height = 0x00000002 (2)
stride = 0x00000002 (2)
elements = 0x00110000 -> 0
}
d1.-1>
```

The above "@parameter" type qualifier is an important part of the cast, because without it the debugger cannot determine the storage kind of the address offset. Casting without the proper type storage qualifier usually results in "Bad address" being displayed, as follows:

```
d1.-1> dprint {*(Matrix*)0x10}
*(Matrix*)0x10 = <Bad address: 0x00000010> (struct Matrix)
d1.-1>
```

You can perform similar casts for global storage addresses. We know that "A.elements" is a pointer to a 2x2 array in global storage. The value of the pointer is 0x110000 in global storage. You can use C/C++ cast syntax:

```
d1.-1> dprint {A.elements}
A.elements = 0x00110000 -> 0
d1.-1> dprint {*(@global float(*)[2][2])0x00110000}
*(@global float(*)[2][2])0x00110000 = {
[0][0] = 0
[0][1] = 1
[1][0] = 10
```

```
[1][1] = 11
}
d1.-1>
```

Or you can use TotalView cast syntax, which is an extension to C/C++ cast syntax that allows you to simply read the type from right to left to understand what it is:

```
d1.-1> dprint {*(@global float[2][2]*)0x00110000}
*(@global float[2][2]*)0x00110000 = {
[0][0] = 0
[0][1] = 1
[1][0] = 10
[1][1] = 11
}
d1.-1>
```

If you know the address of a pointer and you want to print out the target of the pointer, you must specify a storage qualifier on both the pointer itself and the target type of the pointer. For example, if we take the address of "A.elements", we see that it is at address offset 0x20 in parameter storage, and we know that the pointer points into global storage. Consider this example:

```
d1.-1> dprint {*(@global float[2][2]*@parameter*)0x20}
*(@global float[2][2]*@parameter*)0x20 = 0x00110000 -> (@global float[2][2])
d1.-1> dprint {**(@global float[2][2]*@parameter*)0x20}
**(@global float[2][2]*@parameter*)0x20 = {
[0][0] = 0
[0][1] = 1
[1][0] = 10
[1][1] = 11
}
d1.-1>
```

Above, using the TotalView cast syntax and reading right to left, we cast 0x20 to a pointer in parameter storage to a pointer to a 2x2 array of floats in global storage. Dereferencing it once gives the value of the pointer to global storage. Dereferencing it twice gives the array in global storage. The following is the same as above, but this time in C/C++ cast syntax:

```
d1.-1> dprint {*(@global float(*@parameter*)[2][2])0x20}
*(@global float(*@parameter*)[2][2])0x20 = 0x00110000 -> (@global float[2][2])
d1.-1> dprint {**(@global float(*@parameter*)[2][2])0x20}
**(@global float(*@parameter*)[2][2])0x20 = {
[0][0] = 0
[0][1] = 1
[1][0] = 10
[1][1] = 11
}
d1.-1>
```

PTX Registers

In CUDA, PTX registers are more like symbolic virtual locations than hardware registers in the classic sense. At any given point during the execution of CUDA device code, a variable that has been assigned to a PTX register may live in one of three places:

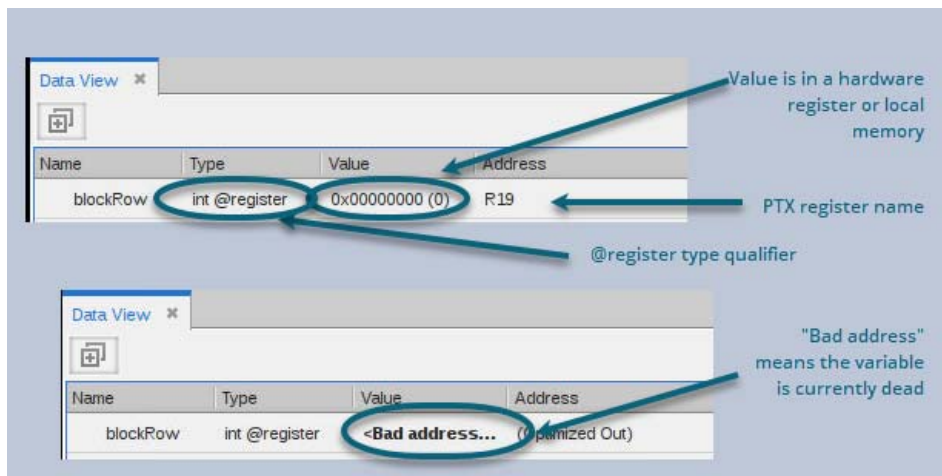
- A hardware (SAS) register

- Local storage
- Nowhere (its value is dead)

Variables that are assigned to PTX registers are qualified with the "@register" type storage qualifier, and their locations are PTX register names. The name of a PTX register can be anything, but the compiler usually assigns a name in one of the following formats: %rN, %rdN, or %fN, where N is a decimal number.

Using compiler-generated location information, TotalView maps a PTX register name to the SASS hardware register or local memory address where the PTX register is currently allocated. If the PTX register value is "live", then TotalView shows you the SASS hardware register name or local memory address. If the PTX register value is "dead", then TotalView displays **Bad address** and the PTX register name as show in Figure 144.

Figure 144, PTX register variables: one live, one dead



The GPU Status View

Along with using GPUs and other hardware accelerator processing units designed for HPC computing, programmers may also incorporate hybrid parallelism combining multiple MPI processes with other multi-threaded and device-aware programming models intended to effectively harness on-node parallelism, such as OpenMP.

Given the scales and number of layers involved, global GPU analysis features are required to be able to determine the execution state of one or more GPUs in a job. For example, a process running code on a single GPU may contain tens of thousands of executing lanes (i.e., threads). The individual lanes are grouped into warps of 32 lanes while hundreds of warps are grouped into blocks. Thousands of blocks are then spread across dozens of streaming multiprocessors.

Compounding the complexity found on a single GPU, a single process may be running code on multiple GPUs, and that process may be part of an MPI job consisting of tens, hundreds, or even thousands of processes, each running code on their own GPUs. Even a modest-sized MPI job may consist of millions of blocks and tens of millions of lanes.

The **GPU Status View** provides the ability to filter, sort, and aggregate GPU execution status and to control the focus to support setting group, process, and device (CUDA context thread) widths.

Using this view, you can:

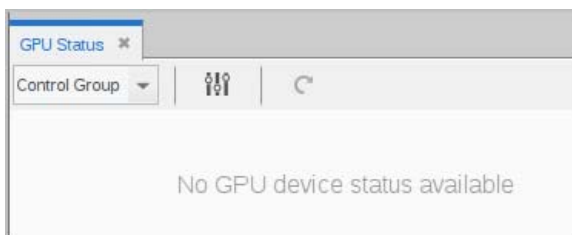
- Aggregate and filter the GPU status information from one or more GPU devices and from one or more processes, based on a number of user-selected state variables.

By default, the selected state variables used in the aggregation include the process ID, the device, the CUDA thread execution state (i.e., stopped, breakpoint, etc.), the function, and the GPU physical device information (SM, warp, lane).

- Sort the output by any of the state variables used in the aggregated tree output.
- Change the focus width of the view to show the status for either a group (control or share group), a single process (the current process in focus in the UI), or a single TotalView CUDA context thread.

The GPU Status View is a visual representation in the UI of the CLI command **dgpu_status**.

To turn on the view, load a GPU program, then select **Windows > Views > GPU Status**. The view is empty until the program starts running.

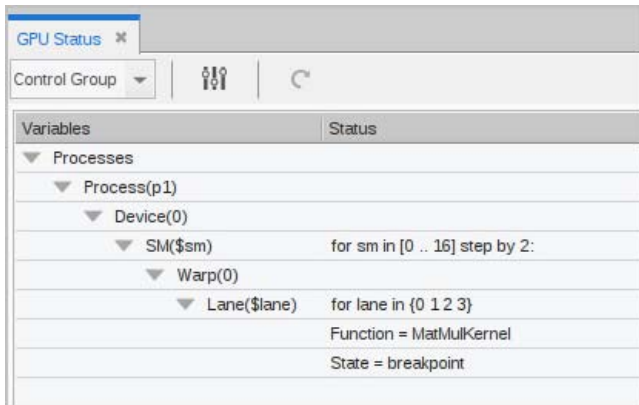


The view also opens automatically when TotalView recognizes a kernel loaded on the device and a CUDA context thread is created.

The GPU Status View Focus Options

Figure 145 shows the view after a program has loaded a GPU kernel and stopped at a breakpoint. By default, the view displays the process ID, physical GPU coordinates, function name, and execution state.

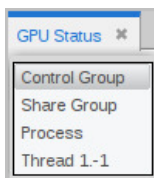
Figure 145, GPU Status View



This GPU program is running:

- A single process (p1)
- On one device (0)
- With 9 SMs, identified using a stride of "2" (which truncates the list for UI display). In this case, the SM identifiers would be 0, 2, 4, 6, 8, 10, 12, 14, and 16.
- With each SM containing one warp (0)
- With each warp containing 4 lanes (0, 1, 2, 3)
- With all lanes stopped at breakpoint in function `MatMulKernel()`.


The view has a **Focus width** dropdown which includes the Control Group, Share Group, Process, and a list of any CUDA context threads within the focus process.



The focus width is based on the process that is currently in focus in the UI. A focus of "Share Group," for example, it would be the share group containing the process in focus in the UI.

Other features of this view:

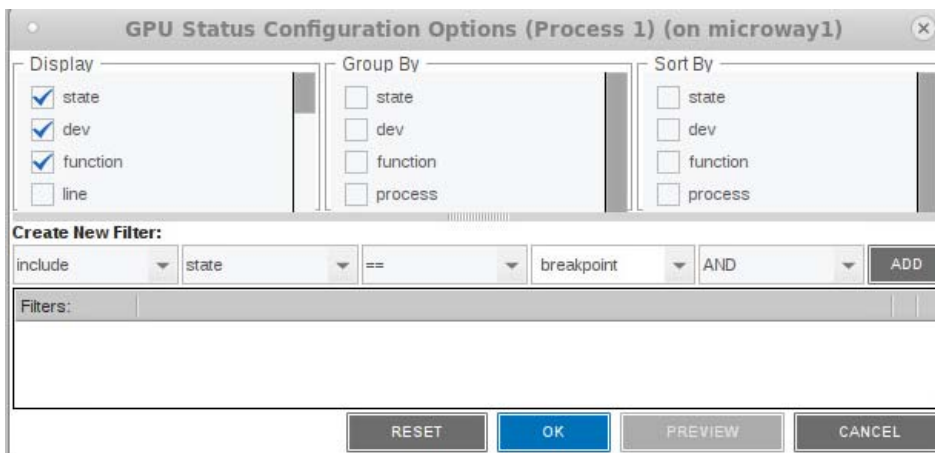
- The configure icon () launches the Configuration Options for grouping, sorting, and filtering data in the view.

- The update icon () updates the view. The view is not automatically updated when status changes. When grayed out, the view is up to date.

Configuring the GPU Status View

You can group and sort the aggregated data based on a range of state variables, using the Configuration Options dialog, opened by selecting the configure icon ().

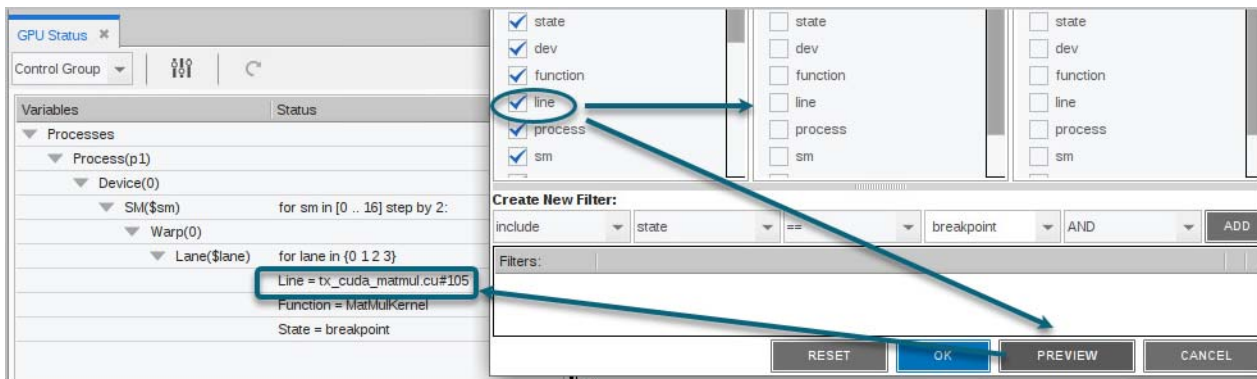
Figure 146, GPU Status Configuration Options dialog



Select Variables to Display

Variables selected in the **Display** column are placed into the **Group By** and **Sort By** columns where they are available to be selected to change the aggregation. Some variables are selected for display by default, including **state**, **dev**, **function**, **process**, **sm**, **warp**, and **lane**. Add or remove variables by checking them in the **Display** column. [Table 18](#) lists all available state variables.

Selecting or deselecting any item in this dialog activates the **Preview** button. For example, select **line** to add it to the display, then press **Preview**.



The line number is added to the display.

Preview provides just a temporary display of the view. Click **OK** to save the view or **Reset** to return the view to its default settings. **Cancel** closes the view with no changes.

Select Variables to Group or Sort By

Select one or more variables in the **Group By** or **Sort By** columns to change the grouping and sorting display.

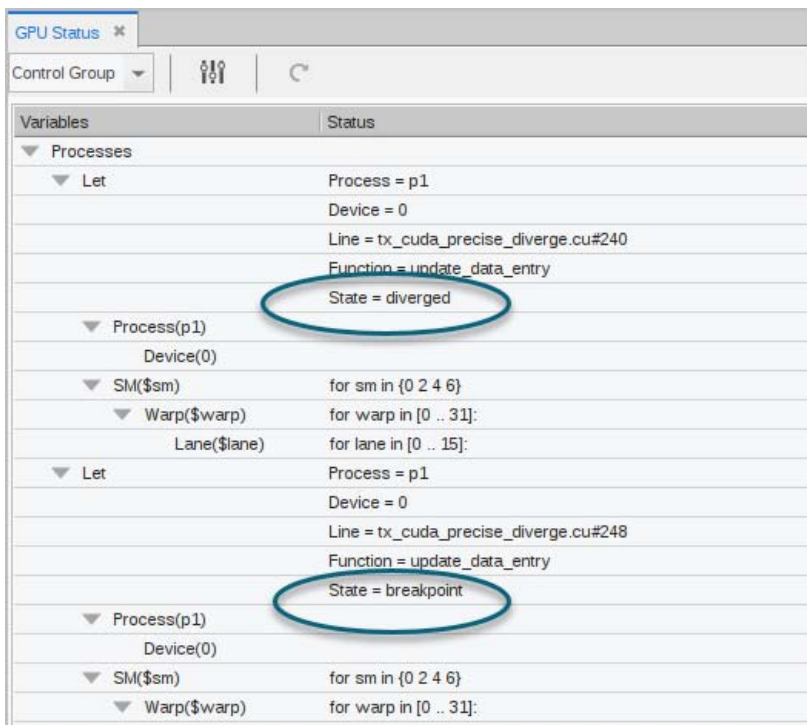
Display by logical coordinates

For example, check the logical coordinates (**bx**, **by**, **bz**, **lx**, **ly**, **lz**) under **Display**, then select them in the **Group By** menu to change the display to logical coordinates, and then group by those coordinates.

- The four active SMs, numbered 0, 2, 4, and 6, have a state of either breakpoint or diverged.
- In each SM, 32 warps, numbered 0 through 31, are valid.
- In each warp, 32 lanes are valid. However, some of the lanes have diverged, visible in the tooltip that displays over long lines. The rest are at breakpoints.

To make the view more useful, add **line** to the display and then group by lane state.

Figure 149, Grouping by thread state



While the information has not changed, it's now easy to see that, in every warp and SM, half the lanes — numbers 16 through 31 — have hit the breakpoint on line 248. The other half of the lanes — numbers 0 through 15 — have diverged and are all at line 240.

You can use an include custom filter to refine the view further. See [Configuring Custom Filters](#).

State Variables for Grouping and Sorting

Table 18: State Variables for Grouping/Sorting

State Variable	Description
state	State of a lane. An enumerated value
dev	Device ID for GPU (an integer)
function	Name of the function that contains a lane PC
line	"filename#number": the file and line number of a lane PC
process	CPU process ID, either as MPI rank or as dpid
sm	ID of an SM (an integer)
warp	ID of a warp within an SM (an integer)
pc	PC of a lane (an integer)
lane	ID of a lane within a warp (an integer)
dev_type	GPU device type (a character string)
sm_type	SM type for the GPU (a character string)
sm_count	Number of SMs in the device (an integer)
warps_per_sm	Number of warps in each SM (an integer)
lanes_per_warp	Number of lanes in each warp (an integer)
regs_per_lane	Number of registers available to each lane (an integer)
valid_warp_mask	Bit mask indicating which warps are valid in an SM
valid_lane_mask	Bit mask indicating which lanes are valid in a warp
active_lane_mask	Bit mask indicating which lanes are active in a warp
broke_lane_mask	Bit mask indicating which lanes are at breakpoints
bx	X coordinate of a block (an integer)
by	Y coordinate of a block (an integer)
bz	Z coordinate of a block (an integer)
lx	X coordinate of a thread within a block (an integer)
ly	Y coordinate of a thread within a block (an integer)
lz	Z coordinate of a thread within a block (an integer)

Configuring Custom Filters

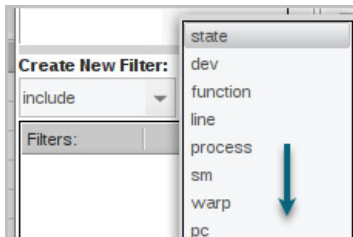
Create a custom filter in the **Create New Filter** pane in which you can include or exclude values that match or do not match the values of certain state variables from the display. This pane contains combo boxes for building the filter.



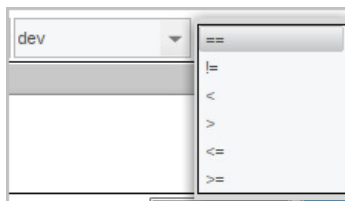
- Include/exclude:** Determine whether to “include” or “exclude” a filter. If both “include” and “exclude” filters are specified, the first one determines the overall behavior.

For example, if “include” comes first, then only threads meeting the criteria will be included. If “exclude” is first, then all threads that *don't* meet the criteria will be displayed. Subsequent “include” and “exclude” filters define exceptions to this overall behavior. See [Figure 151](#) for an example.

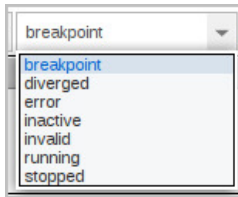
- Variable:** Choose a variable from the dropdown. The variable **state** is the default.



- Comparison operator:** Depends on the selected variable. If the state variable is a string value, the dropdown shows only == and !=. However, if it is an integer value, the dropdown contains additional comparison operators, like so:



- Value to compare:** Enter the value to compare against here. The value field is an editable text box. For variables that have a limited set of valid values, those values are displayed. For instance, choosing the variable **state** populates this dropdown with a list of possible states:



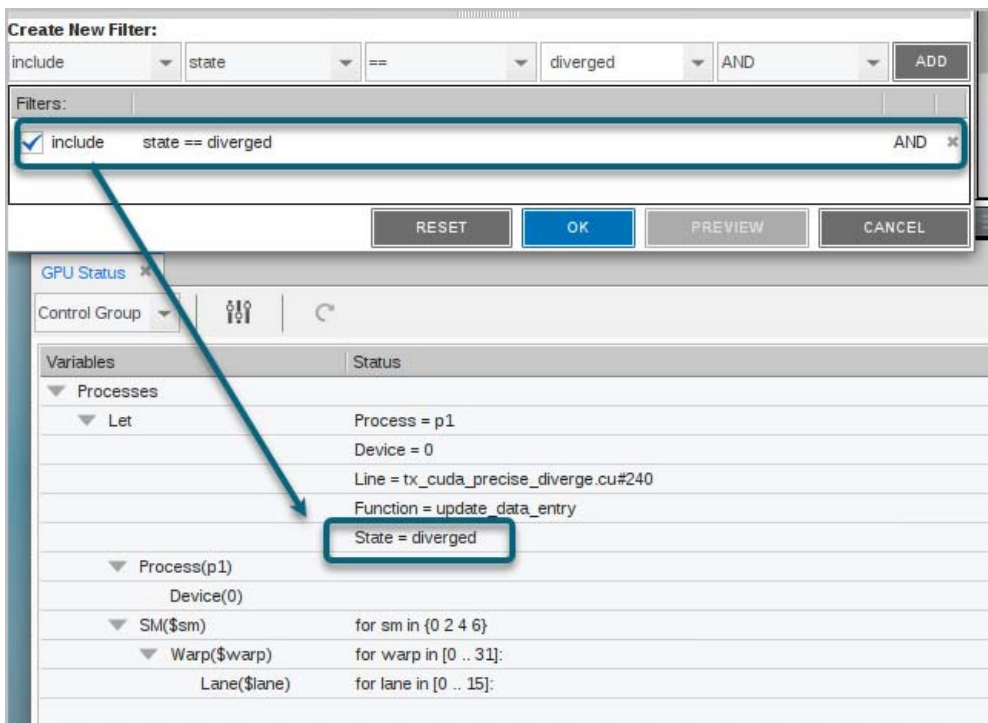
- **AND/OR:** Within a list of “include” or “exclude” filters, the predicates can be either “ANDed” or “ORed” together. If there are both AND and OR predicates, AND takes precedence over OR.

To build a filter, select the elements, then click **ADD**.

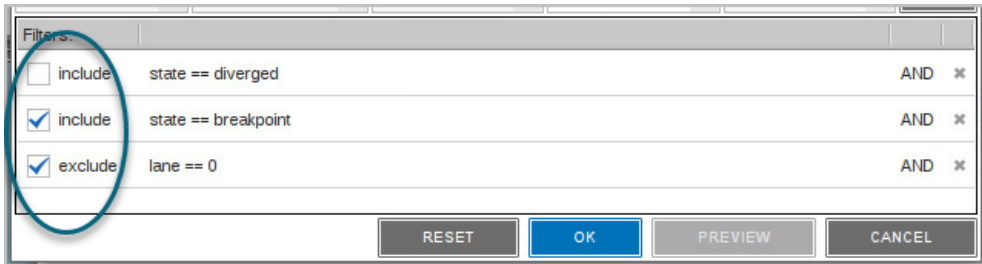
Examples

Let’s look again at the program in which the threads diverged (Figure 148). In addition to grouping the output by thread state, Figure 150 uses a custom filter to limit the display to show only those threads that have diverged.

Figure 150, Custom filter by a state of “diverged”

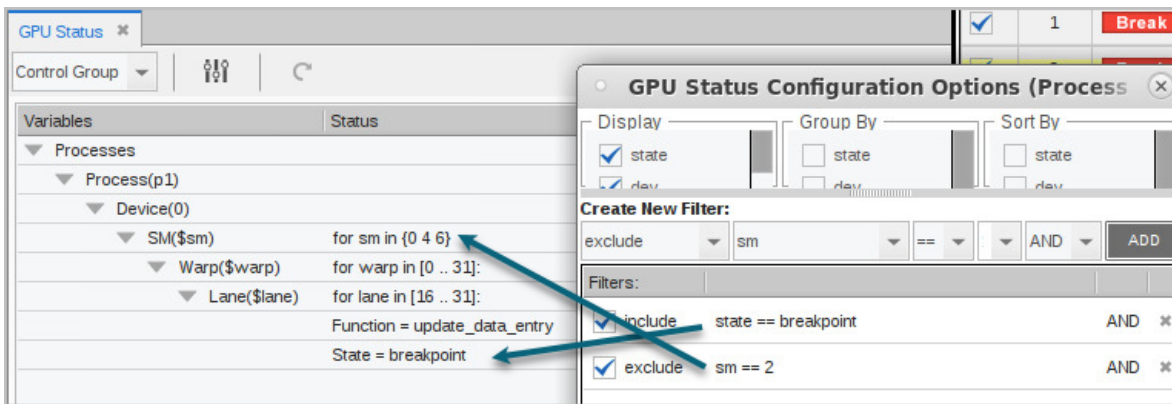


You can create multiple custom filters, and activate or de-activate them using the left checkbox.



Use “include” and “exclude” together to establish the overall filter behavior, then refine it. Figure 151 first includes all threads at breakpoint, but then excludes those running on SM 2.

Figure 151, Mixing include and exclude filters



Enabling CUDA Memory Checker Feature

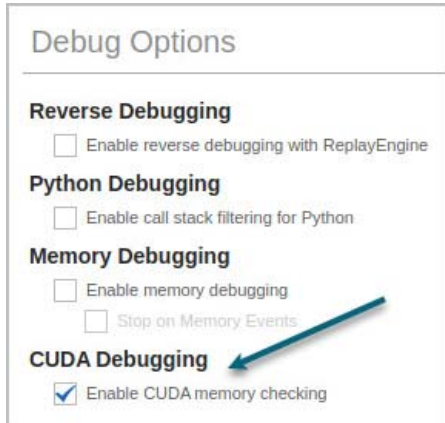
You can detect global memory addressing violations and misaligned global memory accesses by enabling the CUDA Memory Checker feature.

Enable this feature either in the UI or the CLI.

From the UI

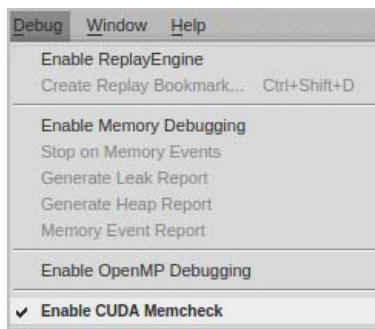
In the UI, either:

- Select "Enable CUDA memory checking" in the **Program Session** dialog.



This dialog is available when you first launch a debugging session and also via the **Process > Modify Arguments** menu from within an existing session.

- From the **Debug** menu, choose "Enable CUDA memcheck":



From the CLI

In the CLI, either:

- Pass the `-cuda_memcheck` option to the `totalview` command, for example:

```
totalview -cuda_memcheck
```

- Set the `TV::cuda_memcheck` CLI state variable to `true`. For example:

```
dset TV::cuda_memcheck true
```

Note that global memory violations and misaligned global memory accesses will be detected only while the CUDA thread is running. Detection will not happen when single-stepping the CUDA thread.

GPU Core Dump Support

CUDA GPU core dumps can be debugged just as you debug any other core dump. To obtain a GPU core dump, you must first set the `CUDA_ENABLE_COREDUMP_ON_EXCEPTION` environment variable to `1` to enable generation of a GPU core dump when a GPU exception is encountered. This option is disabled by default.

To change the default core dump file name, set the `CUDA_COREDUMP_FILE` environment variable to a specific file name. The default core dump file name is in the following format: `core.cuda.<hostname>.<pid>` where `<hostname>` is the host name of machine running the CUDA application and `<pid>` is the process identifier of the CUDA application.

To debug a GPU core dump, TotalView must be running on a machine with the CUDA SDK installed.

As with any core dump, you must also supply the name of the executable that produced the core dump:

```
totalview <executable> <core-dump-file>
```

GPU Error Reporting

By default, TotalView reports GPU exception errors as "signals." Continuing the application after these errors can lead to application termination or unpredictable results.

Table 4 lists reported errors, according to these platforms and settings:

- Exception codes `Lane Illegal Address` and `Lane Misaligned Address` are detected using all supported SDK versions when CUDA memcheck is enabled, on supported Tesla and Fermi hardware.
- All other CUDA errors are detected only for GPUs with sm_20 or higher (for example Fermi) running SDK 3.1 or higher. It is not necessary to enable CUDA memcheck to detect these errors.

Table 19: CUDA Exception Codes

Exception code	Error Precision	Error Scope	Description
CUDA_EXCEPTION_0: "Device Unknown Exception"	Not precise	Global error on the GPU	An application-caused global GPU error that does not match any of the listed error codes below.
CUDA_EXCEPTION_1: "Lane Illegal Address"	Precise (Requires memcheck on)	Per lane/thread error	A thread has accessed an illegal (out of bounds) global address.

Table 19: CUDA Exception Codes

Exception code	Error Precision	Error Scope	Description
CUDA_EXCEPTION_2: "Lane User Stack Overflow"	Precise	Per lane/thread error	A thread has exceeded its stack memory limit.
CUDA_EXCEPTION_3: "Device Hardware Stack Overflow"	Not precise	Global error on the GPU	The application has triggered a global hardware stack overflow, usually caused by large amounts of divergence in the presence of function calls.
CUDA_EXCEPTION_4: "Warp Illegal Instruction"	Not precise	Warp error	A thread within a warp has executed an illegal instruction.
CUDA_EXCEPTION_5: "Warp Out-of-range Address"	Not precise	Warp error	A thread within a warp has accessed an address that is outside the valid range of local or shared memory regions.
CUDA_EXCEPTION_6: "Warp Misaligned Address"	Not precise	Warp error	A thread within a warp has accessed an incorrectly aligned address in the local or shared memory segments.
CUDA_EXCEPTION_7: "Warp Invalid Address Space"	Not precise	Warp error	A thread within a warp has executed an instruction that attempts to access a memory space not permitted for that instruction.
CUDA_EXCEPTION_8: "Warp Invalid PC"	Not precise	Warp error	A thread within a warp has advanced its PC beyond the 40-bit address space.
CUDA_EXCEPTION_9: "Warp Hardware Stack Overflow"	Not precise	Warp error	A thread within a warp has triggered a hardware stack overflow.
CUDA_EXCEPTION_10: "Device Illegal Address"	Not precise	Global error	A thread has accessed an illegal (out of bounds) global address. For increased precision, enable memcheck.

Table 19: CUDA Exception Codes

Exception code	Error Precision	Error Scope	Description
CUDA_EXCEPTION_11: "Lane Misaligned Address"	Precise (Requires memcheck on)	Per lane/thread error	A thread has accessed an incorrectly aligned global address.
CUDA_EXCEPTION_12: "Warp Assert"	Precise	Per warp	Any thread in the warp has hit a device side assertion.
CUDA_EXCEPTION_13: "Lane Syscall Error"	Precise (Requires memcheck on)	Per lane/thread error	A thread has corrupted the heap by invoking free with an invalid address (for example, trying to free the same memory region twice)
CUDA_EXCEPTION_14: "Warp Illegal Address"	Not precise	Per warp	A thread has accessed an illegal (out of bounds) global/local/shared address. For increased precision, enable the CUDA <code>memcheck</code> option. See Enabling CUDA Memory Checker Feature .
CUDA_EXCEPTION_15: "Invalid Managed Memory Access"	Precise	Per host thread	A host thread has attempted to access managed memory currently used by the GPU.

CUDA Problems and Limitations

- Hangs or Initialization Failures
- CUDA and ReplayEngine

CUDA TotalView sits directly on top of the CUDA debugging environment provided by NVIDIA, which is still evolving and maturing. This environment contains certain problems and limitations, discussed in this chapter.

Hangs or Initialization Failures

When starting a CUDA debugging session, you may encounter hangs in the debugger or target application, initialization failures, or failure to launch a kernel. Use the following checklist to diagnose the problem:

Serialized Access

There may be at most one CUDA debugging session active per node at a time. A node cannot be shared for debugging CUDA code simultaneously by multiple user sessions, or multiple sessions by the same user. Use `ps` or other system utilities to determine if your session is conflicting with another debugging session.

Leaky Pipes

The CUDA debugging environment uses FIFOs (named pipes) located in `/tmp` and named matching the pattern `"cudagdb_pipe.N.N"`, where `N` is a decimal number. Occasionally, a debugging session might accidentally leave a set of pipes lying around. You may need to manually delete these pipes in order to start your CUDA debugging session:

```
rm /tmp/cudagdb_pipe.*
```

If the pipes were leaked by another user, that user will own the pipes and you may not be able to delete them. In this case, ask the user or system administrator to remove them for you.

Orphaned Processes

Occasionally, a debugging session might accidentally orphan a process. Orphaned processes might go compute bound or prevent you or other users from starting a debugging session. You may need to manually kill orphaned CUDA processes in order to start your CUDA debugging session or stop a compute-bound process. Use system tools such as `ps` or `top` to find the processes and kill them using the shell `kill` command. If the process were orphaned by another user, that user will own the processes and you may not be able to kill them. In this case, ask the user or system administrator to kill them for you.

Multi-threaded Programs on Fermi

We have seen problems debugging some multi-threaded CUDA programs on Fermi, where the CUDA debugging environment kills the debugger with an internal error (SIGSEGV). We are working with NVIDIA to resolve this problem.

CUDA and ReplayEngine

You can enable ReplayEngine while debugging CUDA code; that is, ReplayEngine record mode will work. However, ReplayEngine does not support replay operations when focused on a CUDA thread. If you attempt this, you will receive a Not -Supported error.

Sample CUDA Program

```
/*
 * NVIDIA CUDA matrix multiply example straight out of the CUDA
 * programming manual, more or less.
 */

#include <cuda.h>
#include <stdio.h>

// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
int width; /* number of columns */
int height; /* number of rows */
int stride;
float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col, float value)
{
A.elements[row * A.stride + col] = value;
}

// Thread block size
#define BLOCK_SIZE 2

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
Matrix Asub;
Asub.width = BLOCK_SIZE;
Asub.height = BLOCK_SIZE;
Asub.stride = A.stride;
Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
+ BLOCK_SIZE * col];
return Asub;
}

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
```

```

// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
// Load A and B to device memory
Matrix d_A;
d_A.width = d_A.stride = A.width; d_A.height = A.height;
size_t size = A.width * A.height * sizeof(float);
cudaMalloc((void*)&d_A.elements, size);
cudaMemcpy(d_A.elements, A.elements, size,
cudaMemcpyHostToDevice);
Matrix d_B;
d_B.width = d_B.stride = B.width; d_B.height = B.height;
size = B.width * B.height * sizeof(float);
cudaMalloc((void*)&d_B.elements, size);
cudaMemcpy(d_B.elements, B.elements, size,
cudaMemcpyHostToDevice);
// Allocate C in device memory
Matrix d_C;
d_C.width = d_C.stride = C.width; d_C.height = C.height;
size = C.width * C.height * sizeof(float);
cudaMalloc((void*)&d_C.elements, size);
// Invoke kernel
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
// Read C from device memory
cudaMemcpy(C.elements, d_C.elements, size,
cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatrixMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
// Block row and column
int blockRow = blockIdx.y;
int blockCol = blockIdx.x;
// Each thread block computes one sub-matrix Csub of C
Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
// Each thread computes one element of Csub
// by accumulating results into Cvalue
float Cvalue = 0;
// Thread row and column within Csub
int row = threadIdx.y;
int col = threadIdx.x;
// Loop over all the sub-matrices of A and B that are
// required to compute Csub
// Multiply each pair of sub-matrices together
// and accumulate the results
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
// Get sub-matrix Asub of A
Matrix Asub = GetSubMatrix(A, blockRow, m);
// Get sub-matrix Bsub of B
Matrix Bsub = GetSubMatrix(B, m, blockCol);
// Shared memory used to store Asub and Bsub respectively
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

```

```

// Load Asub and Bsub from device memory to shared memory
// Each thread loads one element of each sub-matrix
As[row][col] = GetElement(Asub, row, col);
Bs[row][col] = GetElement(Bsub, row, col);
// Synchronize to make sure the sub-matrices are loaded
// before starting the computation
__syncthreads();
// Multiply Asub and Bsub together
for (int e = 0; e < BLOCK_SIZE; ++e)
Cvalue += As[row][e] * Bs[e][col];
// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();
}
// Write Csub to device memory
// Each thread writes one element
SetElement(Csub, row, col, Cvalue);
// Just a place to set a breakpoint in the debugger
__syncthreads();
__syncthreads(); /* STOP: Csub should be fully updated */
}

static Matrix
cons_Matrix (int height_, int width_)
{
Matrix A;
A.height = height_;
A.width = width_;
A.stride = width_;
A.elements = (float*) malloc(sizeof(*A.elements) * width_ * height_);
for (int row = 0; row < height_; row++)
for (int col = 0; col < width_; col++)
A.elements[row * width_ + col] = row * 10.0 + col;
return A;
}

static void
print_Matrix (Matrix A, char *name)
{
printf("%s:\n", name);
for (int row = 0; row < A.height; row++)
for (int col = 0; col < A.width; col++)
printf ("[%5d][%5d] %f\n", row, col, A.elements[row * A.stride + col]);
}

// Multiply an m*n matrix with an n*p matrix results in an m*p matrix.
// Usage: tx_cuda_matmul [ m [ n [ p ] ] ]
// m, n, and p default to 1, and are multiplied by BLOCK_SIZE.
int main(int argc, char **argv)
{
// cudaSetDevice(0);
const int m = BLOCK_SIZE * (argc > 1 ? atoi(argv[1]) : 1);
const int n = BLOCK_SIZE * (argc > 2 ? atoi(argv[2]) : 1);
const int p = BLOCK_SIZE * (argc > 3 ? atoi(argv[3]) : 1);
Matrix A = cons_Matrix(m, n);
Matrix B = cons_Matrix(n, p);
Matrix C = cons_Matrix(m, p);
MatMul(A, B, C);
print_Matrix(A, "A");
}

```

```
print_Matrix(B, "B");  
print_Matrix(C, "C");  
return 0;  
}
```

Debugging AMD ROCm Programs

- AMD ROCm Debugging Overview
 - Installing the AMD Tool Chain
- AMD ROCm Debugging Model and Unified Display
 - The TotalView AMD ROCm Debugging Model
 - Pending and Sliding Breakpoints
 - Unified Source View and Breakpoint Display
- AMD ROCm Debugging Tutorial
 - Compiling for Debugging
 - Starting a TotalView ROCm Session
 - Displaying ROCm Program Elements
 - GPU Core Dump Support
 - GPU Error Reporting
- AMD ROCm Problems and Limitations
 - Hangs or Initialization Failures
- Sample HIP Program

AMD ROCm Debugging Overview

The TotalView debugger is an integrated debugging tool capable of simultaneously debugging HIP (Heterogeneous Interface for Portability) code running on both the host system and in the ROCm environment on an AMD GPU.

Supported major features:

- Debug HIP applications running directly on AMD GPU hardware
- Set breakpoints, pause execution, and single step in HIP code
- Access runtime variables, such as `threadIdx`, `blockIdx`, `blockDim`, etc.
- Debug multiple GPU devices per process
- Debug remote, distributed, and clustered systems
- Support for host debugging features

Requirements:

The AMD ROCm platform and a host distribution supported by AMD. For versions and supported AMD GPUs, see the [TotalView Supported Platforms Guide](#).

Installing the AMD Tool Chain

Before you can debug an AMD program, download and install the ROCm software from AMD. Visit the installation and download page at <https://docs.amd.com>. During the installation, select a supported Linux distribution.

By default, ROCm is installed in `/opt/rocm-<version>`. After installing, you may wish to:

- Add the ROCm binaries to your `PATH`:

```
export PATH=$PATH:/opt/rocm-<version>/bin:/opt/rocm-<version>/opencl/bin
```
- Add the ROCm library version to your `LD_LIBRARY_PATH` environment variable:

```
export LD_LIBRARY_PATH=/opt/rocm-<version>/lib:/opt/rocm-<version>/lib64
```

AMD ROCm Debugging Model and Unified Display

- [The TotalView AMD ROCm Debugging Model](#)
- [Pending and Sliding Breakpoints](#)
- [Unified Source View and Breakpoint Display](#)

Debugging HIP programs running on an AMD GPU presents some challenges when it comes to setting action points. When the host process starts, the GPU code objects have not yet been loaded onto the GPU, so the GPU code is not yet visible to the debugger for setting breakpoints. (This is also true of any libraries that are dynamically loaded using **dlopen** and against which the code was not originally linked.)

To address this issue, TotalView allows setting a breakpoint *on any line* in the Source view, whether or not it can identify executable code for that line. The breakpoint becomes either a *pending* breakpoint or a *sliding* breakpoint until the GPU code is loaded onto the GPU agent at runtime.

The Source view provides a unified display that includes line number symbols and breakpoints that span the host executable, host shared libraries, and the GPU ELF images loaded into the GPU agents. This design allows you to easily set breakpoints and view line number information for the host and HIP code at the same time. This is made possible by the way GPU agents are grouped, discussed in the section [The TotalView AMD ROCm Debugging Model](#).

The TotalView AMD ROCm Debugging Model

Each GPU agent, or device, is represented in the debugger as a single TotalView thread, called the “AMD GPU agent TotalView thread” or **GPU agent thread** for short in this documentation. (This is not to be confused with a single thread of execution, sometimes called a “work-item” or “lane.”). In TotalView, CPU threads have positive IDs, while GPU agent threads have negative IDs.

Breakpoints apply to both the CPU and GPU code

The address spaces of the Linux CPU process and the address spaces of the ROCm threads are placed into the same share group. Breakpoints are created and evaluated within the share group, and apply to all of the image files (executable, shared libraries, and ROCm ELF images) in the share group.

That means that a source-level breakpoint can apply to both the CPU and GPU code. This allows setting breakpoints on source lines in the CPU code that are then planted at the same source location in the GPU code, once the GPU kernel starts.

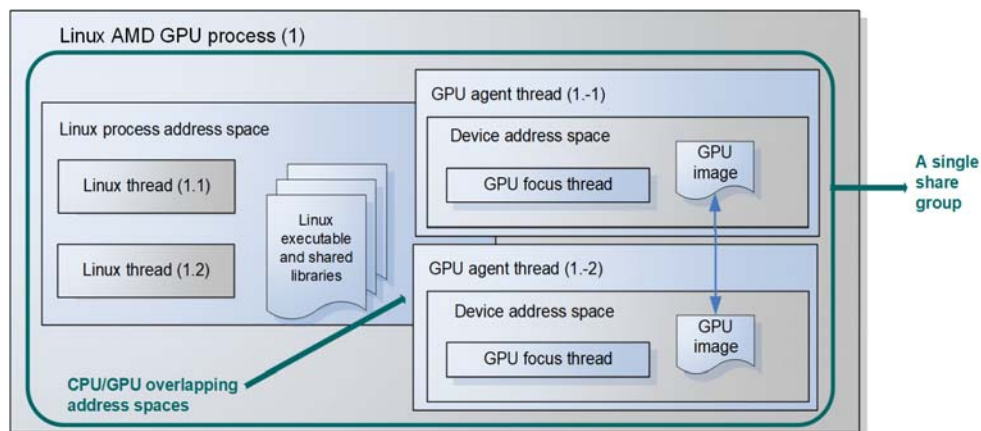
GPU and Linux address spaces overlap and each agent is mapped to all other agents

Unlike CUDA GPUs, the AMD GPU address spaces and Linux process address spaces overlap within a process. Each device in a process gets its own range of virtual addresses. In addition, the global memory of every GPU agent in a process is mapped into every other agent; for example, if there are eight GPUs in the system, each GPU agent is also mapped into the other GPU agents, as well as into the Linux process. However, TotalView models the global address spaces of the Linux process and GPU agents as all being discrete.

Consider a Linux process consisting of two Linux pthreads and two AMD GPU agents. [Figure 152](#) illustrates how TotalView would group the Linux and AMD GPU agents.

NOTE: An AMD GPU agent is represented in TotalView as a thread with a negative thread ID, called the **GPU agent thread**.

Figure 152, TotalView AMD GPU debugging model



The Linux host ROCm process

A Linux host AMD GPU process consists of:

- A Linux process address space, containing a Linux executable and a list of Linux shared libraries.
- A collection of Linux threads, where a Linux thread:
 - Is assigned a positive debugger thread ID.
 - Shares the Linux process address space with other Linux threads.

- A collection of AMD GPU agents, where a GPU agent:
 - Is assigned a negative TotalView thread ID.
 - Has its own global address space, separate from the Linux process address space, and separate from the global address spaces of other GPU agents.
 - Has a "GPU focus thread," which is focused on a specific hardware work-item (also known as a lane).

The above TotalView AMD ROCm debugging model is reflected in the TotalView user interface and command line interface. In addition, ROCm-specific CLI commands allow you to inspect ROCm work-items, change the focus, and display their status.

Disabling Deferred GPU Image Loading

The HIP runtime normally does not load GPU ELF images onto the GPUs until a kernel is launched. However, it supports a feature that allows loading the GPU images onto the GPU before the program enters `main()`. As a result, the debugger can read the GPU symbol table information before the kernel is launched, allowing you to plant breakpoints and inspect the actual kernel code.

Setting the `HIP_ENABLE_DEFERRED_LOADING` environment variable to 0 disables deferred image loading by the HIP runtime. For example:

```
export HIP_ENABLE_DEFERRED_LOADING=0
```

See the AMD documentation for more information on the use and implications of this environment variable.

Pending and Sliding Breakpoints

Because AMD GPU threads and the CPU process are all in the same share group, you can create pending or sliding breakpoints on source lines and functions in the GPU code before the code is loaded onto the GPU. If TotalView can't locate code associated with a particular line in the source view, you can still plant a breakpoint there, if you know that there will be code there once the GPU kernel loads.

Pending and sliding breakpoints are not specific to GPUs and are discussed in more detail in [Setting Source-Level Breakpoints](#).

RELATED TOPICS

Sliding breakpoints	Sliding Breakpoints
Pending breakpoints	Pending Breakpoints
Pending evalpoints	Creating a Pending Evalpoint

RELATED TOPICS

How the unified Source view displays breakpoints in dynamically-loaded code

[Unified Source View and Breakpoint Display](#)

Using **dactions** to display pending and mixed breakpoint detail before and after ROCm code has loaded.

“Examples of Actions Points in Both Host and Dynamically Loaded Code” in the **dactions** entry in the *TotalView Reference Guide*

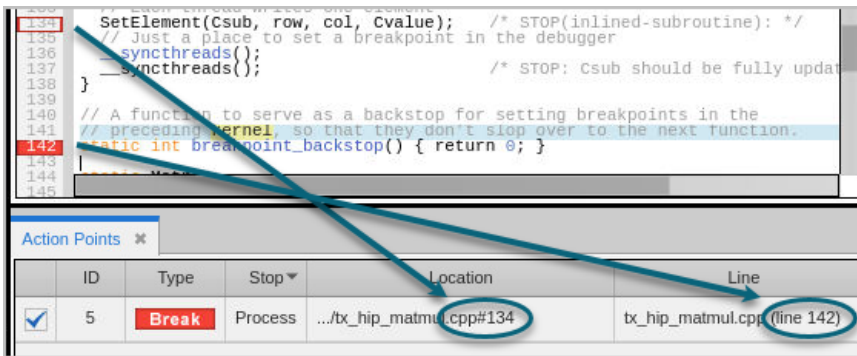
Unified Source View and Breakpoint Display

Because the TotalView GPU agent threads are in the same share group as are their host Linux processes, the Source view displays a unified view of lines and breakpoints set in both the host code and the GPU code. TotalView determines the equivalence of host and HIP source files by comparing the base name and directory path of each source file in the share group; if they are equal, the line number information is unified in the Source view.

NOTE: A unified display is not specific to GPUs but is well suited to debugging HIP programs. It is discussed in more detail in [The Source View](#).

This unified display is particularly visible when breakpoints are set. For example, [Figure 153](#) shows source code before the kernel has launched. A breakpoint has been set at line 134 which slid to line 142 in the host code.

Figure 153, Source view before GPU kernel launch



After kernel launch, [Figure 154](#) shows that TotalView has read the line number information for the GPU image, and there is now a breakpoint corresponding to the line number in the full breakpoint expression in the Action Points tab.

Figure 154, Source view after kernel launch

```

133 // Each thread writes one element
134 SetElement(Csub, row, col, Cvalue); /* STOP(inlined-subroutine): */
135 // Just a place to set a breakpoint in the debugger
136 __syncthreads();
137 __syncthreads(); /* STOP: Csub should be fully updated */
138 }
139
140 // A function to serve as a backstop for setting breakpoints in the
141 // preceding kernel, so that they don't slip over to the next function.
142 static int breakpoint_backstop() { return 0; }
143
144
145

```

ID	Type	Stop	Location	Line
5	Break	Process	.../tx_hip_matmul.cpp#134	tx_hip_matmul.cpp (line 134)
5	Break	Process	.../tx_hip_matmul.cpp#134	tx_hip_matmul.cpp (line 142)

Notice also that the source-line breakpoint markers for the kernel code have been unified with the CPU code. For example, lines 134 and 136-138 appeared with no bold before runtime, but after the kernel was launched, TotalView was able to identify line number symbol information there, so the line numbers now appear bold.

RELATED TOPICS

More on the unified Source view display

[Unified Source View Display](#)

The AMD ROCm share group model

[The TotalView AMD ROCm Debugging Model](#)

Using **dactions** to display pending and mixed breakpoint detail before and after HIP code has loaded.

“Examples of Actions Points in Both Host and Dynamically Loaded Code” in the **dactions** entry in the *TotalView Reference Guide*

AMD ROCm Debugging Tutorial

- Compiling for Debugging
- Starting a TotalView ROCm Session
- Displaying ROCm Program Elements

NOTE: Support for debugging ROCm HIP programs is enabled by default. If you want to disable it, use the `-no_rocm` option when starting TotalView, like this:

```
totalview -no_rocm
```

To re-enable it:

```
totalview -rocm
```

Compiling for Debugging

When compiling a HIP program for debugging, pass the `-g` option to the compiler for source-level debugging. In addition, the option `-O0` turns off all optimizations to better facilitate debugging. For example, to compile the HIP program named `tx_hip_matmul` for debugging, use the following:

ROCm 5.4:

```
/opt/rocm-5.4.0/bin/hipcc -O0 -g -o tx_hip_matmul tx_hip_matmul.cpp
```

Access the source code for this HIP program `tx_hip_matmul` at [Sample HIP Program](#).

Starting a TotalView ROCm Session

A standard TotalView installation supports debugging HIP applications running on both the host and GPU processors. TotalView dynamically detects a ROCm installation on your system. To start the TotalView GUI or CLI, provide the name of your ROCm host executable to the `totalview` or `totalviewcli` command. For example, to start the TotalView UI on the sample program, use the following command:

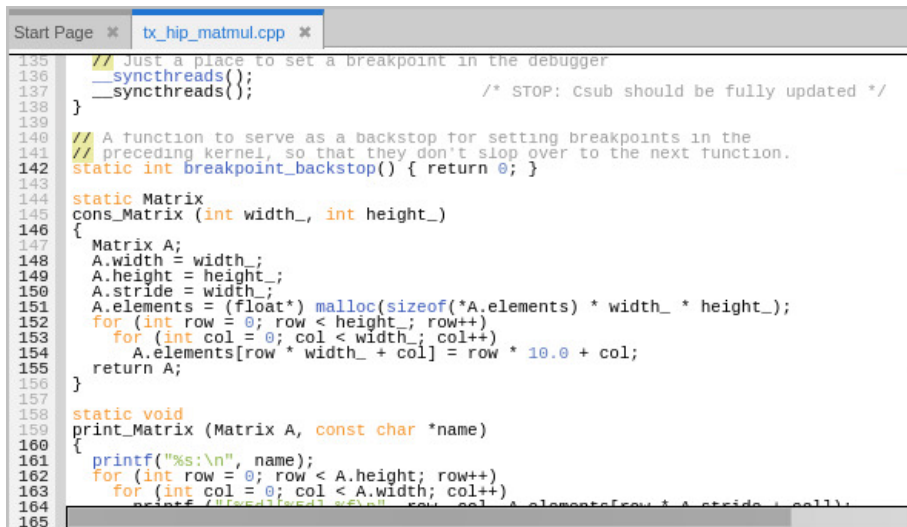
```
% totalview tx_hip_matmul
```

If TotalView successfully loads the HIP debugging library, it prints to the log the current ROCm debugger API version (ROCdbgapi) and the ROCm driver version:

```
ROCm debug library loaded: Current DLL API version is "0.68.0" (build "0.68.0-rocm-
rel-5.4-72")
...
```

After reading the symbol table information for the GPU host executable, TotalView opens the Source view focused on main in the host code, as shown in Figure 155.

Figure 155, Source view opened on GPU host code



```
Start Page x tx_hip_matmul.cpp x
135 // Just a place to set a breakpoint in the debugger
136 __syncthreads();
137 __syncthreads(); /* STOP: Csub should be fully updated */
138 }
139
140 // A function to serve as a backstop for setting breakpoints in the
141 // preceding kernel, so that they don't slip over to the next function.
142 static int breakpoint_backstop() { return 0; }
143
144 static Matrix
145 cons_Matrix (int width_, int height_)
146 {
147     Matrix A;
148     A.width = width_;
149     A.height = height_;
150     A.stride = width_;
151     A.elements = (float*) malloc(sizeof(*A.elements) * width_ * height_);
152     for (int row = 0; row < height_; row++)
153         for (int col = 0; col < width_; col++)
154             A.elements[row * width_ + col] = row * 10.0 + col;
155     return A;
156 }
157
158 static void
159 print_Matrix (Matrix A, const char *name)
160 {
161     printf("%s:\n", name);
162     for (int row = 0; row < A.height; row++)
163         for (int col = 0; col < A.width; col++)
164             printf("%f\t", A.elements[row * A.stride + col]);
165 }
```

You can debug the GPU host code using the normal TotalView commands and procedures.

Controlling Execution

Set breakpoints in GPU code before you start the process. If you start the process without setting any breakpoints, there are no prompts to set them afterward.

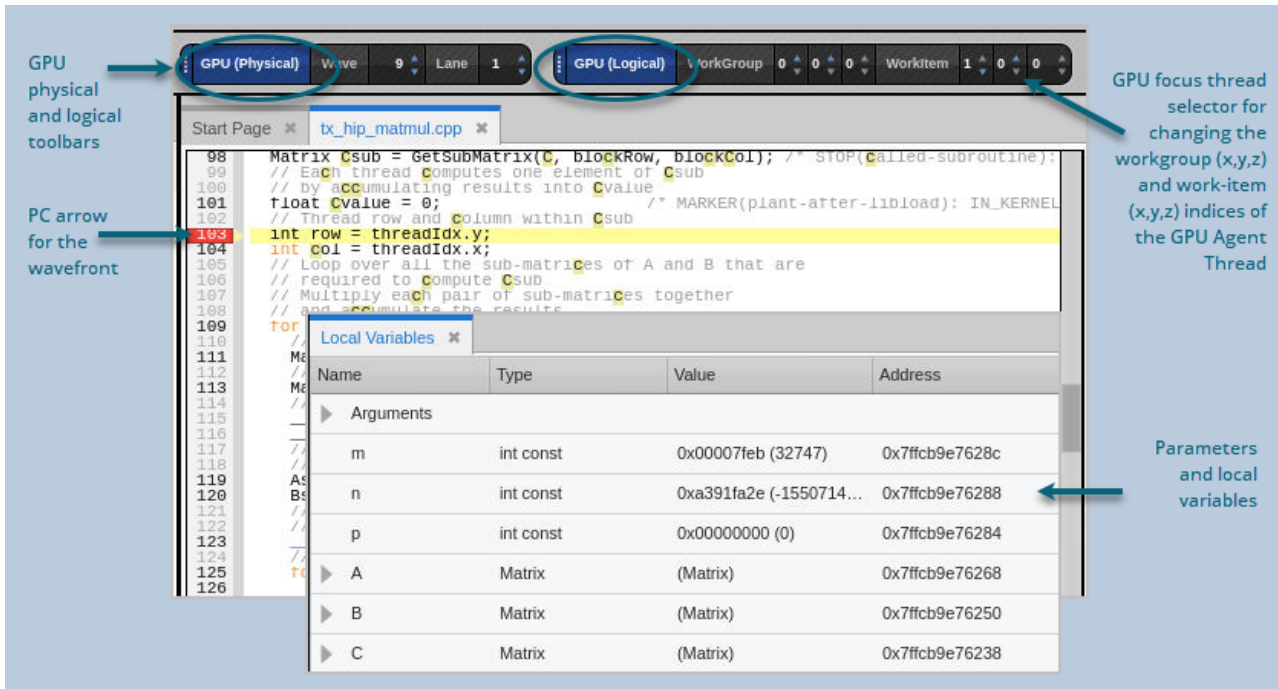
Note that breakpoints set in GPU code will *slide* to the next host (CPU) line in the source file, but once the program is running and the GPU code is loaded, TotalView recalculates the breakpoint expression and plants a breakpoint at the proper location in the GPU code. (See [Sliding Breakpoints](#).)

Note that breakpoints set in GPU source code might slide to a host source line that is executed before the GPU code is loaded; thus, the process might unexpectedly stop at that line before it stops in the GPU kernel.

Viewing GPU Threads

Once the GPU kernel starts executing, it will hit breakpoints planted in the GPU code, as shown in Figure 156.

Figure 156, ROCm thread stopped at a breakpoint, focused on GPU work-item (0,0,0)[1,0,0]



The **logical coordinates** of the GPU focus work-group and work-items are displayed in the **GPU logical toolbar** at the top of Figure 156 above. (See [GPU Toolbars](#).) The WorkGroup control shows the 3-D focus position of the work-group in the kernel dispatch. The WorkItem control shows the 3-D focus position of the work-item in the work-group.

The **GPU focus work-group and/or work-item** can be changed using the GPU focus selector in the logical toolbar. When you change the GPU focus work-group or work-item, the logical coordinates displayed also change, and the Call Stack and Source view are updated to reflect the state of the new GPU focus work-group or work-item.

The **execution location of the GPU focus work-item** is identified by the yellow PC highlighted line in the Source view. The work-items are grouped into "wavefronts" (or "waves" for short) that execute in parallel, so multiple work-items may have the same PC value. The work-items may be part of the same wave (consisting of 32 or 64 lanes that execute concurrently), or part of different waves.

The **Local Variables view** shows the parameters and local variables for the function in the selected stack frame. The variables for the selected GPU kernel code, function, or inlined function expansion are shown.

The **Call Stack** shows the stack backtrace and inlined functions:



Each stack frame in the stack backtrace represents either the PC location of GPU kernel code, or the expansion of an inlined function.

GPU Agent Thread IDs and Coordinate Spaces

Again, TotalView gives host threads a *positive* debugger thread ID and GPU agent threads a *negative* thread ID. In this example, the initial host thread in process "1" is labeled "1.1" and the GPU agent thread is labeled "1.-1".

Figure 157, ROCm GPU Thread IDs

Description	# P	# T	Members
Breakpoint	1	1	p1.-1
MatMulKernel	1	1	p1.-1
tx_hip_matmul.cpp#...	1	1	p1.-1
1.-1	1	1	p1.-1
Stopped	1	4	p1.-2, p1.1-3
<unknown address>	1	1	p1.-2
<unknown address>	1	1	p1.-2
1.-2	1	1	p1.-2
hsa_amd_image_get_in...	1	1	p1.1
<unknown line>	1	1	p1.1

Use the "GPU focus selector" on the GPU toolbar (See [GPU Toolbars](#)) to change the logical coordinates of the GPU focus work-group and/or work-item.

Figure 158, GPU Focus Selector on the GPU logical toolbar



GPU Toolbars

Two GPU toolbars display the two coordinate spaces. One is the logical coordinate space that, in AMD GPU terms, incorporates work-groups and work-items: (Bx,By,Bz)[Tx,Ty,Tz].

The other is the "physical" coordinate space that is, in hardware terms, the wavefront ID (wave ID) assigned by the AMD GPU debug API and the lane number index within the wave. Note that the wave ID is an arbitrary, unique value, assigned by the debug API, and has no relationship to any physical aspect of the GPU agent.

Any given work-item has both a position in the physical coordinate space, and a position in the 6-D logical coordinate space. These indices are shown in the two GPU toolbars. Changing the GPU focus in one coordinate space changes the other.

Figure 159, GPU logical and physical toolbars



To view a host CPU thread, select a thread with a positive thread ID in the Process and Threads view. To view a GPU agent thread, select an agent thread with a negative thread ID, then use the GPU work-group and/or work-item selector on the logical toolbar to focus on a specific GPU work-item.

There is one GPU focus per GPU agent thread, and changing the GPU focus affects all windows displaying information for that agent and all command line interface commands targeting that agent. In other words, changing the GPU focus work-group or work-item can change the GPU data that is displayed and affect other commands, such as single-stepping.

Note that in all cases, when you select a work-item, TotalView automatically updates the Source view, Call Stack, Data View, and Action Points view to match the selected work-item.

Single-Stepping GPU Code

TotalView allows you to single-step GPU code just like normal host code. The GPU focus work-item is used to guide the single-step operation in a manner similar to single-stepping a CPU thread. TotalView uses GPU-level instruction disassembly, instruction-level stepping, and breakpoint hopping to implement GPU source-level single stepping.

- Thread-width single-stepping:** When the single-stepping width is a single GPU agent thread, all the waves on the agent are allowed to execute until the GPU focus work-item reaches the source-level single-stepping goal.

- **Process or group-width single stepping:** When the single-stepping width is a process or a group, all the GPU agent threads in the process or group are allowed to execute. This technique tends to keep all the waves executing in lockstep because they normally hit the temporary breakpoints planted by the source-level single stepper. However, wave and work-item divergence are allowed.

Halting a Running Application

You can temporarily halt a running application at any time by selecting "Halt", which halts the host and GPU agent threads. This can be useful if you suspect the kernel might be hung or stuck in an infinite loop. You can resume execution at any time by selecting "Go" or by selecting one of the single-stepping buttons.

Displaying ROCm Program Elements

GPU Variable and Data Display

TotalView can display variables and data for a specific GPU work-item on a GPU agent. Changing the GPU focus for the GPU agent may also change the displayed data.

In the same way as when debugging CPU code, you can add an expression from the Local Variables view to the Data View or create new expressions in the Data View to analyze your data. The variables are contained within the lexical blocks in which they are defined.

Casting a Pointer to an Array

Casting works the same way as when debugging CPU code. See [Casting to an Array in the Data View](#).

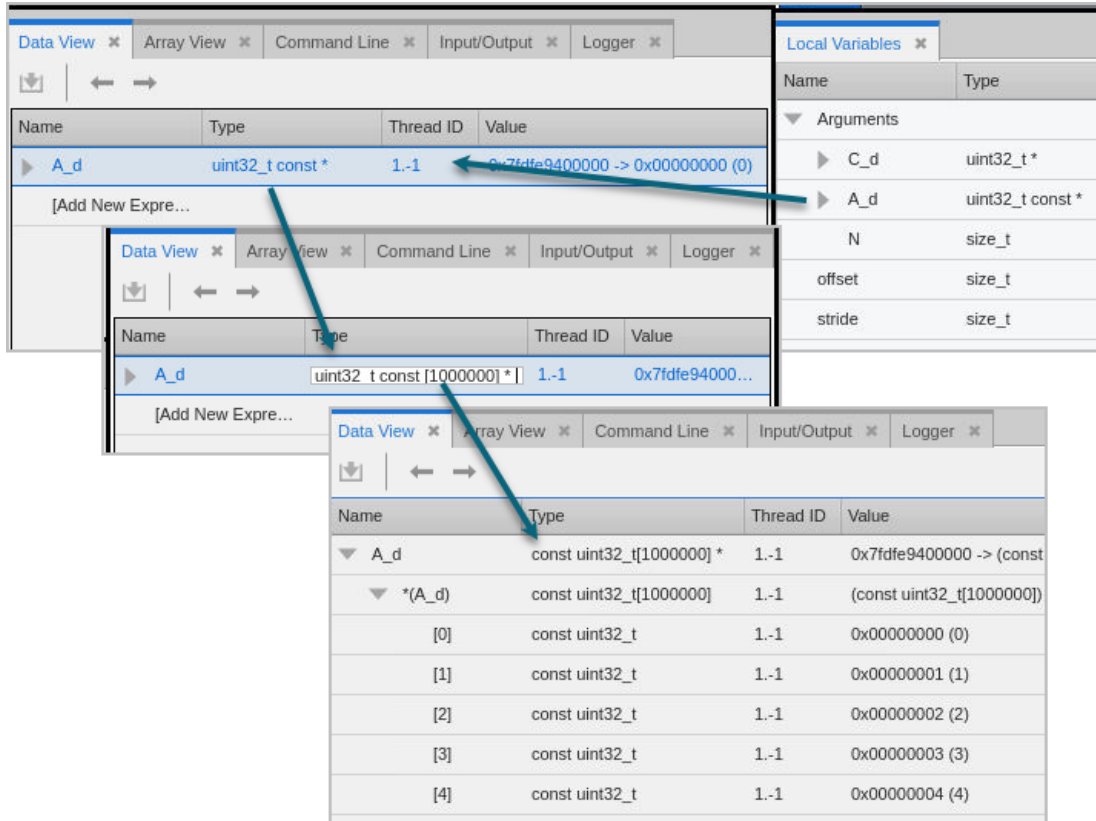
Here, let's add the pointer **A_d** from the Local Variables view to the Data View, then edit the Type field to cast it to a pointer to an array of elements.

Below, **A_d** is defined in the program as a `uint32_t const *`, which is a pointer to a `const uint32_t`. However, we know from the value of **N**, **A_d** is actually a pointer to an array of 1,000,000 `uint32_t` elements.

To cast the pointer, edit the Type field to read `uint32_t const [1000000] *`, which, reading from right to left, is "a pointer to an array of 1,000,000 `const uint32_t`".

You can then dereference the pointer by expanding the tree control to see the array, and then expanding the array to show the elements.

Figure 160, Dereferencing a pointer in the Data View



See the Values of Built-in Variables

You can add ROCm built-in variables to the Data View from the Source pane to view information about the grid or the thread index, for example.

Figure 161, Adding built-in variables to the Data View

```

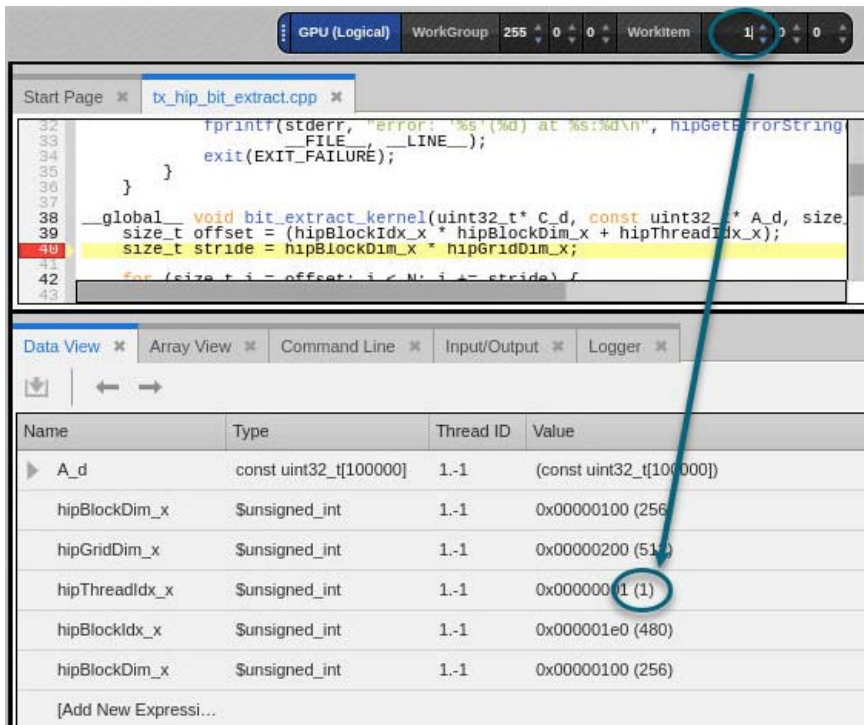
36     }
37
38     __global__ void bit_extract_kernel(uint32_t* C_d, const uint32_t* A_d, size_t
39         size_t offset = (hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x);
40         size_t stride = hipBlockDim_x * hipGridDim_x;
41
42         for (size_t i = offset; i < N; i += stride) {
43             #ifdef HIP_PLATFORM_HCC
44                 C_d[i] = bit_extract_u32(A_d[i] & 0x00000000); /* STOR/in kernel: */
45             }
    
```

Name	Type	Thread ID	Value
▶ A_d	const uint32_t[100000]	1-1	(const uint32_t[100000])
hipBlockDim_x	\$unsigned_int	1-1	0x00000100 (256)
hipGridDim_x	\$unsigned_int	1-1	0x00000200 (512)
hipThreadIdx_x	\$unsigned_int	1-1	0x00000000 (0)
hipBlockIdx_x	\$unsigned_int	1-1	0x000001e0 (480)
hipBlockDim_x	\$unsigned_int	1-1	0x00000100 (256)
[Add New Expressi...			

Change the Focus and the Data View Updates

Note that the built-in variables, such as `hipThreadIdx_x`, reflect the GPU focus. If you change the focus in the GPU toolbar, the value of these variables may change.

Figure 162, Changing the work-item focus is reflected in the Data View



RELATED TOPICS

Using the Data View

The Data View

ROCm's built-in variables

ROCm Built-In Runtime Variables

ROCm Built-In Runtime Variables

TotalView allows access to the ROCm HIP built-in runtime variables, which are handled by TotalView like any other variables, except that you cannot change their values.

The supported ROCm HIP built-in runtime variables are as follows:

- `struct dim3_32 threadIdx;`
- `struct dim3_32 blockIdx;`

- `struct dim3_32 blockDim;`
- `struct dim3_32 gridDim; // Grid sizes in work-group units`
- `struct dim3_32 gridDimWorkItems; // Grid sizes in work-item units`
- `int warpSize;`
- `unsigned int hipThreadId_x, hipThreadId_y, hipThreadId_z;`
- `unsigned int hipBlockIdx_x, hipBlockIdx_y, hipBlockIdx_z;`
- `unsigned int hipBlockDim_x, hipBlockDim_y, hipBlockDim_z;`
- `unsigned int hipGridDim_x, hipGridDim_y, hipGridDim_z;`

The types of the built-in variables are defined as follows:

```
struct dim3_32 { unsigned int x, y, z; };
```

GPU Error Reporting

The AMD debugger's exception system is not yet fully developed, so this release doesn't yet support this feature, although your GPU code may throw exceptions.

AMD ROCm Problems and Limitations

- Hangs or Initialization Failures
- AMD GPU Debugging and ReplayEngine

AMD TotalView sits directly on top of the ROCm debugging environment provided by AMD, which is still evolving and maturing. This environment contains certain problems and limitations, discussed in this chapter.

Hangs or Initialization Failures

When starting an AMD GPU debugging session, you may encounter hangs in the debugger or target application, initialization failures, or failure to launch a kernel. Use the following checklist to diagnose the problem:

Serialized Access

Older AMD GPU models (prior to MI200s) support at most one AMD GPU debugging session active per node at a time. A node cannot be shared for debugging ROCm code simultaneously by multiple user sessions, or multiple sessions by the same user. Use `ps` or other system utilities to determine if your session is conflicting with another debugging session.

Orphaned Processes

Occasionally, a debugging session might accidentally orphan a process. Orphaned processes might go compute bound or prevent you or other users from starting a debugging session. You may need to manually kill orphaned ROCm processes in order to start your AMD GPU debugging session or stop a compute-bound process. Use system tools such as `ps` or `top` to find the processes and kill them using the shell `kill` command. If the process was orphaned by another user, that user will own the processes and you may not be able to kill them. In this case, ask the user or system administrator to kill them for you.

AMD GPU Debugging and ReplayEngine

Enabling ReplayEngine on processes that are using AMD GPUs is not supported.

Sample HIP Program

```

#include "hip/hip_runtime.h"
/*
 * A "hipified" version of the NVIDIA CUDA matrix multiple example
 */
#include <hip/hip_runtime.h>
#include <stdio.h>

// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width; /* number of columns */
    int height; /* number of rows */
    int stride;
    float* elements;
} Matrix;

// Get a matrix element
/*__forceinline__*/ __device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

// Set a matrix element
__forceinline__ __device__ void SetElement(Matrix A, int row, int col, float value)
{
    A.elements[row * A.stride + col] = value;
}

// Thread block size
#define BLOCK_SIZE 2

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
/*__forceinline__*/ __device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width = BLOCK_SIZE;
    Asub.height = BLOCK_SIZE;
    Asub.stride = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
        + BLOCK_SIZE * col];
    return Asub;
}

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory

```

```

Matrix d_A;
d_A.width = d_A.stride = A.width; d_A.height = A.height;
size_t size = A.width * A.height * sizeof(float);
hipMalloc((void**)&d_A.elements, size);
hipMemcpy(d_A.elements, A.elements, size,
          hipMemcpyHostToDevice);
Matrix d_B;
d_B.width = d_B.stride = B.width; d_B.height = B.height;
size = B.width * B.height * sizeof(float);
hipMalloc((void**)&d_B.elements, size);
hipMemcpy(d_B.elements, B.elements, size,
          hipMemcpyHostToDevice);
// Allocate C in device memory
Matrix d_C;
d_C.width = d_C.stride = C.width; d_C.height = C.height;
size = C.width * C.height * sizeof(float);
hipMalloc((void**)&d_C.elements, size);
// Invoke kernel
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
hipLaunchKernelGGL(MatMulKernel, dim3(dimGrid), dim3(dimBlock), 0, 0, d_A, d_B,
d_C);
// Read C from device memory
hipMemcpy(C.elements, d_C.elements, size,
          hipMemcpyDeviceToHost);
// Free device memory
hipFree(d_A.elements);
hipFree(d_B.elements);
hipFree(d_C.elements);
}

// Matrix multiplication kernel called by MatrixMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;
    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol); /* STOP(called-subroutine): */
    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0; /* MARKER(plant-after-libload): IN_KERNEL_LINE */
    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;
    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);
        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);

```

```

    Bs[row][col] = GetElement(Bsub, row, col);
    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    __syncthreads();
    // Multiply Asub and Bsub together
    for (int e = 0; e < BLOCK_SIZE; ++e)
        Cvalue += As[row][e] * Bs[e][col];
    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
}
// Write Csub to device memory
// Each thread writes one element
SetElement(Csub, row, col, Cvalue);/* STOP(inlined-subroutine): */
// Just a place to set a breakpoint in the debugger
__syncthreads();
__syncthreads();/* STOP: Csub should be fully updated */
}

// A function to serve as a backstop for setting breakpoints in the
// preceding kernel, so that they don't sloop over to the next function.
static int breakpoint_backstop() { return 0; }

static Matrix
cons_Matrix (int width_, int height_)
{
    Matrix A;
    A.width = width_;
    A.height = height_;
    A.stride = width_;
    A.elements = (float*) malloc(sizeof(*A.elements) * width_ * height_);
    for (int row = 0; row < height_; row++)
        for (int col = 0; col < width_; col++)
            A.elements[row * width_ + col] = row * 10.0 + col;
    return A;
}

static void
print_Matrix (Matrix A, const char *name)
{
    printf("%s:\n", name);
    for (int row = 0; row < A.height; row++)
        for (int col = 0; col < A.width; col++)
            printf ("[%5d][%5d] %f\n", row, col, A.elements[row * A.stride + col]);
}

// Multiply an m*n matrix with an n*p matrix results in an m*p matrix.
// Usage: tx_cuda_matmul [ m [ n [ p ] ] ]
// m, n, and p default to 1, and are multiplied by BLOCK_SIZE.
int main(int argc, char **argv)
{
    // hipSetDevice(0);
    const int m = BLOCK_SIZE * (argc > 1 ? atoi(argv[1]) : 1);
    const int n = BLOCK_SIZE * (argc > 2 ? atoi(argv[2]) : 1);
    const int p = BLOCK_SIZE * (argc > 3 ? atoi(argv[3]) : 1);
    Matrix A = cons_Matrix(m, n);/* MARKER(plant-after-libload): IN_MAIN_LINE */
    Matrix B = cons_Matrix(n, p);
    Matrix C = cons_Matrix(m, p);
    MatMul(A, B, C);
}

```

```
print_Matrix(A, "A");  
print_Matrix(B, "B");  
print_Matrix(C, "C");  
return breakpoint_backstop();  
}
```

PART VI Memory Debugging

- About TotalView Memory Debugging
- Running a Memory Debugging Session
- Memory Scripting
- Preparing Programs for Memory Debugging
-

About TotalView Memory Debugging

- Debugging Memory in TotalView
- About Program Memory
- How TotalView Intercepts Memory Data
- Your Program's Data

Debugging Memory in TotalView

NOTE: TotalView has long included the memory debugging features of TotalView in its Classic UI. The new UI now includes a range of memory-related features including leak detection, heap and event reports, and the ability to identify dangling pointers, with additional functionality added in each release. For the full-featured functionality of TotalView, see *Debugging Memory Problems with TotalView* in the Classic UI documentation in `<installdir>\doc\pdf`.

TotalView helps you locate many of your program's memory problems. In most cases, TotalView creates and records a backtrace for memory blocks, so you can immediately know where your program allocated or freed the memory block.

For example, you can:

- **Detect Leaks**

TotalView can display reports of your program's memory leaks, i.e., memory blocks that are allocated and are no longer referenced, in which case the program can no longer access the memory block, and that memory is unavailable for any other use. See [Memory Leaks](#).

- **Analyze Heap Allocations**

TotalView can locate heap allocations and display related information. Heap allocations are derived from monitoring program requests for memory (**malloc** or **new**). See [Memory Heap Reports](#).

- **Report Memory Events**

A number of memory events can notify you and stop execution if you choose. For example, if a block was previously deallocated, TotalView can stop execution and report a "double free" error. See [Memory Event Reports](#).

- **Memory Block Notifications**

You can tag specific memory blocks to have TotalView notify you when memory is allocated or freed. See [Memory Block Notification](#).

- **Memory Debugging Options**

Refine your debugging session using multiple options to help find memory problems, including painting memory, hoarding memory blocks, and guarding allocated memory. See [Memory Debugging Options](#).

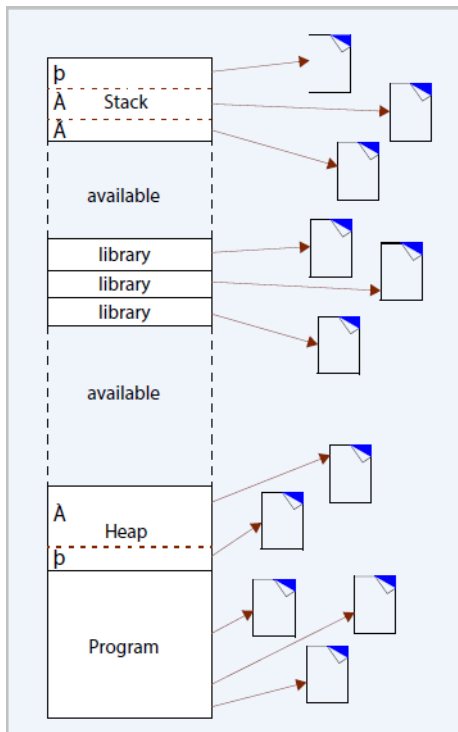
About Program Memory

When you run a program, your operating system loads the program into memory and defines an address space in which the program can operate. For example, if your program is executing in a 32-bit computer, the address space is approximately 4 gigabytes.

NOTE: This discussion is generally relevant to most computer architectures. For information specific to your system, check your vendor documentation.

An operating system does not actually allocate the memory in this address space. Instead, operating systems *memory map* this space, which means that the operating system relates the theoretical address space your program could use with what it actually will be using. Typically, operating systems divide memory into pages. When a program begins executing, the operating system creates a map that correlates the executing program with the pages that contain the program's information. [Figure 163](#) shows regions of a program with arrows pointing to the memory pages that contain different portions of your program, as well as a stack containing three stack frames, each mapped to its own page.

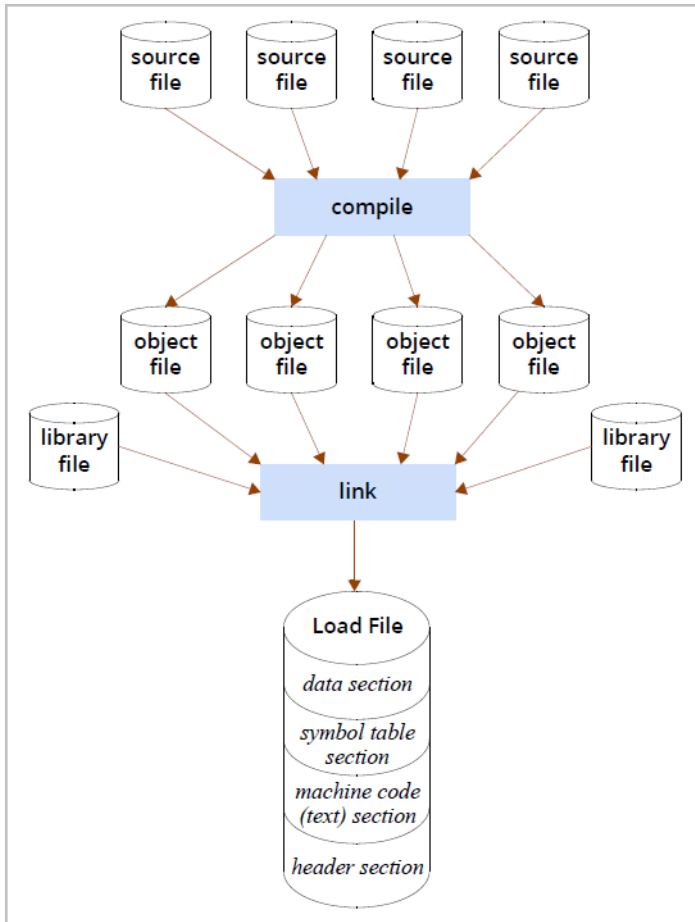
Figure 163, Mapping Program Pages



Similarly, the heap shows two allocations, each mapped to its own page. (This illustration vastly simplifies actual memory mapping, since a page can have many stack frames and many heap allocations.)

Figure 164 shows the compiling and linking dependencies for a program whose source code resides in four files.

Figure 164, Compiling Programs



Compiling these files creates object files, which a linker merges, along with any external libraries, into a load file. This load file is the executable program stored on your computer's file system.

When the linker creates the load file, it combines the information contained in each of the object files into one unit. The load file at the bottom of Figure 164 also details this file's contents, as this file contains a number of sections and additional information. For example:

- **Data section**—contains static variables and variables initialized outside of a function, for example:

```
int my_var1 = 10;
void main ()
```

```

{
    static int my_var2 = 1;
    int my_var3;
    my_var3 = my_var1 + my_var2;
    printf("here's what I've got: %i\n", my_var3);
}

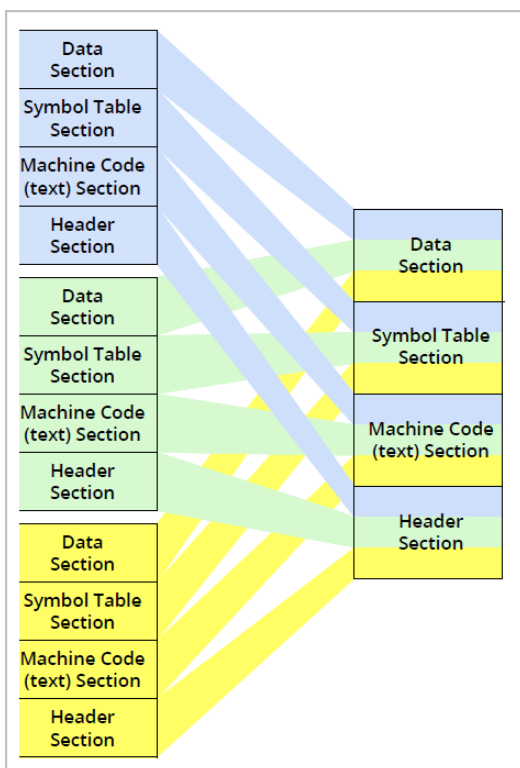
```

The data section contains the **my_var1** and **my_var2** variables. In contrast, the memory for the **my_var3** variable is dynamically and automatically allocated and deallocated within the stack by your program's runtime system.

- **Symbol table section**—contains addresses to the locations of routines and variables.
- **Machine code section**—contains an intermediate binary representation of your program. (It is intermediate because the linker has not yet resolved the addresses.)
- **Header section**—contains information about the size and location of information in all other sections of the object file.

The linker creates one file from all these sections that can be loaded into memory, [Figure 165](#).

Figure 165, Linking a Program



TotalView can provide information about these sections and generate a leak detection report, [Figure 166](#).

Figure 166, Memory Debugging Leak Report

Process	Bytes	Count	Begin Address	End Address
Process 1 (22583): tx_array_statistics	432	3		
tx_array_statistics.cxx	432	3		
main	432	3		
Line 36	400	1		
Block 3.1	400	1	0x01646090	0x0164621f
Line 27	16	1		
Line 26	16	1		

ID	Function	Line #	Source Information
3	malloc	172	malloc_wrappers_dlopen.c
	_Znw		libstdc++.so.6
	_Znam		libstdc++.so.6
	main	36	tx_array_statistics.cxx
	__libc_start_main		libc.so.6

For information, see [Memory Leak Detection](#).

How TotalView Intercepts Memory Data

TotalView intercepts calls made by your program to heap library functions that allocate and deallocate memory by using the **malloc()** and **free()** functions and related functions such as **calloc()** and **realloc()**. The technique it uses is called *interposition*. TotalView's interposition technology uses an agent routine to intercept calls to functions in this library. This agent routine is sometimes called the **Heap Interposition Agent (HIA)**.

You can use TotalView with any allocation and deallocation library that uses such functions as **malloc()** and **free()**. Typically, this library is called *the malloc library*. For example, the C++ **new** operator is almost always built on top of the **malloc()** function. If it is, TotalView can track it. Similarly, if your Fortran implementation use **malloc()** and **free()** functions to manage memory, TotalView can track Fortran heap memory use.

You can interpose the agent in two ways:

- TotalView can preload the agent. Preloading means that the loader places an object before the object listed in the application's loader table.

When a routine references a symbol in another routine, the linker searches for that symbol's definition. Because the agent's routine is the first object in the table, your program invokes the agent's routine instead of the routine that was initially linked in.

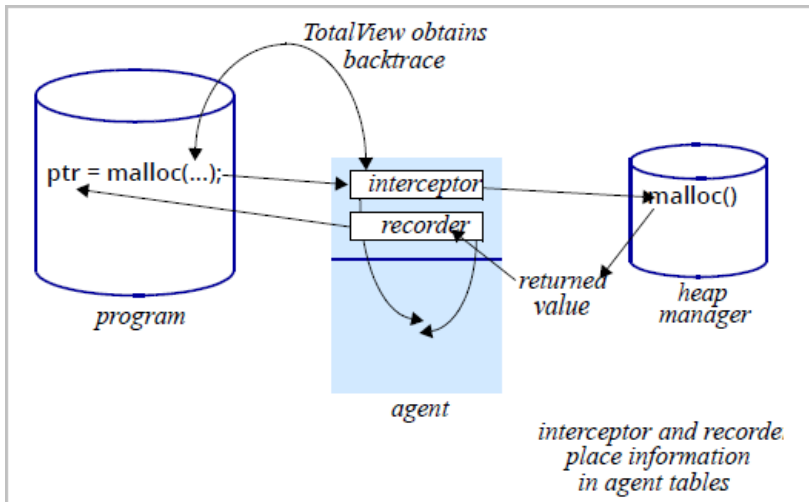
On Linux and Sun, TotalView sets an environment variable that contains the pathname of the agent's shared library. For more information, see [Using env to Insert the HIA](#).

- If TotalView cannot preload the agent, you will need to explicitly link it into your program. For details, see [Linking Your Application with the HIA](#).

If your program attaches to an already running program, you must explicitly link this other program with the agent.

After the agent intercepts a call, it calls the original function. This means that you can use TotalView with most memory allocators. [Figure 167](#) shows how the agent interacts with your program and the heap library.

Figure 167, Interposition



Because TotalView uses interposition, memory debugging can be considered non-invasive. That is, TotalView doesn't rewrite or augment your program's code, and you don't have to do anything in your program. Because the agent lives in the user space, it will add a small amount of overhead to the program's behavior, but it should not be significant.

Your Program's Data

Your program's data resides in the following places:

- [The Data Section](#)
- [The Stack](#)
- [The Heap](#)

The Data Section

Your program uses the data section for storing static and global variables. Memory in this section is permanently allocated, and the operating system sets its size when it loads your program. Variables in this section exist for the entire time that your program executes.

Errors can occur if your program tries to manage this section's memory. For example, you cannot free memory allocated to variables in the data section. In general, data section errors are usually related to misunderstandings regarding the program's inability to manage data section memory.

The Stack

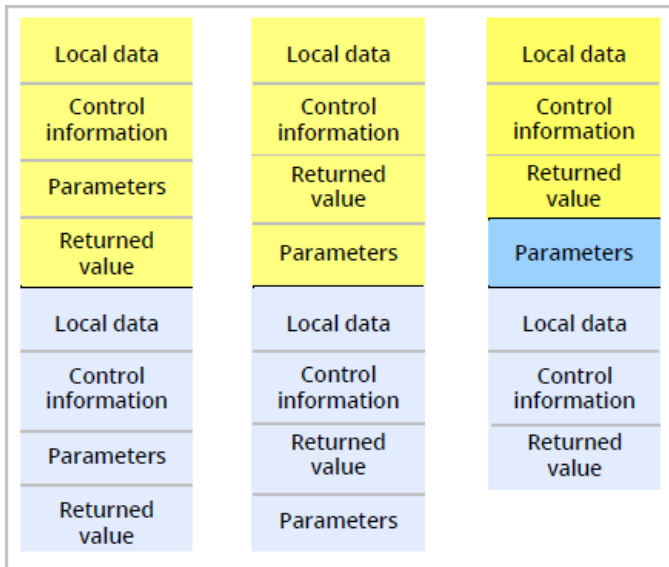
Memory in the stack section is dynamically managed by your program or operating system's memory manager. Consequently, your program cannot allocate memory in the stack or deallocate memory in it.

NOTE: "Deallocates" means that your program reports to a memory manager that it is no longer using this memory. The next time your program calls a routine, the new stack frame overwrites the memory previously used by other routines. In almost all cases, deallocated memory, whether on the stack or the heap, just hangs around in its pre-deallocation state until it gets reassigned.

The stack differs from the data section in that your program implicitly manages its space. What's in it one minute might not be there a minute later. Your program's runtime environment allocates memory for stack frames as your program calls routines, and deallocates these frames when execution exits from the routine.

A stack frame contains control information, data storage, and space for passed-in arguments (parameters) and the returned value (and much more). [Figure 168](#) shows three ways in which a compiler can arrange stack frame information.

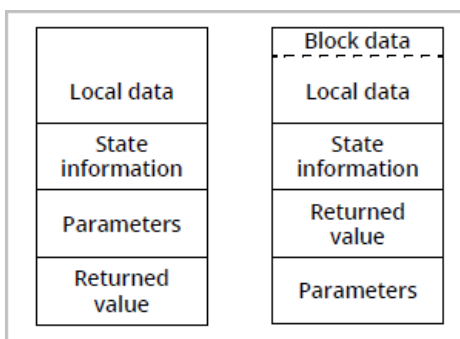
Figure 168, Placing Parameters



In this figure, the left and center stack frames have different positions for the parameters and returned value. The stack frame on the right is a little more complicated. In this version, the parameters reside in a stack memory area that doesn't belong to either stack frame.

If a stack frame contains local (sometimes called *automatic*) variables, where is this memory placed? If the routine has blocks in which memory is allocated, where on the stack is this memory for these additional variables placed? Although there are many variations, Figure 169 shows two of the more common ways to allocate memory.

Figure 169, Local Data in a Stack Frame

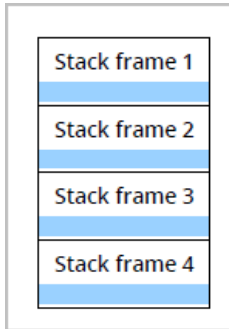


The blocks on the left show a data block allocated within a stack frame on a system that ignores your routine's block structure. The compiler figures how much memory your routine needs, and then allocates enough memory for all of a routine's automatic variables. These kinds of systems minimize the time necessary to allocate memory.

Other systems dynamically allocate the memory required within a routine as the block is entered, and then deallocate it as execution leaves the block. (The blocks on the right show this.) These systems minimize a routine's size.

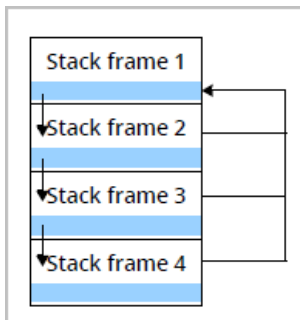
As your program executes routines, routines call other routines, placing additional routines on the stack. [Figure 170](#) shows four stack frames. The shaded areas represent local data.

Figure 170, Four Stack Frames



What happens when a program passes a pointer to memory in a stack frame to lower frames? [Figure 171](#) shows a program passing a pointer to memory in stack frame 1 down to lower stack frames.

Figure 171, Passing Pointers



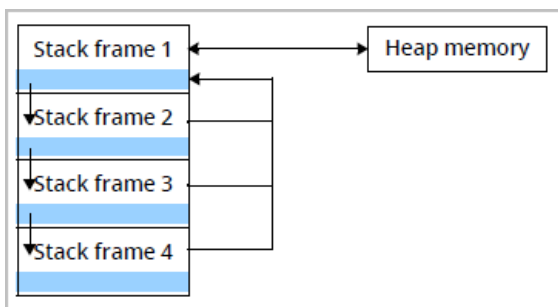
Here, the arrows on the left represent an address contained within a pointer, an address that is passed down the stack. The lines and arrow on the right indicate the place to which the pointer is pointing. A pointer to memory in frame 1 is passed to frame 2, which passes the pointer to frame 3, and then to frame 4. In all frames, the pointer points to a memory location in frame 1. Stated in another way, the pointers in frames 2, 3, and 4 point to memory in another stack frame. This is the most efficient way for your program to pass data from one routine to another, since your program passes the pointer instead of the actual data. Using the pointer, the program can both access and alter the information that the pointer is pointing to.

NOTE: We know that data can be passed “by-value” (which means copying it) or “by-reference” (which means passing a pointer), but it’s important to remember that something is always copied, because “pass by reference” copies a pointer to the data.

Because the program’s runtime system owns stack memory, you cannot free it. Instead, your program’s runtime system frees it when it pops a frame from the stack.

One of the reasons that memory problems occur is that it may not be clear which component owns a variable’s memory. For example, [Figure 172](#) shows a routine in frame 1 that has allocated memory in the heap, and which passes a pointer to that memory to other stack frames.

Figure 172, Allocating a Memory Block

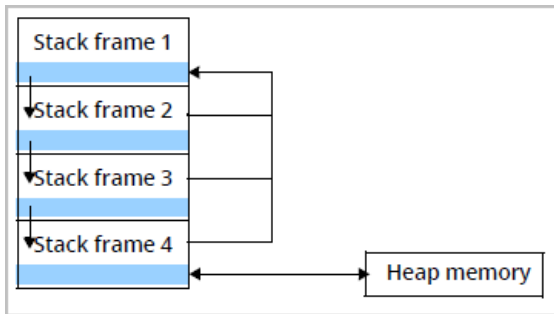


If the routine executing in frame 4 frees this memory, all pointers to that memory are dangling; that is, they point to deallocated memory. If the program’s memory manager reallocates this heap memory block, the data accessible by all the pointers is both invalid and wrong. Note that if the memory manager doesn’t immediately reuse the block, the data accessed through the pointers is still correct.

The timing of the reallocation and reuse of a block by another allocation request means there is no guarantee that the data is correct when the program accesses the block, and there is never a pattern to when the block’s data changes. Consequently, the problem occurs only intermittently, which makes it nearly impossible to locate. Worse, development systems usually are not as memory stressed as production systems, so the problem may occur only in the production environment.

Another common problem occurs when you allocate memory and assign its location to an automatic variable, shown in [Figure 173](#).

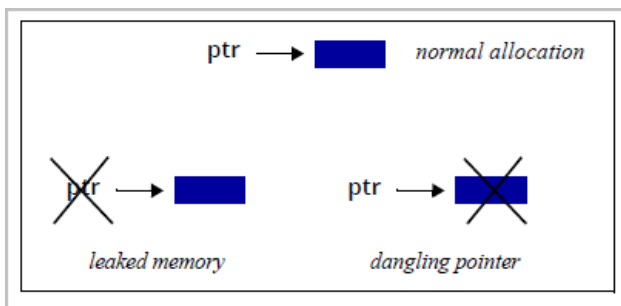
Figure 173, Allocating a Block from a Stack Frame



If frame 4 returns control to frame 3 without deallocating the heap memory it created, this memory is no longer accessible, and your program can no longer use this memory block. It has *leaked* this memory block.

NOTE: If you have trouble remembering the difference between a leak and a dangling pointer, [Figure 174](#) may help. In both cases, your program allocates heap memory, and the address of this memory block is assigned to a pointer. A leak occurs when the pointer gets deleted, leaving a block with no reference. In contrast, a dangling pointer occurs when the memory block is deallocated, leaving a pointer that points to deallocated memory. Both are shown here.

Figure 174, Leaks and Dangling Pointers



TotalView can report your program’s leaks. For information on detecting leaks, see [Memory Leak Detection](#).

The Heap

The *heap* is an area of memory that your program uses when it wants to dynamically allocate space for data. While using the heap gives you a considerable amount of flexibility, your program must manage this resource. That is, the program must explicitly allocate and deallocate this space. In contrast, the program does not allocate or deallocate memory in other areas.

Because allocations and deallocations are intimately linked with your program's algorithms and, in some cases, the way the program uses this memory is implicit rather than explicit, problems associated with the heap are the hardest to find.

Finding Heap Allocation Problems

Memory allocation problems are seldom due to allocation requests. Because an operating system's virtual memory space is large, allocation requests usually succeed. Problems most often occur if you are either using too much memory or leaking it. Although problems are rare, you should always check the value returned from calls to allocation functions such as **malloc()**, **calloc()**, and **realloc()**. Similarly, you should always check whether the C++ **new** operator returns a null pointer. (Newer C++ compilers throw a **bad_alloc** exception.) If your compiler supports the **new_handler** operator, you can throw your own exception.

Finding Heap Deallocation Problems

TotalView can let you know when your program encounters a problem in deallocating memory. Some of the problems it can identify are:

- **free() not allocated:** An application calls the **free()** function by using an address that is not in a block allocated in the heap.
- **realloc() not allocated:** An application calls the **realloc()** function by using an address that is not in a block allocated in the heap.
- **Address not at start of block:** A **free()** or **realloc()** function receives a heap address that is not at the start of a previously allocated block.

If a library routine uses the program's memory manager (that is, it is using the heap API) and a problem occurs, TotalView still locates the problem. For example, the **strdup()** string library function calls the **malloc()** function to create memory for a duplicated string. Since the **strdup()** function is calling the **malloc()** function, TotalView can track this memory.

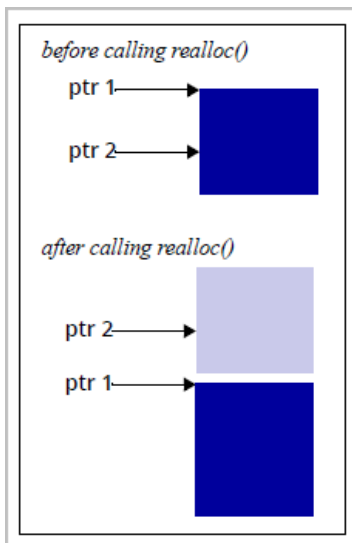
TotalView can stop execution just before your program misuses a heap API operation, which allows you to see what the problem is before it actually occurs. (See [How TotalView Intercepts Memory Data](#).)

realloc() Problems

The **realloc()** function can either extend a current memory block, or create a new block and free the old. When it creates a new block, it can create problems. Although you can check to see which action occurred, you need to code **realloc()** usage defensively so that problems do not occur. Specifically, you must change every pointer pointing to the memory block that was reallocated so that it points to the new one. Also, if the pointer doesn't point to the beginning of the block, you need to take some corrective action.

In [Figure 175](#), two pointers are pointing to a block. After the **realloc()** function executes, **ptr1** points to the new block. However, **ptr2** still points to the original block, a block that a program deallocated and returned to the heap manager.

Figure 175, realloc() Problem



Memory Leaks

A memory "leak" describes a block of memory that a program allocates that is no longer referenced. For example, when your program allocates memory, it assigns the block's location to a pointer. A leak can occur in one of the following cases:

- You assign a different value to that pointer.
- The pointer was a local variable and execution exited from the block.

If your program leaks a lot of memory, it can run out of memory. Even if it doesn't run out of memory, your program's memory footprint becomes larger. This increases the amount of paging that occurs as your program executes, making your program run more slowly.

Here are some of the circumstances in which memory leaks occur:

- **Orphaned ownership**—Your program creates memory but does not preserve the address so that it can deallocate it at a later time.

Consider this example:

```
char *str;
for( i = 1; i <= 10; i++ )
{
    str = (char *)malloc(10*i);
}
free( str );
```

In the loop, your program allocates a block of memory and assigns its address to **str**. However, each loop iteration overwrites the address of the previously created block. Because the address of the previously allocated block is lost, its memory can never be made available to your program.

- **Concealed allocation**—Creating a memory block is separate from using it.

Because all programs rely on libraries in some fashion, programmers are responsible for allocating and managing memory. As an example, contrast the **strcpy()** and **strdup()** functions. Both do the same thing—they copy a string. However, the **strdup()** function uses the **malloc()** function to create the memory it needs, while the **strcpy()** function uses a buffer that your program creates.

In many cases, your program receives a handle from a library. This handle identifies a memory block that a library allocated. When you pass the handle back to the library, it knows which memory block contains the data you want to use or manipulate. There may be a considerable amount of memory associated with the handle, and deleting the handle without telling the library to deallocate the memory associated with the handle leaks memory.

- **Changes in custody**—The routine creating a memory block is not the routine that frees it. (This is related to concealed allocation.)

For example, routine 2 asks routine 1 to create a memory block. At a later time, routine 2 passes a reference to this memory to routine 3. Which of these blocks is responsible for freeing the block?

This type of problem is more difficult than other types of problems in that it is not clear when your program no longer needs the data. The only thing that seems to work consistently is reference counting. In other words, when routine 2 gets a memory block, it increments a counter. When it passes a pointer to routine 3, routine 3 also increments the counter. When routine 2 stops executing, it decrements the counter. If it is zero, the executing routine frees the memory. If it isn't zero, another routine frees it at another time.

- **Underwritten destructors**—When a C++ object creates memory, it must have a destructor that frees it. No exceptions. This doesn't mean that a block of memory cannot be allocated and used as a general buffer. It just means that when an object is destroyed, it needs to completely clean up after itself; that is, the program's destructor must completely clean up its allocated memory.

Running a Memory Debugging Session

- Starting Memory Debugging in TotalView
- Memory Leak Detection
- Memory Heap Reports
- Corrupt Guard Block Reports
- Memory Event Reports
- Memory Block Notification
- Memory Debugging Options
- Dangling Pointer Problems

Starting Memory Debugging in TotalView

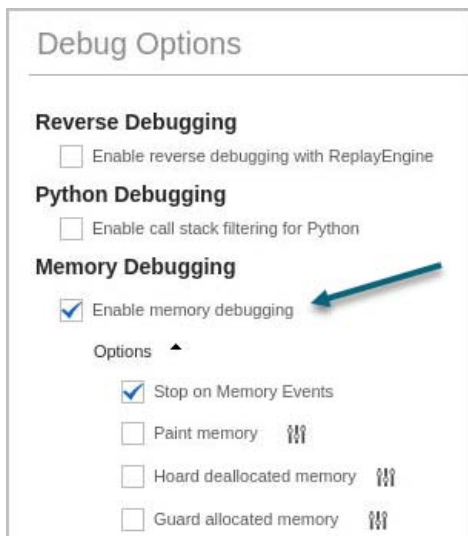
On most architectures, there is little you need to do to prepare your program for memory debugging. In most cases, just compile your program using the **-g** command-line option. In some cases, you may need to link your program with the TotalView's Heap Interposition Agent (**HIA**). (See [How TotalView Intercepts Memory Data](#) and [Linking Your Application with the HIA](#) for more information.)

NOTE: TotalView must be able to preload your program with its **HIA**. In many cases, it does this automatically. However, it cannot preload the HIA for applications that run on IBM RS/6000 platforms. For more information, see [Linking Your Application with the HIA](#).

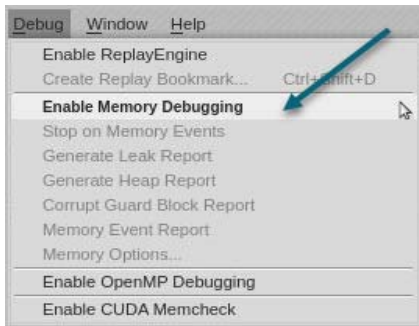
For more general information on preparing programs to debug memory issues, including platform-specific details, see [Preparing Programs for Memory Debugging](#).

To start debugging memory in TotalView:

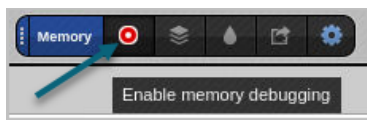
1. **Start TotalView** and load a debugging session in the normal way.
See [Starting TotalView and Creating a Debugging Session](#) for detail.
2. **Enable Memory Debugging**, in these ways:
 - Before starting your program, check “Enable memory debugging” in the Debug Options section of the Session Editor.



- From the Debug menu, check Enable Memory Debugging.



- From the memory toolbar, by selecting the Enable memory debugging icon:



Whenever your program is stopped—which happens when you halt the program, when a memory problem occurs, or just before the program exits—TotalView can create a report that describes leaks.

Memory Leak Detection

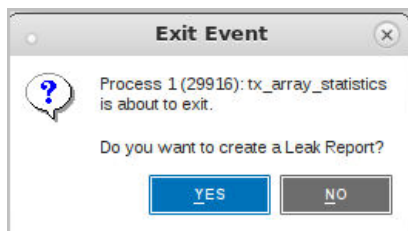
TotalView can locate your program's memory leaks and display related information. For an overview of memory leaks in general, see [Memory Leaks](#).

Leaks are reported in the **Leak Report** view. To generate a leak report, either:

- **Respond to the Leak Report prompt** before the program exits:

Run the program and then halt it when you want to look at memory problems. (Allow your program to run for a while before stopping execution to give it enough time to create leaks.)

If your program runs to the end without stopping for a breakpoint or other reason, and memory debugging is enabled, a prompt displays before the program exits, asking if you want to create a link report.



Clicking **Yes** generates the report.

or

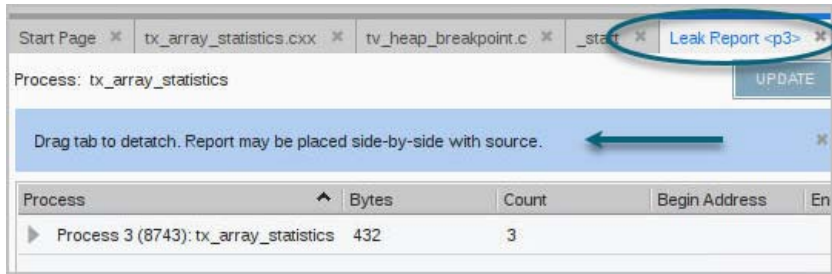
- **Use the Debug menu**

Select Debug > Generate Leak Report.



At launch, the Leak Report displays in the UI's central area, [Figure 176](#).

Figure 176, Leak Report, initial view



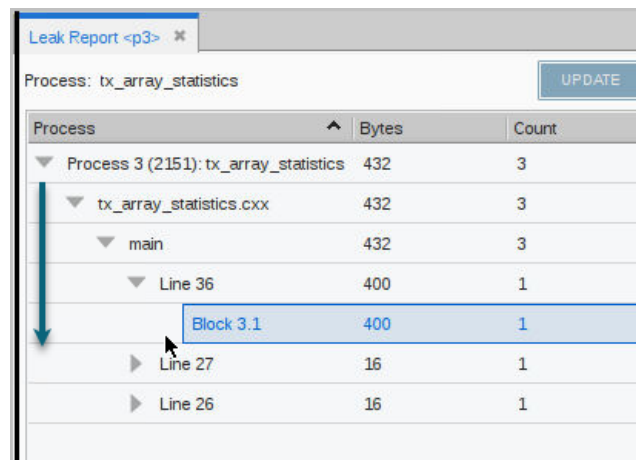
For the best viewing, detach the report and dock it beside any open source window to view the code and the report side-by-side, as suggested when it initially displays. This enables you to examine source level details as you explore the report's leak data.

Using the Leak Report

The Leak Report is based on the process in focus and includes two panes: at the top are the program's files; at the bottom is the Backtrace pane. All entries in the Program pane identify memory leaks in the program. To locate the problems:

1. Drill down to identify the block in which each leak resides, [Figure 177](#).

Figure 177, Leak Report: the Program Pane

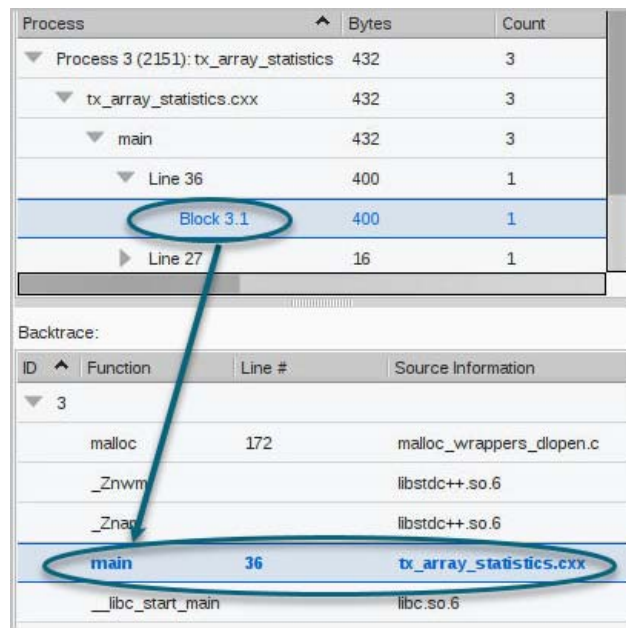


Note that:

- This example program has a process that contains a single file with three memory leaks (identified by the **Count** column), at lines 36, 27, and 26.

- The **Bytes** column displays the number of bytes that have been lost to leaks.
 - The **Update** button is grayed-out because the report is up to date. This will become active if the program continues to run past this point.
2. Select the block to analyze, which populates the Backtrace pane with the line of code where the memory was initially allocated.

Figure 178, Leak Report, Backtrace pane



The Backtrace is not the active backtrace (which is that displayed in the Call Stack view); rather, it is the backtrace that existed when your program made the heap allocation request. It identifies the block number, the function name, line number, and filename.

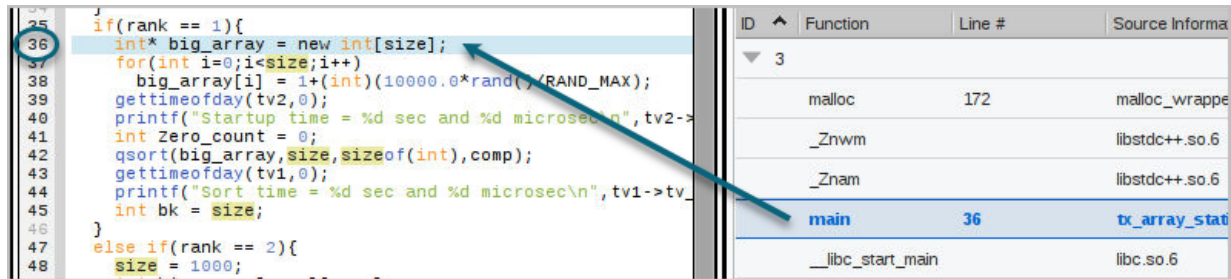
Selecting a different entry in the top pane refocuses the Backtrace pane.

You can also navigate directly to the source code location of the leak from the Program pane by selecting the Line entry:



3. Select the function in the Backtrace pane to highlight its location in the source code.

Figure 179, Leak Report, source pane



TotalView uses a conservative approach to finding memory leaks, searching roots from the stack, registers, and data sections of the process for references into the heap. Although leaks will not be falsely reported, some leaks may be missed. If you are within a method that has leaks, you may need to step out of the method for the leak to be reported. In addition, leak detection may be sensitive to the compiler used to build the program.

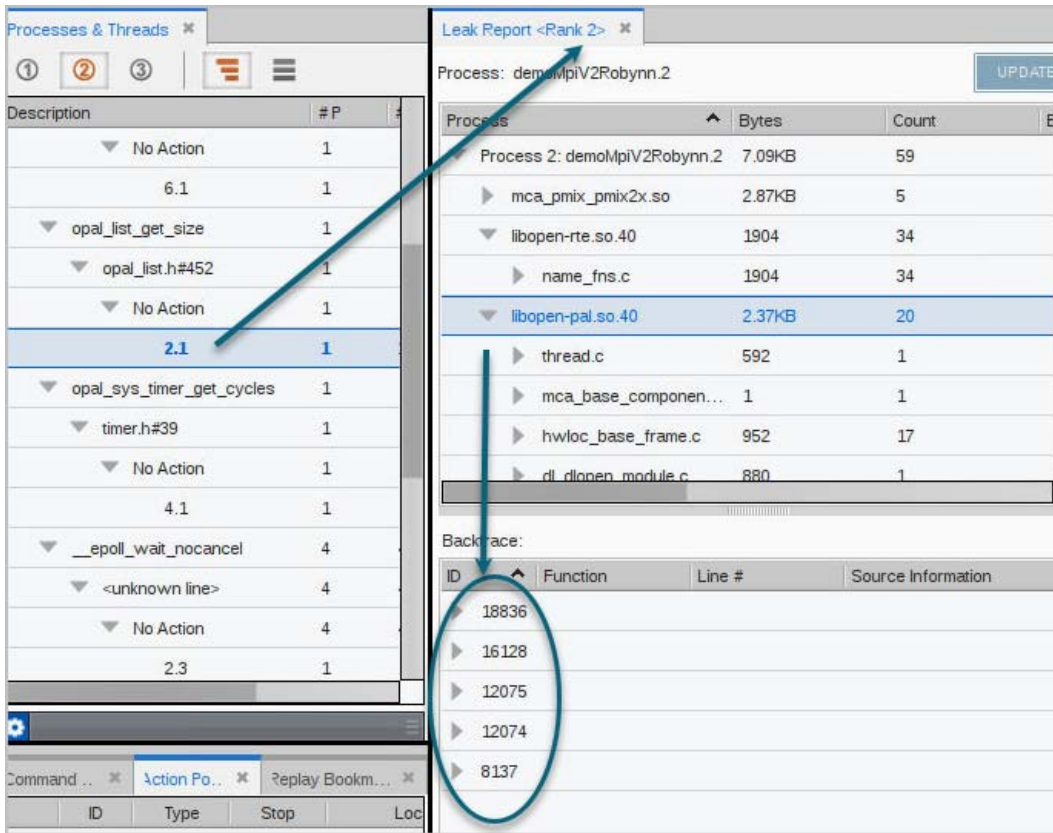
Updating the Leak Report

If you are stepping through your code or it has stopped at a breakpoint, the Leak Report view does not update automatically. At this point, the Update button becomes active, and selecting it will update the report.

MPI Programs and Leak Reports

If you are debugging an MPI program, it has ranks, rather than processes. A generated leak report will be based on the rank in focus. For example, the report in [Figure 180](#) is based on rank 2.

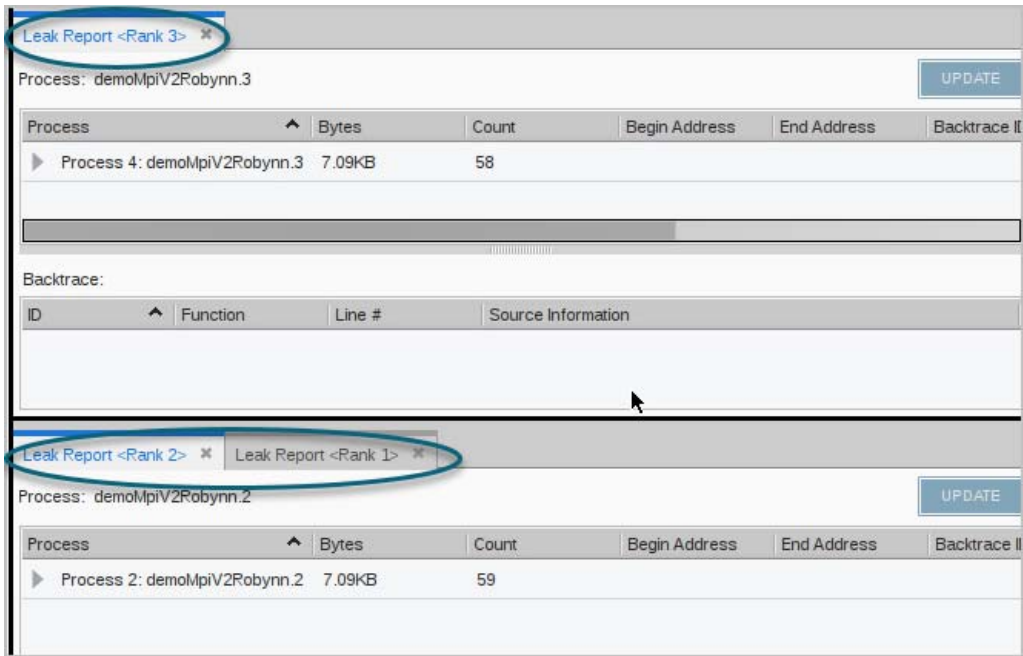
Figure 180, Leak Reports for MPI Programs



In this example, there are three libraries displayed in the upper pane. Opening one library reveals multiple leaks in the Backtrace.

You can generate multiple reports, one for each rank, [Figure 181](#):

Figure 181, Multiple Leak Reports display

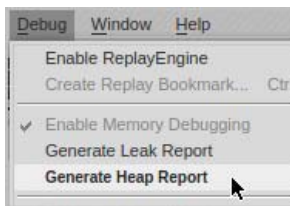


Memory Heap Reports

TotalView can locate heap allocations and display related information. Heap allocations are derived from monitoring program requests for memory (**malloc** or **new**).

Heap allocations are reported in the **Heap Report** view. To generate a heap report, enable memory debugging (**Debug > Enable Memory Debugging**), then run your program either to the end or to a breakpoint.

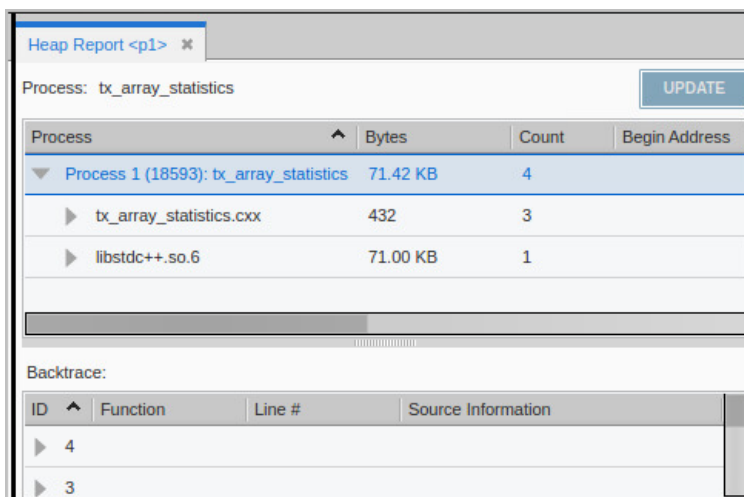
Select **Debug > Generate Heap Report**.



(Note that, when memory debugging is enabled and your program runs to the end without stopping for a breakpoint or other reason, a prompt displays before the program exits, asking if you want to create a Leak Report. A Heap Report is not dependent on a Leak Report so you may click No.)

The Heap Report displays docked in its own pane by default, [Figure 182](#), or as a tab next to the Leak Report if there is one open.

Figure 182, Heap Report, initial view



Using the Heap Report

Similar to the Leak Report, the Heap Report is based on the process in focus and includes two panes: at the top are the program's files; at the bottom is the Backtrace pane. All entries in the Program pane identify memory allocations in the program. To locate the allocations:

1. Drill down to identify the block in which each memory allocation resides, [Figure 183](#).

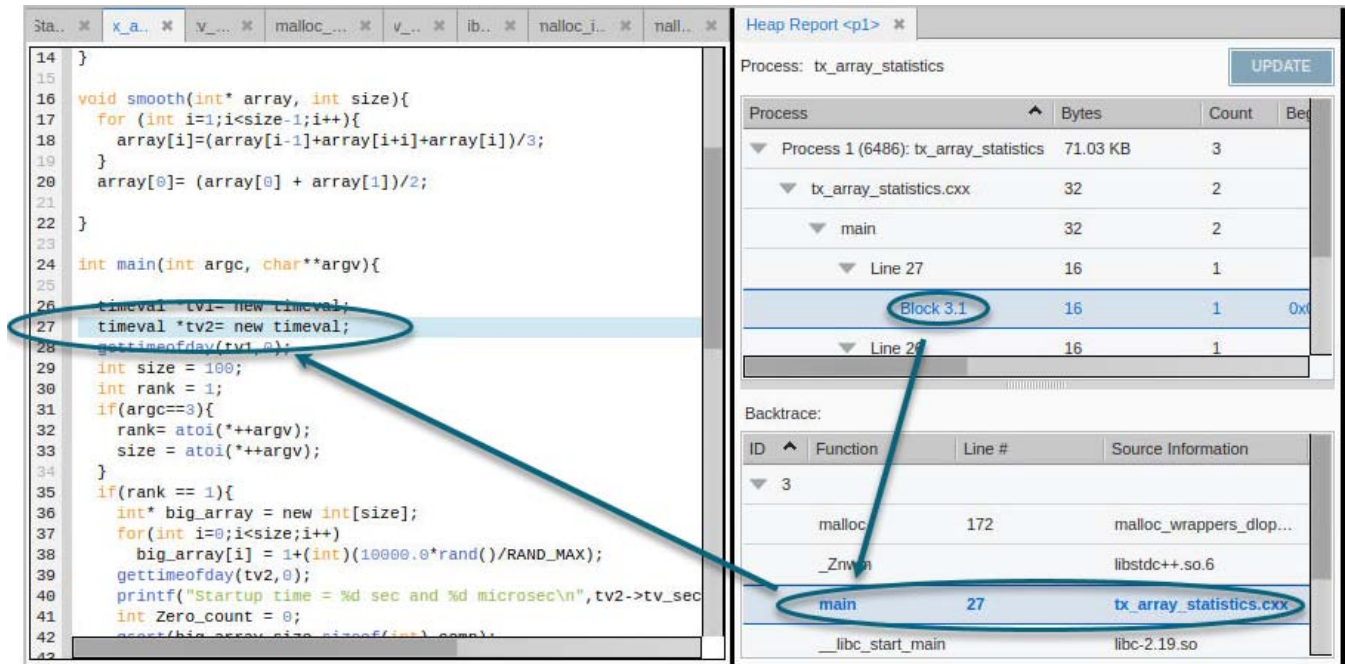
Figure 183, Heap Report: the Program Pane

Process	Bytes	Count	Begin Address
Process 1 (6486): tx_array_statistics	71.03 KB	3	
tx_array_statistics.cxx	32	2	
main	32	2	
Line 27	16	1	
Block 3.1	16	1	0x0112dc80
Line 26	16	1	
Block 2.1	16	1	0x0112dc60
libstdc++.so.6	71.00 KB	1	

Note that:

- This example program has a process that contains three separately-allocated memory blocks (identified by the **Count** column), two allocated by the program at lines 27 and 26.
 - The report is based on **p1**, or Process 1. If this program had multiple processes, focusing on another process would generate a separate Heap Report.
 - The **Bytes** column displays the number of bytes of allocated memory, rounded.
 - The **Update** button is grayed-out because the report is up to date. This will become active if the program continues to run past this point.
2. Select the block to analyze, which populates the Backtrace pane with the line of code where the memory was initially allocated, and focuses on the relevant line of code in the source pane:

Figure 184, Heap Report, Backtrace pane and source



Like the Leak Report, you can navigate from both the Backtrace pane and the program pane to the relevant line in the source pane.

The Memory Report backtrace reflects the backtrace that existed when your program made the heap allocation request. The backtrace ID helps associate each block of code allocated from this line.

There is a distinction between backtraces associated with the statement and the statement in your program that allocates memory. Suppose you have a function called **create_list()**. This function could be called from many different places in your code, and each location will have a separate backtrace. For example, if **create_list()** is called from eight different places and each was called five times, there would be eight different backtraces (and eight different backtrace IDs) associated with it. Each individual backtrace would have five items associated with it.

Updating the Heap Report

If you are stepping through your code or it has stopped at a breakpoint, the Heap Report view does not update automatically. At this point, the Update button becomes active, and selecting it will update the report.


Corrupt Guard Block Reports

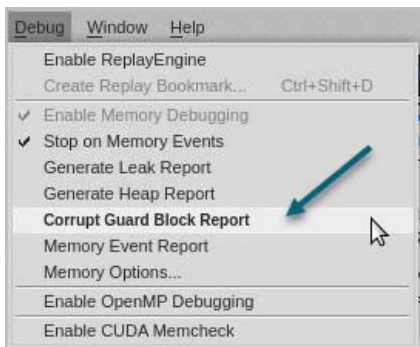
When your program allocates a memory block, no data should be written outside the block. For instance, an allocation of 16 bytes could not accommodate 32 bytes of information, as that data would write over the information in the next block, corrupting it.

If you enable guard blocks—by selecting the option “Guard Allocated Memory” either in the Session Editor or from the Debug menu—TotalView writes small blocks of information before and after all blocks that your program allocates. These additional blocks are called **guard blocks** and are 8 bytes of memory initialized with a specific bit pattern (although this can be customized). See [Enabling and Configuring Guard Blocks](#) for details.

If your program writes data into either of the guard blocks, it will result in corrupted data. View these corrupt memory blocks in two ways:

- By enabling the option **Debug > Stop on Memory Events**, which will halt your program and produce an event report when memory that has been overwritten is deleted. See [Example: Viewing a Guard Corruption Event Report](#) for detail.
- By running a **Corrupt Guard Block Report**, which reports all corrupt guard blocks in your program.

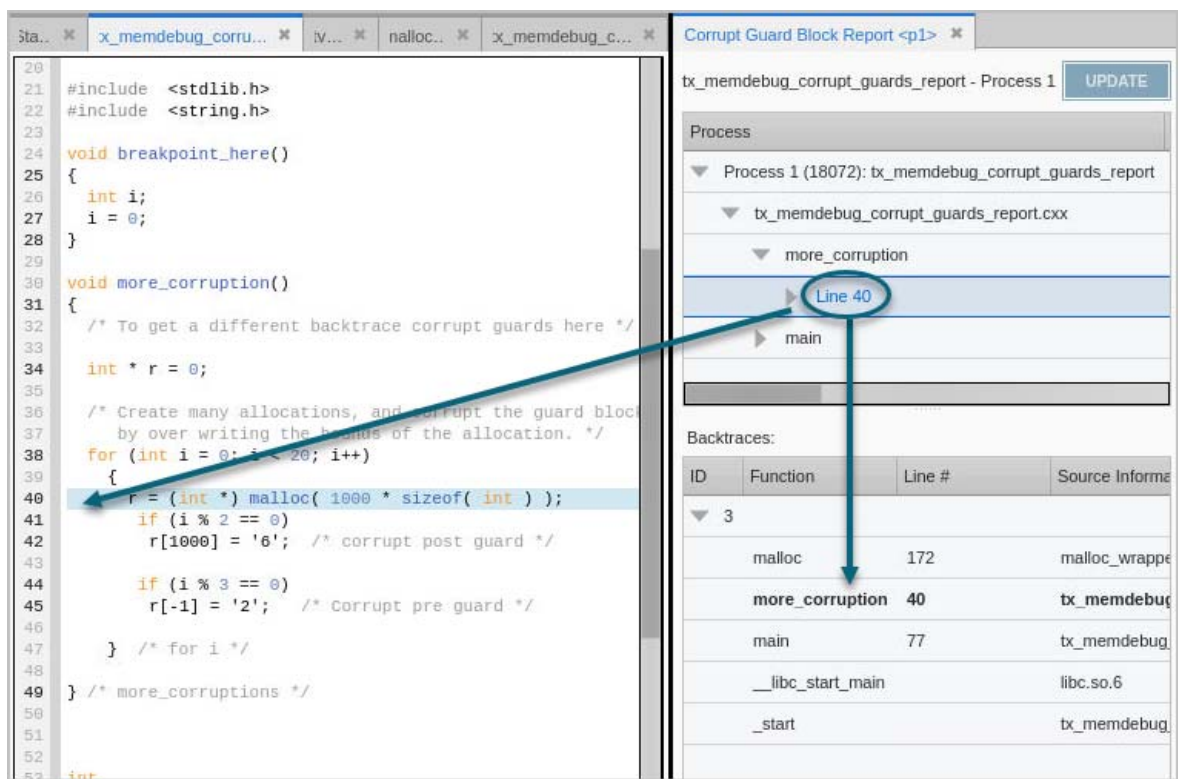
To run a Corrupt Guard Block Report, stop your program and choose **Debug > Corrupt Guard Block Report**, or press the memory toolbar button () “Generate a Corrupt Guard Block report”.



The report opens, displaying the files, the methods, and the specific lines in which any corrupted guard block exists. Like other reports, selecting an entry in the top pane focuses the Backtrace pane and the Source pane on that line.

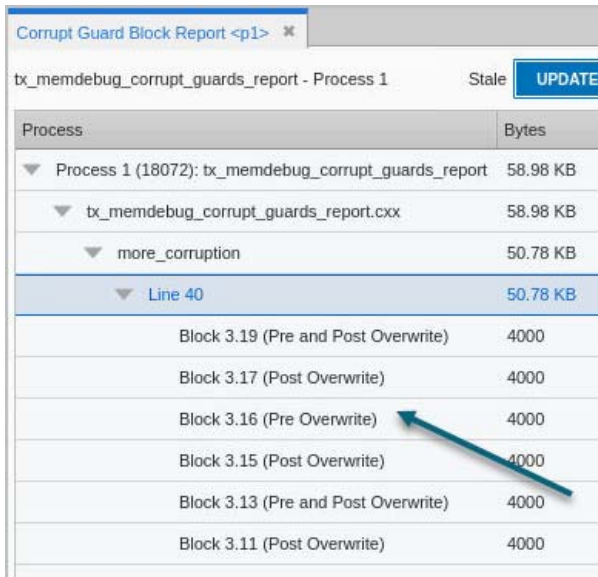
NOTE: Remember that this report displays where the guard blocks were allocated; it cannot actually identify where the overwrite occurred in your program. Strategies for locating the overwrite would entail rerunning your program and focusing on the block where the corrupt memory occurred. In addition, you might use watchpoints and ReplayEngine together by placing a watchpoint on the corrupt guard block address and then playing your program backward to the location where the watchpoint is triggered.

Figure 185, Corrupt Guard Block Report



Drill down into the line in the top pane to see detail about the memory block, whether the pre- or post-guard blocks were overwritten, or both.

Figure 186, Corrupt Guard Block Report, viewing detail



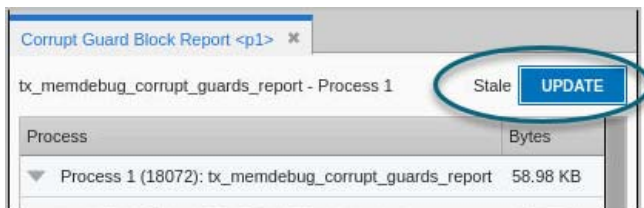
Corrupt Guard Block Report <p1> x

tx_memdebug_corrupt_guards_report - Process 1 Stale **UPDATE**

Process	Bytes
▼ Process 1 (18072): tx_memdebug_corrupt_guards_report	58.98 KB
▼ tx_memdebug_corrupt_guards_report.cxx	58.98 KB
▼ more_corruption	50.78 KB
▼ Line 40	50.78 KB
Block 3.19 (Pre and Post Overwrite)	4000
Block 3.17 (Post Overwrite)	4000
Block 3.16 (Pre Overwrite)	4000
Block 3.15 (Post Overwrite)	4000
Block 3.13 (Pre and Post Overwrite)	4000
Block 3.11 (Post Overwrite)	4000

Updating the Report

If you are stepping through your code or it has stopped at a breakpoint, the Corrupt Guard Block Report does not update automatically. At this point, the **Update** button becomes active; select it to update the report.



Memory Event Reports

A number of memory events can notify you and stop execution if you choose. For example, if the block was previously deallocated, TotalView can stop execution and report a “double free” error.

Table 20 lists all possible events for which you can receive notifications.

Table 20: Memory Events Logged by TotalView

Error	Description
API usage error	Incorrect API or API instance used in operation.
Allocation failed	Memory allocation failed.
Double free	Attempt to free a block that was already freed.
Free interior pointer	Attempt to free a block, but the address points into the interior of the allocated block rather than the address returned when the block was allocated.
Free notification	The user requested to stop the program when memory was freed.
Free unknown block	Attempt to free a non-allocated block.
Guard corruption	Attempt to free a block, but the guard blocks surrounding it have been altered. This indicates that your program wrote data where it should not have.
Invalid aligned allocation request	Supplied a bad address to a malloc routine.
Misaligned allocation	The block returned by the malloc library is not aligned on a byte boundary required by your operating system. The heap may be corrupted. (This is not a program error.)
Realloc notification	The user requested to stop the program when this block was reallocated.
Realloc unknown block	Attempt to reallocate an unallocated block.
Red Zone overrun error	Attempt to read or write past the end of your allocated block.
Red Zone overrun error on deallocated block	Attempt to read or write past the end of your deallocated block.
Red Zone underrun error	Attempt to read or write before the start of your allocated block.
Red Zone underrun error on deallocated block	Attempt to read or write before the start of your deallocated block.
Red Zone use-after-free error	Attempt to access a block after it has been deallocated.

Table 20: Memory Events Logged by TotalView

Error	Description
Termination notification	User has requested to halt before the program exits.
Unknown error	An error unknown to TotalView has occurred.

NOTE: Events that are controlled via a separate setting are not yet supported in the UI. For more detail, see **dheap** in the TotalView Reference Guide.

The Memory Event Report View

Memory events are logged in the **Event Report** view. To stop your program for memory errors and generate an event report, enable memory debugging (**Debug > Enable Memory Debugging**), then choose **Debug > Stop on Memory Events**.

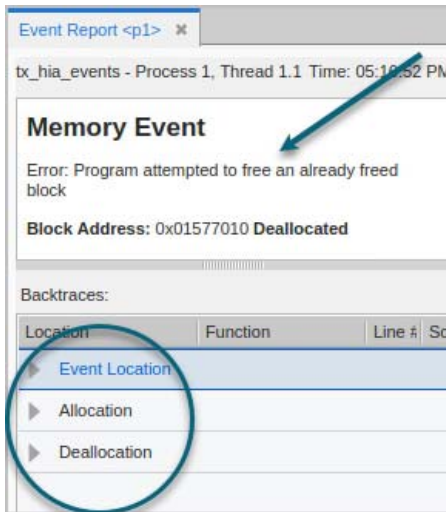


When a memory error occurs, TotalView stops your program and launches the Event Report view to display the error(s).

If you close the view during a debugging session, reopen it by choosing **Debug > Memory Event Report**.

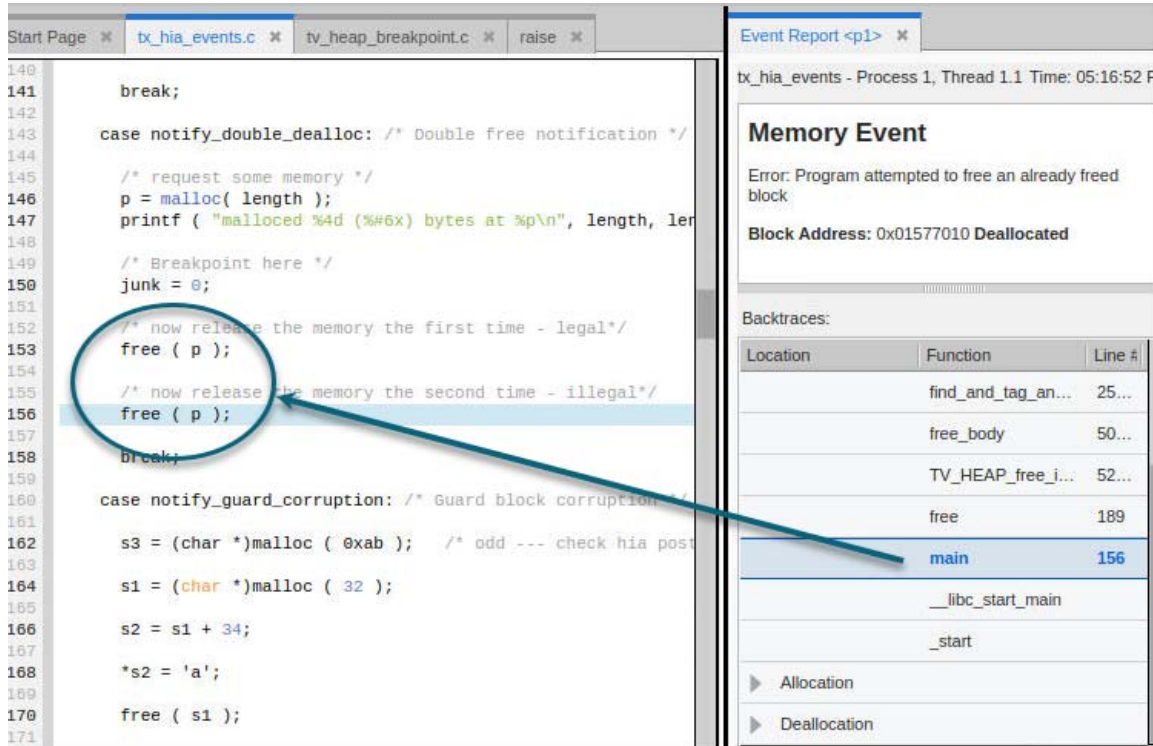
The Event Report view reports the error (in this example, a double free allocation error) and its block address. The view's Backtrace pane provides detail on the location of the error, as well as where it was allocated or deallocated, [Figure 187](#).

Figure 187, The Memory Event view



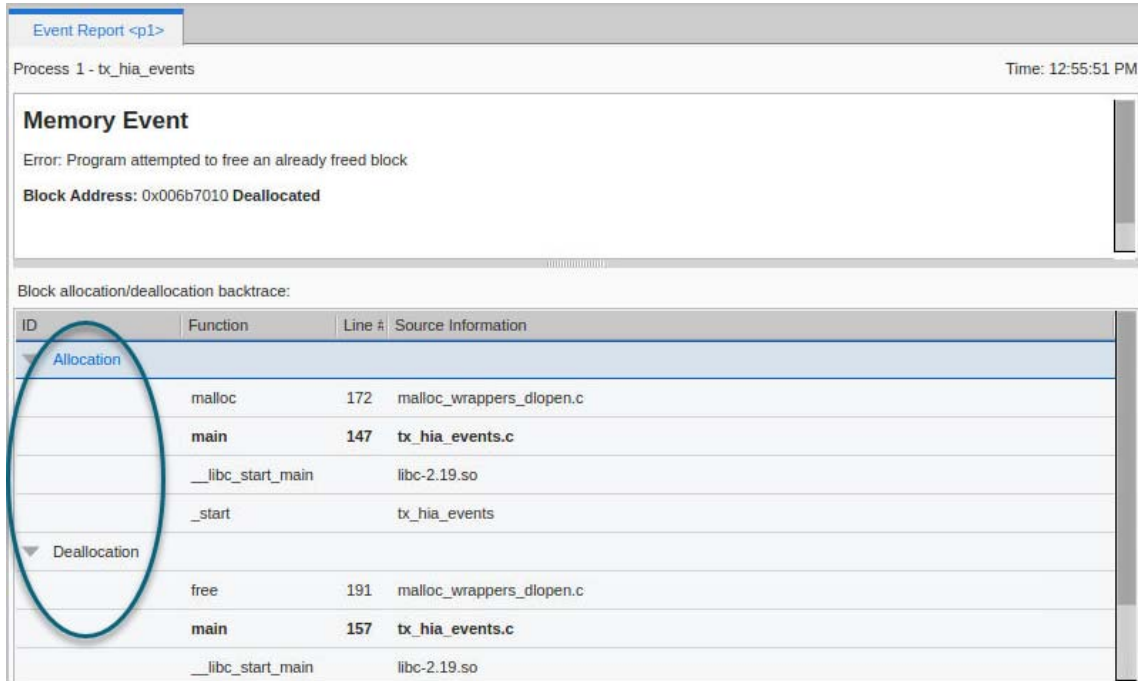
To locate the error, open the Event Location dropdown, which highlights the location of the error. Selecting the error in the dropdown displays the location of the error in the Source view.

Figure 188, Memory Event Report, Event Location



Additional information is available via the Allocation and Deallocation dropdowns in the Backtrace, [Figure 189](#).

Figure 189, Memory Event Report, backtrace pane



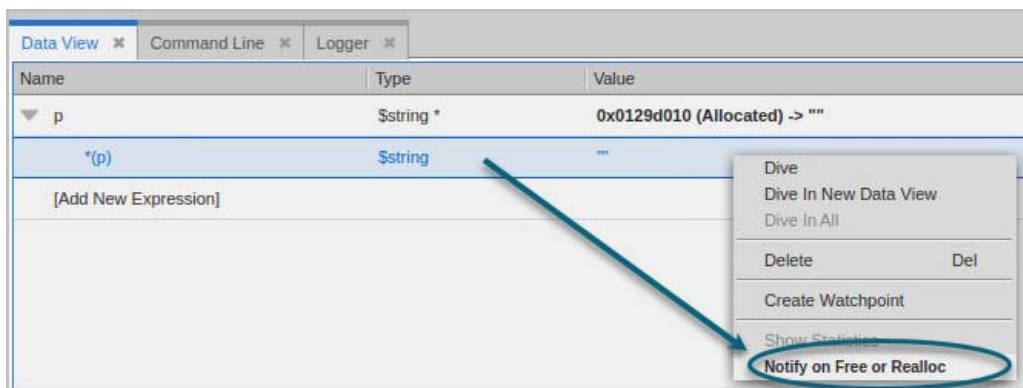
These dropdowns identify the lines at which any memory was allocated and deallocated in the program. Selecting a line in the backtrace displays that line in the Source view.

Memory Block Notification

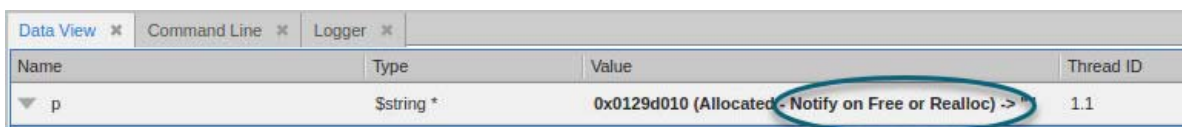
When a specific block of memory has been freed but your program tries to access that memory later through a dangling pointer reference, the result can be invalid data that causes sporadic crashes or data problems whose cause can be hard to pinpoint. To help identify these kinds of memory problems, you can set TotalView to watch a specific allocated memory block and then raise an event when the memory is freed or deleted.

To tag a specific memory block, run to a location in your code and examine a pointer variable in either the Local Variable view or Data View. Dereference the pointer, then right click on it, and select **Notify on Free or Realloc** from the context menu.

Figure 190, Memory blocks, “Notify on Free or Realloc”



A checkbox displays next to the context menu entry, and the variable’s entry in the Data View or Local Variables view reports that notifications are turned on for this variable:



When a tagged block of memory is freed or reallocated, the Event Report opens, displaying the exact location where the memory was both initially allocated and freed.

Figure 191, Memory blocks, Event Report

The screenshot displays a memory event report for 'tx_hia_events - Process 1, Thread 1.1'. The main section is titled 'Memory Event' and contains the error message: 'Error: Program attempted to free an already freed block'. A red arrow points to this message. Below the error, the 'Block Address' is listed as '0x0098c010 Deallocated'. The 'Backtraces' section is divided into three parts: a general backtrace table, an 'Allocation' section, and a 'Deallocation' section. In the 'Allocation' section, the entry for 'main' at line 146 in 'tx_hia_events.c' is circled in red. In the 'Deallocation' section, the entry for 'main' at line 156 in 'tx_hia_events.c' is also circled in red. The general backtrace table is as follows:

Location	Function	Line #	Source Information
	free_body	50...	malloc_interposers.c
	TV_HEAP_free_i...	52...	malloc_interposers.c
	free	189	malloc_wrappers_dlopen.c
	main	156	tx_hia_events.c
	__libc_start_main		libc.so.6
	_start		tx_hia_events

Allocation:

malloc	172	malloc_wrappers_dlopen.c
main	146	tx_hia_events.c
__libc_start_main		libc.so.6
_start		tx_hia_events

Deallocation:

free	191	malloc_wrappers_dlopen.c
main	156	tx_hia_events.c
__libc_start_main		libc.so.6
start		tx_hia_events

Memory Debugging Options

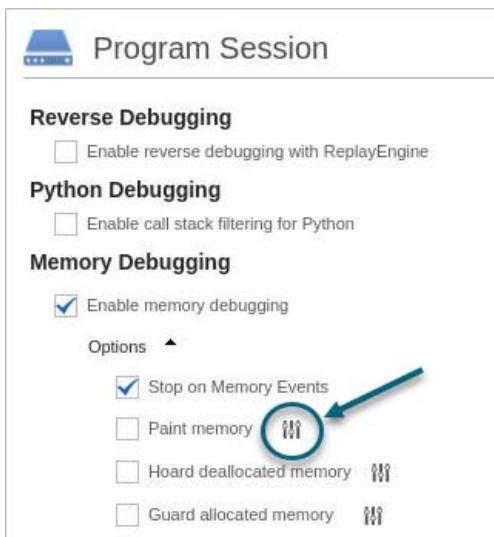
Several memory debugging options are available to refine your debugging session. Set these either in the Session Editor before starting your debugging session or from within a debugging session using the menu item **Debug > Memory Options** or by selecting the icon “Enable memory debugging” from the memory toolbar.

Available options are:

- Option: Painting Memory
- Option: Hoarding Memory Blocks
- Option: Guarding Allocated Memory

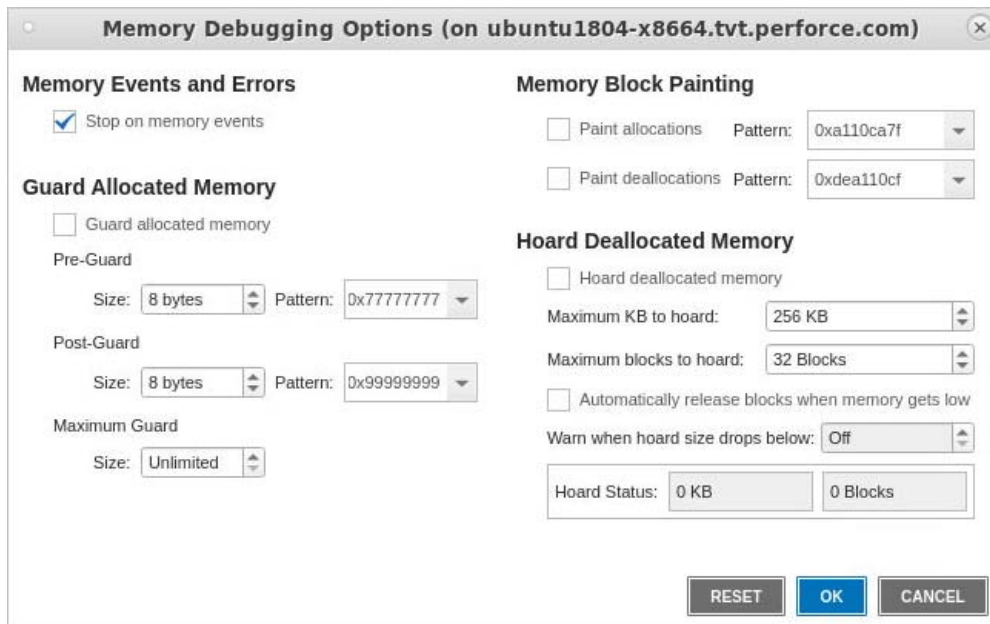
In the Session Editor, choose the settings icon to configure individual options:

Figure 192, Memory Options Available from the Session Editor



From within a debugging session, first choose **Debug > Enable Memory Debugging**, then **Memory Options** to open the Memory Options dialog:

Figure 193, Memory Options Available from the Debug Menu and the Memory Toolbar



CLI Commands

In the CLI, the **dheap** command controls memory debugging behavior, including memory options. For more detail, see **dheap** in the *TotalView Reference Guide*.

Option: Painting Memory

Your program may be using memory either before it is initialized or after it is deallocated. TotalView can help identify these kinds of problems by initializing allocated or deallocated memory to a bit pattern. This is called *painting*. Recognizing this bit pattern can help more quickly identify the problem.

Painting memory blocks is useful in:

- Identifying the use of uninitialized or deallocated memory. The bit pattern immediately tells you that memory is either allocated or deallocated.
- Ensuring consistency for multiple users, i.e., if a program works for some users and not for others, it will be clear if uninitialized or deallocated memory is the problem.
- Changing your program's behavior if it is not using memory correctly, which can help in identifying the problem. In addition, it may correct the problem so that the program doesn't appear to fail.
- Forcing an error such as a crash to occur, which can in turn help you identify the problem.

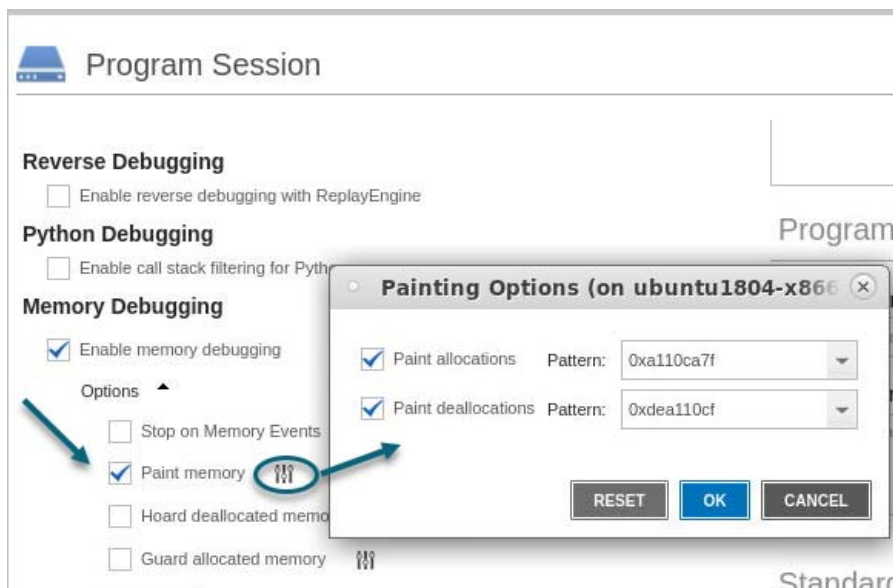
- Finding initialized memory by searching for the bit pattern to ensure all memory was correctly initialized.

NOTE: You can separately enable allocations and deallocations, and you can turn painting on and off without restarting your program.

Enabling and Configuring Painting

Enable memory painting from either the Program Session page in the Session Editor or from the Debug menu after your debugging session has started.

Figure 194, Enabling Memory Painting in the Session Editor



To customize the pattern used for painting, select the options () icon to open the Painting Options dialog:

- **Paint allocations pattern:**

The default is 0xa110ca7f, chosen for its resemblance to the word “allocate.”

- **Paint deallocations pattern:**

The default is 0xde110cf, chosen for its resemblance to the word “deallocate.”

Example: Viewing Painted Memory

If you turn on memory block painting before starting your program, you'll be ready to observe memory being allocated or deallocated. For example, the variable **addr** has not yet been allocated:

Name	Type	Value	Address
Arguments			
argc	int	0x00000002 (2)	0x7...
argv	\$string **	0x7ffc90e206b8 -> 0x7ffc90e211c3 -> "/nfs/...	0x7...
addr	int *	0x7f180c6d4b40 (&_dl_fini) -> 0xe5894855 (...	0x7...

After allocation, note the bit paint value:

Name	Type	Value	Address
Arguments			
argc	int	0x00000002 (2)	0x7...
argv	\$string **	0x7ffc65e69a28 -> 0x7ffc65e6a1c3 -> "/nfs/ho...	0x7...
addr	int *	0x0115feb0 (Allocated) -> 0xa110ca7f (159...	0x7...

NOTE: To identify exactly when a program has allocated or deallocated memory, change the deallocation pattern during a debugging session. This will show you memory allocated or deallocated before or after the bit pattern changed, so you can isolate the point in your program when it occurred.

Option: Hoarding Memory Blocks

When your program deallocates memory, it's possible that a pointer still points to the deallocated block. If your program then tries to access that block, the data can become corrupt. To give you more options to identify problems like dangling pointers or other issues with deallocation, you can *hoard* memory blocks.

Hoarding memory simply means that TotalView doesn't immediately release memory that has been deallocated; instead, it records the size of the block that *would have been deallocated*, and your program can continue accessing that memory for a defined amount of time.

How Hoarding Works

TotalView holds on to hoarded blocks for a specific amount of time before returning them to the heap manager for reuse. As TotalView adds blocks to the hoard, it places them in a first-in, first-out list. When the hoard is full, TotalView releases the oldest blocks back to your program's memory manager.

When hoarding is turned on, deallocated memory isn't available to be overwritten by your program, which allows your program to continue running normally even though it might be accessing memory that should have been deallocated. This can help uncover related errors, and these errors can help you track down the problem.

For example, if you are both painting memory and hoarding deallocated memory, you might be able to force an error when your program accesses the painted memory.

Consider a couple of examples.

Finding a Multithreaded Problem

When a multithreaded program shares memory, problems can occur if one thread deallocates a memory block while another thread is still using it. Because threads execute intermittently, these kinds of deallocation problems may not be immediately apparent. If you hoard memory, however, the memory stays available for longer because it cannot be reused immediately.

In this case, if intermittent program failures stop occurring, you'll be able to identify the problem more easily.

One advantage of this technique is that you can relink your program (see [Linking Your Application with the HIA](#)) and then run TotalView against a production program that was not compiled using the `-g` compiler debugging option. If you see instances of the hoarded memory, you'll instantly know problems have occurred.

This technique often requires that you increase the number of blocks being hoarded and the hoard size.

Finding Dangling Pointer References

Hoarding is most often used to find dangling pointer references. Once you know the problem is related to a dangling pointer, you need to locate where your program deallocated the memory. One technique is to use block tagging (see [Memory Block Notification](#)) to stop execution when memory is allocated or deallocated. Another is to use block painting (see [Option: Painting Memory](#)) to write a pattern into deallocated memory. If you also hoard painted memory, the heap manager cannot reallocate the memory as quickly.

For example, if the memory were not hoarded, the heap manager could reallocate the memory block at which point a program can legitimately use the block, changing the data in the painted memory. If this occurs, the block is both legitimately allocated and its contents are legitimate in some context. However, the older context was destroyed. Hoarding delays the recycling of the block. In this way, it extends the time available for you to detect that your program is accessing deallocated memory.

Enabling and Configuring Hoarded Memory

Enable memory hoarding from either the Program Session page in the Session Editor or from the Debug menu after your debugging session has started.

The option available from the Debug menu also reports hoard status, once your program is running:

- **Maximum KB to hoard:**

The default is 256 KB. To hoard all deallocated memory, enter **0**, which will then display “Unlimited.”

- **Maximum blocks to hoard:**

The default is 32 blocks. Enter **0** to hoard an unlimited number of blocks.

- **Automatically release blocks when memory gets low:**

If choosing “Unlimited” in the above options, consider also selecting this option to prevent or delay your program running out of memory.

Warn when hoard size drops below:

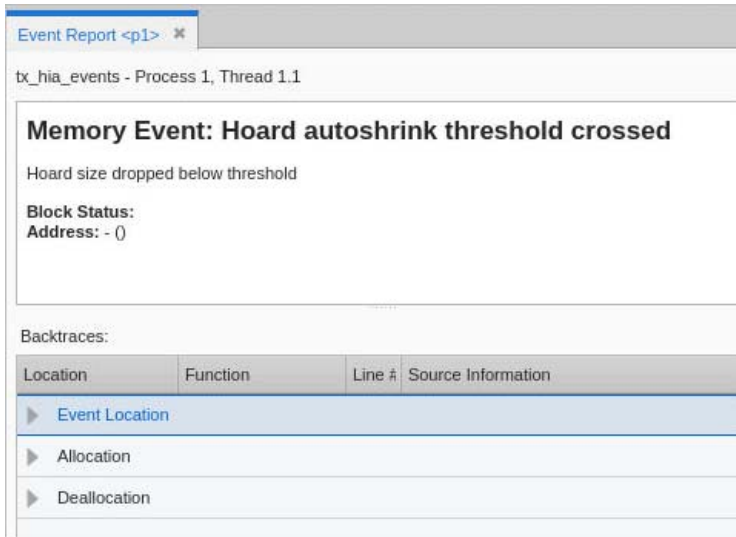
If blocks are automatically being released for low memory, you can also choose to have TotalView warn you if the hoard size drops below a certain number, in which case TotalView halts execution and notifies you. You can then view a heap or leak report to see where your memory is being allocated. Choose this option, then enter a value, for example:

- **Hoard Status:**

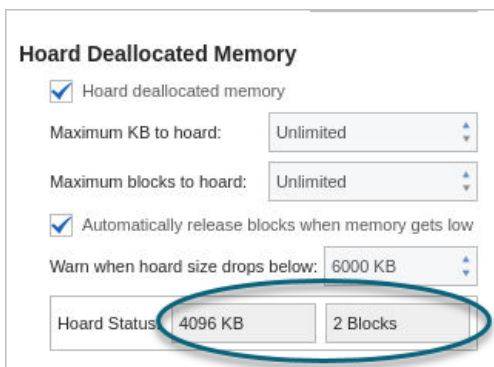
Reports the block hoarding status as your program runs.

Example: Hoarding Memory

Consider this program in which the maximum blocks to hoard and their size has been set to “unlimited,” along with a warning when hoard size drops below 6,000 KB. During program execution, if the hoard threshold is crossed, a memory event is written to the Event Report:



In addition, the Memory Options dialog reports hoard status:



Option: Guarding Allocated Memory

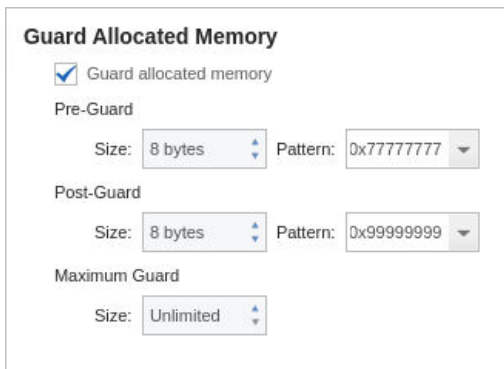
When your program allocates a memory block, any data written outside of the block results in data corruption.

To quickly identify instances in which your program may be incorrectly writing data either before or after a block, use guard blocks. Guard blocks are small blocks of additional memory created just before and after an allocated block. At initialization, TotalView writes a bit pattern into these guard blocks so you can quickly see if they are overwritten or corrupted via a memory event report or event notification.

Enabling and Configuring Guard Blocks

Enable allocated memory blocks from either the Program Session page in the Session Editor or from the Debug menu after your debugging session has started.

Figure 195, Guard Allocated Memory configuration



The screenshot shows a configuration window titled "Guard Allocated Memory". It contains a checked checkbox labeled "Guard allocated memory". Below this, there are three sections: "Pre-Guard" with "Size" set to "8 bytes" and "Pattern" set to "0x77777777"; "Post-Guard" with "Size" set to "8 bytes" and "Pattern" set to "0x99999999"; and "Maximum Guard" with "Size" set to "Unlimited".

- **Pre-Guard:**

The size of the pre-guard block and its pattern. The default is 0x77777777.

By default, each guard block uses 8 bytes of memory.

- **Post-Guard:**

The size of the post-guard block and its pattern. The default is 0x99999999.

- **Maximum Guard:**

The maximum size for the guard blocks. Because different operating systems align information differently, setting a maximum size here can ensure that blocks don't use an excessive amount of memory, if memory is tight.

The default is **0**, in which no maximum size is set.

Example: Viewing a Guard Corruption Event Report

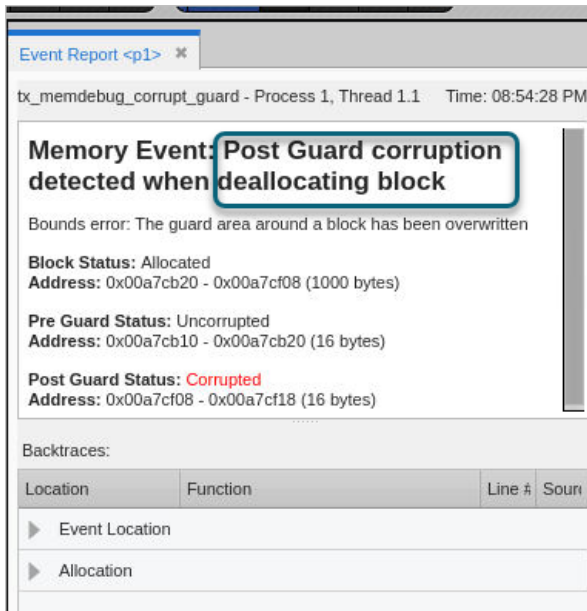
You can view corrupted guard blocks in your program in two ways:

- The guard corruption event report discussed in this section. This is a memory event generated when a specific corrupted block of memory is deleted.
- The Corrupt Guard Block Report, which lists *all* known blocks of memory with corrupted guard blocks. See [Corrupt Guard Block Reports](#) for detail.

When guard blocks are enabled, you'll want to choose to generate a memory event report by selecting **Debug > Stop on Memory Events** from the menu (see [Memory Event Reports](#)) for the types of memory events that produce an event report.

If your program has overwritten memory and guard blocks are enabled, TotalView generates an event report when the memory is deleted, for example:

Figure 196, Memory Event Report on Guard Corruption



In this case, TotalView has identified a corrupt post-guard block at memory deallocation, with the following information:

- **Block Status:**
The current status of this block is “allocated” at 1,000 bytes.
- **Pre Guard Status:**
The previous status was uncorrupted, at 16 bytes.
- **Post Guard status:**
The status is now corrupted, as it has been allocated with 1,000 bytes, overwriting the 16 bytes it previously had and corrupting memory outside of its allocation.

To find more information about this piece of memory, drill down into the Backtrace pane at the bottom of the event report:

Figure 197, Guard corruption event report, backtrace, deallocation event

The screenshot displays a debugger interface with two main panes. On the left is the source code view for `tx_memdebug_corrupt_guard.c`. The code includes a copyright notice, an update log, and a `main` function. The `main` function allocates a 1000-byte block with `malloc`, sets its content to 'a', and then calls `free(p);`. This line is highlighted in blue, and a red arrow points to it from the backtrace pane. On the right is the 'Event Report' pane, titled 'Memory Event: Post Guard corruption detected when deallocating block'. It provides details: 'Bounds error: The guard area around a block has been overwritten', 'Block Status: Allocated' (Address: 0x00a7cb20 - 0x00a7cf08, 1000 bytes), 'Pre Guard Status: Uncorrupted' (Address: 0x00a7cb10 - 0x00a7cb20, 16 bytes), and 'Post Guard Status: Corrupted' (Address: 0x00a7cf08 - 0x00a7cf18, 16 bytes). Below the report is a 'Backtraces' table with columns 'Location', 'Function', and 'Line #'. The table lists several frames, with the 'main' frame at line 38 highlighted in blue. A dropdown menu labeled 'Event Location' is open, showing the 'main' frame selected.

The Event Location dropdown in the Backtrace pane highlights the location of the deallocation, which occurred in `main()`. When selected, the focus in the Source view changes to the line where deallocation occurred.

To see the allocated block, drill down under Allocation in the Backtrace pane.

Figure 198, Guard corruption event report, backtrace, allocation event

The screenshot displays a memory debugging tool interface. On the left, a code editor shows the source file `tx_memdebug_corrupt_guard.c`. The code includes a license notice, an update log, and a `main` function. In `main`, a character array `p` is allocated with `malloc(1000 * sizeof(char))` on line 33 and then freed on line 38. A blue arrow points from the event report to line 33.

The right pane shows an event report for `tx_memdebug_corrupt_guard - Process 1, Thread 1.1 Time: 01:38:43 PM`. The report title is **Memory Event: Post Guard corruption detected when deallocating block**. The details are as follows:

- Bounds error: The guard area around a block has been overwritten
- Block Status: Allocated
Address: 0x01ff0b20 - 0x01ff0f08 (1000 bytes)
- Pre Guard Status: Uncorrupted
Address: 0x01ff0b10 - 0x01ff0b20 (16 bytes)
- Post Guard Status: **Corrupted**
Address: 0x01ff0f08 - 0x01ff0f18 (16 bytes)

Below the report is a backtrace table:

Location	Function	Line #	Sc
▶ Event Location			
▼ Allocation			
	malloc	172	
	main	33	
	__libc_start_main		
	_start		

Here, note that the 1,000 bytes allocated in main, line 33, were subsequently overwritten, resulting in a corrupt guard.

Because you now know which block was corrupted, you can begin to locate where the overwrite occurred. In many cases, you will rerun your program, focusing on those blocks.

Dangling Pointer Problems

Fixing dangling pointer problems is usually more difficult than fixing other memory problems. First of all, you become aware of them only when you realize that the information your program is manipulating isn't what it is supposed to be. Even more troubling, these problems can be intermittent, happening only when your program's heap manager reuses a memory block. For example, if nothing else is running on your computer, your program may never reuse the block. If there are a large number of jobs running, it could reuse a deallocated block quickly.

After you identify that you have a dangling pointer problem, you have two problems to solve. The first is to determine where your program freed the memory block. The second is to determine where it *should* free this memory.

When memory debugging is enabled, TotalView identifies dangling pointers in the Local Variable view and the Data View. You can also tag specific memory blocks so you are notified when memory is allocated or freed. See [Memory Block Notification](#).

Dangling Pointers in the Local Variables and Data Views

If you enable memory debugging, TotalView displays information in the Local Variables view about the variable's memory status; that is, whether the memory is allocated or deallocated. The following small program allocates a memory block, sets a pointer to the start of the block, and then deallocates the block:

```
main(int argc, char **argv)
{
    int *addr = 0; /* Pointer to start of block. */

    addr = (int *) malloc (10 * sizeof(int));

    /* Deallocate the block. addr is now dangling. */
    free (addr);
    /* Add some data to the array */
    addr[0] = 1;
    addr[1] = 2;
    ...
}
```

[Figure 199](#) shows the Local Variables view. In the top view, execution was stopped before your program executed the **free()** function. Note the memory indicator reporting that blocks are allocated.

Figure 199, Pointer allocated, then dangling

The screenshot shows the 'Local Variables' window in Visual Studio. It contains two panels. The top panel shows the state of local variables before a function call, and the bottom panel shows the state after the function call. A blue arrow points from the 'addr' variable in the top panel to the 'addr' variable in the bottom panel, illustrating the change in its state.

Name	Type	Value	Address
Arguments			
argc	int	0x00000002 (2)	0x...
argv	\$string **	0x7fff05faf458 -> 0x7fff05fb0a75 -> "/nfs/ntk-scratch/ho...	0x...
addr	int *	0x01baa010 (Allocated) -> 0x00000000 (0)	0x...
*addr	int	0x00000000 (0)	0x...
misaddr	int *	0x00000000	0x...
nbytes	int	0x000000ab (171)	0x...
ret_val	int	0x00000000 (0)	0x...
prog_name	\$string *	0x7fff05fb0a75 -> "/nfs/ntk-scratch/home/maryann/build...	0x...

argv	\$string **	0x7fff4dbd09d8 -> 0x7fff4dbd2a75 -> "/nfs/ntk-scratch/...	
addr	int *	0x01595010 (Dangling) -> 0xf50d77b8 (-183666760)	
misaddr	int *	0x00000000	
nbytes	int	0x000000ab (171)	
ret_val	int	0x00000000 (0)	
prog_name	\$string *	0x7fff4dbd2a75 -> "/nfs/ntk-scratch/home/maryann/buil...	

After your program executes the **free()** function, the message changes to “Dangling.”

If you drag the variable from the Local Variables view to the Data View, it also displays the memory indicator:

The screenshot shows the 'Data View' window in Visual Studio. It displays the state of a pointer variable 'addr' after it has been moved from the Local Variables window. The variable is shown as 'Dangling' with its current memory address and a warning icon.

Name	Value	Type
addr	0x0113b010 (Dangling) -> 0x15d807b8 (366479288)	int *
*addr	0x15d807b8 (366479288)	int
[Add New E...]		

Memory Scripting

You can execute memory debugging in batch mode using the **memscript** command and its options, like so:

```
memscript command_line_options
```

display_specifiers Command-Line Option

-display_specifiers controls information written to the log file.

```
-display_specifiers "list_item"
```

Specifies one or more items that can be added or excluded from the log file. Separate items with a comma.

list_item values are described in [Table 21](#). Use the prefix **no** to suppress the display.

Table 21: display_specifiers Command Line Options

Item	Controls display of ...
[no]show_allocator	The allocator for the address space
[no]show_backtrace	The backtrace for memory blocks
[no]show_backtrace_id	The backtrace ID for memory blocks
[no]show_block_address	The start and end addresses for a memory block
[no]show_flags	Memory block flags
[no]show_guard_id	The guard ID for memory blocks
[no]show_guard_settings	The guard settings for memory blocks
[no]show_image	The process/library associated with a backtrace frame
[no]show_owner	The owner of the allocation
[no]show_pc	The backtrace frame PC
[no]show_pid	The process PID
[no]show_red_zones_settings	The Red Zone entries for allocations and deallocations in the entire address space

event_action Command-Line Option

-event_action defines actions to perform for a particular event, like so:

-event_action "event=action list"

Specifies one or more actions to perform if an event occurs. The *"event=action list"* consists of comma-separated set *event=action* pairs. For example:

```
"alloc_null=save_memory_debugging_file, \
dealloc_notification=list_allocations"
```

event can be:

Table 22: event_action Command Line Option: events

Event	Description
addr_not_at_start	A block is being freed, and the address is not at the beginning of the block.
alloc_not_in_heap	The block being freed is not in the heap.
alloc_null	The malloc() function returned a null block.
alloc_returned_bad_alignment	The block is misaligned.
any_memory_event	All memory notification events.
bad_alignment_argument	The block returned by the malloc library is not aligned on a byte boundary required by your operating system. The heap may be corrupted. (This is not a program error.)
double_alloc	Allocator returned a block already in use. The heap may be corrupted.
double_dealloc	Program is attempting to free a block already freed.
free_not_allocated	Program is attempting to free a block that was not allocated.
guard_corruption	Guard corruption was detected when program deallocated a block.
hoard_low_memory_threshold	Hoard low memory threshold is crossed.
realloc_not_allocated	Program attempted to reallocate a block that was not allocated.
rz_overrun	Program attempted to access memory beyond end of allocated block.
rz_underrun	Program attempted to access memory before start of allocated block.
rz_use_after_free	Program attempted to access block after it was deallocated.
rz_use_after_free_overrun	Program attempted to access memory beyond end of deallocated block.
rz_use_after_free_underrun	Program attempted to access memory before start of deallocated block.
termination_notification	Program is about to execute its _exit routine.

action is as follows:

Table 23: event_action Command Line Option: actions

Action	Description
check_guard_blocks	Check for guard blocks and generate a corruption list.
list_allocations	Create a list of all your program's allocations.
list_leaks	Create a list of all of your program's leaks.
save_html_heap_status_source_view	Save the Heap Status Source report as an HTML file.
save_memory_debugging_file	Save a memory debugging file; you can reload this file at a later time.
save_text_heap_status_source_view	Save the Heap Status Source report as a text file.

Other Command Line Options

The **memscript** command takes these additional options.

-guard_blocks

Turn on guard blocks.

-red_zones_overruns

Turn on testing for Red Zone overruns.

-red_zones_underruns

Turn on testing for Red Zone underruns.

-detect_use_after_free

Turn on testing for use after memory is freed.

-hoard_freed_memory

Turn on the hoarding of freed memory.

-hoard_low_memory_threshold *nnnn*

Specify the low memory threshold that will generate an event.

-detect_leaks

Turn on leak detection.

-red_zones_size_ranges *min:max,min:max,...*

Specify the memory allocation ranges for which Red Zones are in effect. Ranges can be in the following formats:

x:y allocations from x to y: **y** allocations from 1 to y **x:** allocations of x and higher **x** allocation of x

-maxruntime *hh:mm:ss*

Specify the maximum amount of time the script should run where:

hh: number hours **mm**: number of minutes **ss**: number of seconds

As a script begins running, TotalView adds information to the beginning of the log file. This information includes time stamps for both the file when processes start, the name of the program, and so on.

memscript Example

The example here performs these actions:

- Runs the **filterapp** program under TotalView control.
- Passes an argument of **2** to the **filterapp** program.
- Whenever any event occurs—Heap Interposition Agent (**HIA**) event, SEGV, and the like—saves a memory debugging file.
- Allows the script to run for no longer than 5 seconds.
- Performs the following activities: use guard blocks, hoard freed memory, and detect memory leaks.

```
memscript -maxruntime "00:00:05" \  
-event_action "any_event=save_memory_debugging_file" \  
-guard_blocks -hoard_freed_memory -detect_leaks \  
~/Work/filterapp -a 2
```

Preparing Programs for Memory Debugging

- Compiling Programs for Memory Debugging
- Linking Your Application with the HIA
- Using env to Insert the HIA
- Installing tvheap_mr.a on AIX
- Using TotalView in Selected Environments

Compiling Programs for Memory Debugging

The first step when preparing a program to load for memory debugging is adding your compiler's **-g** debugging command-line option to generate symbol table debugging information; for example:

```
cc -g -o executable source_program
```

You can also take advantage of memory debugging on programs that you did not compile using the **-g** option, or programs for which you do not have source code. However, TotalView may not be able to provide source code information.

Table 24 presents some general considerations.

Table 24: Compiler Considerations

Compiler Option or Library	What It Does	When to Use It
Debugging symbols option (usually -g)	Generates debugging information in the symbol table.	Before debugging any program with TotalView.
Optimization option (usually -O)	Rearranges code to optimize your program's execution. Some compilers won't let you use the -O option and the -g option at the same time. Even if your compiler lets you use the -O option, don't use it when debugging your program, since unexpected results often occur.	After you finish debugging your program.
Multiprocess programming library (usually dbfork)	Uses special versions of the <code>fork()</code> and <code>execve()</code> system calls. In some cases, you need to use the lpthread option. For more information about dbfork , see Linking with the dbfork Library in the <i>TotalView Reference Guide</i> .	Before debugging a multiprocess program that explicitly calls <code>fork()</code> or <code>execve()</code> .

RELATED TOPICS

More on compiling programs

Saving action points

The **TV::stop_all** variable in the *TotalView Reference Guide*

Linking Your Application with the HIA

TotalView puts its Heap Interposition Agent ([HIA](#)) between your program and its heap library, which allows the HIA to intercept the calls that your program makes to this library. After it intercepts the call, it checks the call for errors and then sends it on to the library so that it can be processed. The TotalView HIA does not replace standard memory functions; it just monitors what they do.

In most cases, TotalView arranges for the HIA to be loaded automatically when it starts your program. In some cases, however, special steps must be taken to ensure the HIA loads. One example is when you are starting an MPI program using a launcher that does not propagate environment variables. (If you start your MPI program in TotalView using the Add Parallel Program page, TotalView propagates the information for you.) Another is when you want to start your program outside, or independently of, TotalView, and want to attach to the program later after it has started.

There are two ways you can arrange for the HIA to be loaded

Link the application with the HIA, as described in this section.

Request that the heap HIA be preloaded by setting the runtime loader's preloading environment variable. See [Using env to Insert the HIA](#).

Here is some important platform-specific information:

- **On AIX**, the `malloc` replacement code and HIA application must be in directories searched by the dynamic loader. If they are not in any of the standard directories (you can check with your system administrator), you can set `LIBPATH` to search these directories when you run the program. Another option is to add the directories to the program's list of search directories when you link the program. To do this, use the `-L` option as described in the table below. If you are in doubt about the directories being searched, you can obtain a list of the searched directories with `dump -Hv <program-name>`.

For additional requirements with AIX, see Installing tvheap_mr.a on AIX on page 15.

- **On Cray**, TotalView supports both static and dynamic linking. See the table below for the link lines you need to use.
- **On Apple Mac OSX**, you cannot link the HIA into your program.

[Table 25](#) lists additional command-line linker options that you must use when you link your program:

Table 25: Command Line Linker Options for Memory Debugging

Platform	Compiler	Binary Interface	Additional linker options
Cray XT, XE, XK CLE (dynamic)	-	64	<code>-dynamic -L<path> -ltvheap_64 -Wl,-rpath,<path></code>
Cray XT, XE, XK CLE (static)	-	64	<code>-L<path> -ltvheap_cnl</code>
IBM RS/6000 (all)	IBM/GCC	32/64	<code>-L<path_mr> -L<path></code>
AIX 5	IBM/GCC	32	<code>-L<path_mr> -L<path> <path>/aix_malloctype.o</code>
		64	<code>-L<path_mr> -L<path> <path>/aix_malloctype64_5.o</code>
Linux x86-64 1	GCC/Intel/PGI	32	<code>-L<path> -ltvheap -Wl,-rpath,<path></code>
		64	<code>-L<path> -ltvheap_64 -Wl,-rpath,<path></code>
		64	<code>-L<path> -ltvheap_64 -Wl,-rpath,<path></code>
Linux PowerLE	GCC	64	<code>-L<path> -ltvheap_64 -Wl,-rpath,<path></code>
Linux ARM64	GCC	64	<code>-L<path> -ltvheap_64 -Wl,-rpath,<path></code>
Sun	Sun/Apogee	32	<code>-L<path> -ltvheap -R <path></code>
		64	<code>-L<path> -ltvheap_64 -R <path></code>
	GCC	32	<code>-L<path> -ltvheap -Wl,-R,<path></code>
		64	<code>-L<path> -ltvheap_64 -Wl,-R,<path></code>

¹ On Ubuntu platforms, if the link line fails to start TotalView, try adding the additional flag `-Wl,-no-as-needed`. This flag should occur before the linking of `tvheap`, so on 64-bit platforms the link line would be:
`-L<path> -Wl,-no-as-needed -ltvheap_64 -Wl,-rpath,<path>`

Table Options:

- **<path>**
 The absolute path to the HIA in the TotalView installation hierarchy, located at:
`<installdir>/toolworks/TotalView.<version>/<platform>/lib`
- **<installdir>**
 The installation base directory name
- **<version>**
 TotalView version number

- **<platform>**
The platform tag
- **<path_mr>**
The absolute path of the **malloc** replacement library. This value is determined by the person who installs the TotalView **malloc** replacement library.

Using env to Insert the HIA

NOTE: The Heap Interposition Agent (HIA) library path can be hard to determine. Use this CLI command to print its path:

```
puts $TV::hia_local_dir
```

When TotalView attaches to a process that is already running, the HIA must already be associated with it. You can do this in two ways:

- Manually link the HIA as described in previous sections.
- Start the program using **env** (see `man env` on your system). This pushes the HIA into your program.

NOTE: Preloading cannot be used with Cray. For information on preloading with Cray, see [Linking Your Application with the HIA](#).

Table 26 lists the variables required by each platform. The placeholder `<hia_dir>` represents the directory in which the HIA is found.

Table 26: Variables by Platform for Preloading the HIA

Platform	Variable
Apple Mac OS X	DYLD_INSERT_LIBRARIES=<hia_dir>/libtvheap.dylib Note: See Mac OS for detail on how this environment variable works.
IBM AIX	MALLOCTYPE=user:tvheap_mr.a If you are already using MALLOCTYPE for another purpose, reassign its value to the variable TVHEAP_MALLOCTYPE and assign MALLOCTYPE as above; when the HIA starts it will correctly pass on the options.
Linux	
32-bit	LD_PRELOAD=<hia_dir>/libtvheap.so
64-bit	LD_PRELOAD=<hia_dir>/libtvheap_64.so
Sun	
32-bit generic	LD_PRELOAD=<hia_dir>/libtvheap.so
32-bit specific	LD_PRELOAD_32=<hia_dir>/libtvheap.so

Table 26: Variables by Platform for Preloading the HIA

Platform	Variable
64-bit generic	LD_PRELOAD=<hia_dir>/libtvheap_64.so
64-bit specific	LD_PRELOAD_64=<hia_dir>/libtvheap_64.so If the HIA is the only library you are preloading, use the generic variable. Otherwise, use whichever variable was used for the other preloaded libraries.

Installing tvheap_mr.a on AIX

NOTE: Installing tvheap_mr.a on AIX requires that the system have the bos.adt.syscalls System Calls Application Toolkit page installed.

You must install the **tvheap_mr.a** library on each node on which you plan to run the Heap Interposition Agent (HIA). The **aix_install_tvheap_mr.sh** script contains most of the required setup, and is located in this directory:

```
toolworks/totalview.<version>/rs6000/lib/
```

For example, after you become **root**, enter the following commands:

```
cd toolworks/totalview.<VERSION>/rs6000/lib
mkdir /usr/local/tvheap_mr
./aix_install_tvheap_mr.sh ./tvheap_mr.tar /usr/local/tvheap_mr
```

Use **poe** to create **tvheap_mr.a** on multiple nodes.

The pathname for the **tvheap_mr.a** library must be the same on each node. This means that you cannot install this library on a shared file system. Instead, you must install it on a file system that is private to the node. For example, because **/usr/local** is usually accessible only from the node on which it is installed, you might want to install it there.

NOTE: The tvheap_mr.a library depends heavily on the exact version of libc.a that is installed on a node. If libc.a changes, you must recreate tvheap_mr.a by re-executing the **aix_install_tvheap_mr.sh** script.

If this malloc replacement library changes (which is infrequent) you'll need to rerun this procedure. Any change will be noted among a release's new features.

LIBPATH and Linking

This section discusses compiling and linking your AIX programs. The following command adds **path_mr** and **path** to your program's libpath:

```
xlc -Lpath_mr -Lpath -o a.out foo.o
```

When **malloc()** dynamically loads **tvheap_mr.a**, it should find the library in **path_mr**. When **tvheap_mr.a** dynamically loads **tvheap.a**, it should find it in **path**.

The AIX linker supports relinking executables. This means that you can make an already complete application ready for the TotalView HIA; for example:

```
cc a.out -Lpath_mr -Lpath -o a.out.new
```

Here's an example that does not link in the heap replacement library. Instead, it allows you to dynamically set **MALLOCTYPE**:

```
xlc -q32 -g \
-L/usr/local/tvheap_mr \
-L/home/totalview/interposition/lib prog.o -o prog
```

This next example shows how a program can be set up to access the TotalView **HIA** by linking in the **aix_malloctype.o** module:

```
xlc -q32 -g \
-L/usr/local/tvheap_mr \
-L/home/totalview/interposition/lib prog.o \
/home/totalview/interposition/lib/aix_malloctype.o \
-o prog
```

You can check that the paths made it into the executable by running the **dump** command; for example:

```
% dump -Xany -Hv tx_memdebug_hello

tx_memdebug_hello:

      ***Loader Section***
      Loader Header Information
VERSION#  #SYMtableENT  #RELOCent  LENidSTR
0x00000001 0x0000001f    0x00000040 0x000000d3

#IMPfilID  OFFidSTR      LENstrTBL  OFFstrTBL
0x00000005 0x000000608   0x00000080 0x0000006db

      ***Import File Strings***
INDEX  PATH                BASE                MEMBER
0      /.../lib:/usr/.../lib:/usr/lib:/lib
1      libC.a              shr.o
2      libC.a              shr.o
3      libpthreads.a     shr_comm.o
4      libpthreads.a     shr_xpg5.o
```

Index 0 in the **Import File Strings** section shows the search path the runtime loader uses when it dynamically loads a library. Some systems propagate the preload library environment to the processes they will run; others, do not. If they do not, you need to manually link them with the **tvheap** library.

In some circumstances, you might want to link your program instead of setting the **MALLOCTYPE** environment variable. If you set the **MALLOCTYPE** environment variable for your program and it uses **fork()/exec()** a program that is not linked with the HIA, your program will terminate because it fails to find **malloc()**.

Using TotalView in Selected Environments

This topic describes using the memory debugger within various environments:

- MPICH
- IBM PE
- Mac OS
- Linux

MPICH

Here's how to use TotalView with MPICH MPI codes. This has been tested only on Linux x86-64.

1. You must link your parallel application with the TotalView [HIA](#) as described in [LIBPATH and Linking](#). On most Linux x86-64 systems, you'll enter:

```
mpicc -g test.o -o test -Lpath \  
-ltvheap -Wl,-rpath,path
```

2. Start TotalView using the **-tv** command-line option to the **mpirun** script in the usual way. For example:

```
mpirun -tv mpirun-args test args  
TotalView will start up on the rank 0 process.
```

3. If you need to configure TotalView, you should do it now.
4. Run the rank 0 process.

IBM PE

To use TotalView with IBM PE MPI codes:

1. You must prepare your parallel application to use the TotalView [HIA](#); see [LIBPATH and Linking](#) and [Installing tvheap_mr.a on AIX](#). Here is an example that usually works:

```
mpicc_r -g test.o -o test -Lpath_mr -Lpath \  
path/aix_malloctype.o
```

2. Start TotalView on **poe** as usual:

```
totalview poe -a test args
```

Because **tvheap_mr.a** is not in `poe`'s `LIBPATH`, enabling TotalView upon the `poe` process will cause problems because `poe` will not be able to locate the **tvheap_mr.a** malloc replacement library.

3. If you need to configure TotalView, you should do it now.
4. Run the `poe` process.

Mac OS

In most circumstances, memory debugging works seamlessly on the Mac OS.

From 10.11 El Capitan and onwards, however, the Mac OS introduced some changes that can affect some programs when memory debugging. While these should not affect how your program runs, in some rare cases you may want to fine-tune how the Heap Interposition Agent ([HIA](#)) behaves.

Background

In the Mac OS environment, interposition works only for preloaded DLLs, meaning that the HIA can only be preloaded rather than linked with the target as in some other operating systems. (See [Linking Your Application with the HIA](#) for more information on interposition and the HIA.)

The HIA makes sure that any environment variables related to preloading are correctly propagated if your program calls `execve()` or `system()`. The required Mac OS environment variable is `DYLD_INSERT_LIBRARIES`.

For all Mac OS releases from El Capitan onwards, however, a new feature System Integrity Protection (SIP) implemented a protocol that disallows passing `DYLD_INSERT_LIBRARIES` to a protected program or a program that resides in a protected directory. Calls to `system()` are affected because it is defined as invoking `/bin/sh`, which is in a SIP-protected directory.

Calls to `system()` on Mac OS

To work around the Mac OS SIP feature, for every `system()` call, the HIA copies `bin/sh` to a temporary directory (in `/tmp`) and arranges for the copy to be used so that `DYLD_INSERT_LIBRARIES` is not filtered out during the call. Once the child process has completed, the parent deletes the temporary directory.

This is the default behavior. To modify this, enable the environment variable `TV_MACOS_SYSTEM`.

Setting the Environment Variable `TV_MACOS_SYSTEM`

The `TV_MACOS_SYSTEM` environment variable allows customization of memory debugging behavior for Mac OS programs that call `system()`, and includes the following options:

pass_through=boolean

If **true**, the call to **system ()** is passed through to the underlying implementation.

The default is **false**, in which case the HIA controls the call to **system()** as described in [Calls to system\(\) on Mac OS](#). Setting this option to **true** may be useful if you have disabled SIP.

Example: "TV_MACOS_SYSTEM=pass_through=true"

shell=<pathname>

Defines the shell for the HIA to use instead of the default **bin/sh**.

If defined, be sure that the SIP does not control access to this shell so that the HIA has access to it. Note that the named shell is not deleted after the return from **system()**.

This setting may be useful to avoid any potential performance issues caused by copying the shell for each **system()** call, and then deleting it later.

Be aware, however, that using a previously stashed copy of **bin/sh** may require some maintenance, since the copy will not be updated when the operating system is updated.

The pathname must not contain commas or whitespace characters.

Example: "TV_MACOS_SYSTEM=shell=/path/to/some/copy/of/bin/sh"

tmpdir=<pathname>

Defines a temporary directory where the HIA will copy the shell (given the default setting of the option **pass_through=false**).

The HIA does not create the directory, assuming that it exists already. The HIA deletes the copy of the shell after processing is complete, but does not remove the directory.

If not set, the HIA creates a temporary directory in **/tmp**, which it removes after the call to **system ()** completes.

Example: "TV_MACOS_SYSTEM=tmpdir=/path/to/some/directory"

Linux

dlopen and RTLD_DEEPBIND

In most circumstances, memory debugging works seamlessly with programs that call **dlopen** to dynamically load shared objects (DSOs).

However, the Linux implementation of **dlopen** accepts **RTLD_DEEPBIND** in the **flags/mode** argument.

RTLD_DEEPBIND affects how undefined references in a DSO are bound. By default, when **RTLD_DEEPBIND** is not set, the dynamic linker first looks up any symbols needed by a newly-loaded DSO in the global scope.

RTLD_DEEPBIND modifies this behavior. When set, the dynamic linker places the lookup scope of the DSO ahead of the global scope. This means that the dynamic linker seeks to bind any undefined references in the DSO to definitions in the DSO, or any of the DSOs on which it depends. Only after these have been searched and a symbol not found is the global scope examined.

RTLD_DEEPBIND can affect memory debugging because, in some circumstances, references to all or part of the heap manager interface in a DSO can become bound to definitions in the standard library directly, rather than to those in the **HIA**. As a result, the HIA may not see all the traffic between the program and heap manager. If this occurs, the information the HIA is able to collect for TotalView will be incomplete, reducing its usefulness. In some circumstances, memory debugging may even fail.

How the HIA Handles RTLD_DEEPBIND

The HIA deals with the challenges posed by **RTLD_DEEPBIND** by intercepting calls to **dlopen**. If the program specifies **RTLD_DEEPBIND**, the HIA inserts itself as one of the to-be-loaded DSO's dependents. It does this by creating a new ELF wrapper file that lists the HIA and the DSO the program wants to **dlopen** as needed files. Instead of opening the DSO the program named, the HIA **dlopens** the new wrapper DSO it constructed. Since the DSO given by the program code is listed as a needed file, it too is opened.

As far as the program is concerned, the **dlopen** behaves as it would in the absence of the HIA. After the call to **dlopen**, the HIA cleans up and deletes the wrapper DSO that it created.

Modifying How the HIA Handles RTLD_DEEPBIND

The basic behavior described in [How the HIA Handles RTLD_DEEPBIND](#) can be modified by setting the **TVHEAP_DEEPBIND** environment variable. The following comma-separated settings are supported options:

pass_through=boolean

If **true**, the HIA does no special processing to handle **RTLD_DEEPBIND**. It does not create the ELF wrapper, and instead passes the operation through to the standard **dlopen**.

The default is **false**, in which case the HIA takes the steps described in [How the HIA Handles RTLD_DEEPBIND](#).

Example: "TVHEAP_DEEPBIND=pass_through=true"

keep_wrapper=boolean

If **true**, the HIA does not delete the ELF wrapper it creates after it has been used. The default is **false**, in which case the HIA deletes the ELF wrapper after it **dlopens** it.

Example: "TVHEAP_DEEPBIND=keep_wrapper=true"

tmpdir=<directory_name>

If defined, the HIA creates the ELF wrapper it generates in the directory specified by the **tmpdir** setting. The default is to use the setting of the environment variable **TMPDIR**. If **TMPDIR** is not defined, the ELF wrapper is created in **/tmp**.

PART VII Appendices

Appendix A

More on Expressions

Calling Functions: Problems and Issues

Unfortunately, calling functions using expressions can cause problems, such as in these scenarios:

- What happens if the function has a side effect? For example, suppose you have entered these two expressions into the Data View: `my_var[ctr]` and `my_var[++ctr]`. If `ctr` equals 3, you'll see the values of `my_var[3]` and `my_var[4]`. However, since `ctr` now equals 4, the first entry will no longer be correct.
- What happens when the function crashes (after all, you are debugging), doesn't return, returns the wrong value, or hits a breakpoint?
- What do calling functions do to your debugging interaction if the expression evaluation takes an excessive amount of time?
- What happens if a function creates processes and threads? Or worse, kills them?

In general, there are some protections in the code. For example, if you're displaying items in the Data View, TotalView avoids an infinite loop by evaluating items only once. This does mean that the information is accurate only at the time at which TotalView made the evaluation.

In most other cases, you're basically on your own. If there's a problem, you'll get an error message. If something takes too long, press the **Halt** button. But if a function alters memory values or starts or stops processes or threads that interfere with your debugging, restart your program. However, if an error occurs while using the Data View, pressing the **Stop** button pops the stack, leaving your program in the state it was in before you invoked the expression. However, changes made to heap variables will, of course, not be undone.

Using Built-in Variables and Statements

TotalView contains a number of built-in variables and statements that can simplify your debugging activities. You can use these variables and statements in evalpoints.

Topics in this section are:

- [Using TotalView Variables](#)
- [Using Built-In Statements](#)

Using TotalView Variables

TotalView variables that let you access special thread and process values. All variables are 32-bit integers, which is an **int** or a **long** on most platforms. The following table describes built-in variables:

Name	Returns
\$clid	The cluster ID. (Interpreted expressions only.)
\$duid	The TotalView-assigned Debugger Unique ID (DUID). (Interpreted expressions only.)
\$newval	The value just assigned to a watched memory location. (Watchpoints only.)
\$nid	The node ID. (Interpreted expressions only.)
\$oldval	The value that existed in a watched memory location before a new value modified it. (Watchpoints only.)
\$pid	The process ID.
\$processduid	The DUID (debugger ID) of the process. (Interpreted expressions only.)
\$systid	The thread ID assigned by the operating system. When this is referenced from a process, TotalView throws an error.
\$tid	The thread ID assigned by TotalView. When this is referenced from a process, TotalView throws an error.

The built-in variables let you create thread-specific breakpoints from the expression system. For example, the **\$tid** variable and the **\$stop** built-in function let you create a thread-specific breakpoint, as the following code shows:

```
if ($tid == 3)
```

```
$stop;
```

This tells TotalView to stop the process only when the third thread evaluates the expression.

You can also create complex expressions using these variables; for example:

```
if ($pid != 34 && $tid > 7)
    printf ("Hello from %d.%d\n", $pid, $tid);
```

Using any of the following variables means that the evalpoint is interpreted instead of compiled: **\$clid**, **\$duid**, **\$nid**, **\$processduid**, **\$systid**, **\$tid**, and **\$visualize**. In addition, **\$pid** forces interpretation on AIX.

You can't assign a value to a built-in variable or obtain its address.

Using Built-In Statements

TotalView statements help you control your interactions in certain circumstances. These statements are available in all languages, and are described in the following table. The most commonly used statements are **\$count**, **\$stop**, and **\$visualize**.

Statement	Use
\$count <i>expression</i>	Sets a process-level countdown breakpoint.
\$countprocess <i>expression</i>	When any thread in a process executes this statement for the number of times specified by <i>expression</i> , the process stops. The other processes in the control group continue to execute.
\$countall <i>expression</i>	Sets a program-group-level countdown breakpoint. All processes in the control group stop when any process in the group executes this statement for the number of times specified by <i>expression</i> .
\$countthread <i>expression</i>	Sets a thread-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by <i>expression</i> , the thread stops. Other threads in the process continue to execute. If the target system cannot stop an individual thread, this statement performs the same as \$countprocess . A thread evaluates <i>expression</i> when it executes \$count for the first time. This <i>expression</i> must evaluate to a positive integer. When TotalView first encounters this variable, it determines a value for <i>expression</i> . TotalView does not reevaluate until the <i>expression</i> actually stops the thread. This means that TotalView ignores changes in the value of <i>expression</i> until it hits the breakpoint. After the breakpoint occurs, TotalView reevaluates the <i>expression</i> and sets a new value for this statement. The internal counter is stored in the process and shared by all threads in that process.

Statement	Use
\$hold	Holds the current process.
\$holdprocess	If all other processes in the group are already held at this evalpoints, TotalView releases all of them. If other processes in the group are running, they continue to run.
\$holdstopall \$holdprocessstopall	Like \$hold , except that any processes in the group which are running are <i>stopped</i> . The other processes in the group are not automatically held by this call—they are just stopped.
\$holdthread	Freezes the current thread, leaving other threads running.
\$holdthreadstop \$holdthreadstopprocess	Like \$holdthread , except that it <i>stops</i> the process. The other processes in the group are left running.
\$holdthreadstopall	Like \$holdthreadstop , except that it stops the entire group.
\$stop \$stopprocess	Sets a process-level breakpoint. The process that executes this statement stops; other processes in the control group continue to execute.
\$stopall	Sets a program-group-level breakpoint. All processes in the control group stop when any thread or process in the group executes this statement.
\$stopthread	Sets a thread-level breakpoint. Although the thread that executes this statement stops, all other threads in the process continue to execute. If the target system cannot stop an individual thread, this statement performs the same as to \$stopprocess .
\$visualize(expression[slice])	Visualizes the data specified by <i>expression</i> and modified by the optional <i>slice</i> value. <i>Expression</i> and <i>slice</i> must be expressed using the code fragment's language. The <i>expression</i> must return a dataset (after modification by <i>slice</i>) that can be visualized. <i>slice</i> is a quoted string that contains a slice expression. For more information on using \$visualize in an expression, see "Using the Visualizer" on page 314.

Using Programming Language Elements

Using C and C++

This section contains guidelines for using C and C++ in expressions.

- You can use C-style (***/* comment */***) and C++-style (***// comment***) comments; for example:

```
// This code fragment creates a temporary patch  
i = i + 2; /* Add two to i */
```

- You can omit semicolons if the result isn't ambiguous.
- You can use dollar signs (\$) in identifiers. However, we recommend that you do not use dollar signs in names created within the expression system.

If your program does not use a templated function within a library, your compiler may not include a reference to the function in the symbol table. That is, TotalView does not create template instances. In some cases, you might be able to overcome this limitation by preloading the library. However, this only works with some compilers. Most compilers only generate STL operators if your program uses them.

You can use the following C and C++ data types and declarations:

- You can use all standard data types such as **char**, **short**, **int**, **float**, and **double**, modifiers to these data types such as **long int** and **unsigned int**, and pointers to any primitive type or any named type in the target program.
- You can only use simple declarations. Do not define **struct**, **class**, **enum** or **union** types or variables.

You can define a pointer to any of these data types. If an **enum** is already defined in your program, you can use that type when defining a variable.

- The **extern** and **static** declarations are not supported.

You can use the following the C and C++ language statements.

- You can use the **goto** statement to define and branch to symbolic labels. These labels are local to the window. You can also refer to a line number in the program. This line number is the number displayed in the Source Pane. For example, the following **goto** statement branches to source line number 432 of the target program:

```
goto 432;
```

- Although you can use function calls, you can't pass structures.

- You can use type casting.
- You can use assignment, **break**, **continue**, **if/else** structures, **for**, **goto**, and **while** statements. Creating a **goto** that branches to another TotalView evaluation is undefined.

Using Fortran

When writing code fragments in Fortran, you need to follow these guidelines:

- In general, you can use free-form syntax. You can enter more than one statement on a line if you separate the statements with semi-colons (;). However, you cannot continue a statement onto more than one line.
- You can use **GOTO**, **GO TO**, **ENDIF**, and **END IF** statements; Although **ELSEIF** statements aren't allowed, you can use **ELSE IF** statements.
- Syntax is free-form. No column rules apply.
- The space character is significant and is sometimes required. (Some Fortran 77 compilers ignore all space characters.) For example:

Valid	Invalid
<code>DO 100 I=1,10</code>	<code>DO100I=1,10</code>
<code>CALL RINGBELL</code>	<code>CALL RING BELL</code>
<code>X .EQ. 1</code>	<code>X.EQ.1</code>

You can use the following data types and declarations in a Fortran expression:

- You can use the **INTEGER**, **REAL**, **DOUBLE PRECISION**, and **COMPLEX** data types.
- You can't define or declare variables that have implied or derived data types.
- You can only use simple declarations. You can't use a **COMMON**, **BLOCK DATA**, **EQUIVALENCE**, **STRUCTURE**, **RECORD**, **UNION**, or array declaration.
- You can refer to variables of any type in the target program.
- TotalView assumes that **integer (kind=n)** is an n-byte integer.

Fortran Statements

You can use the Fortran language statements:

-
- You can use assignment, **CALL** (to subroutines, functions, and all intrinsic functions except **CHARACTER** functions in the target program), **CONTINUE**, **DO**, **GOTO**, **IF** (including block **IF**, **ENDIF**, **ELSE**, and **ELSE IF**), and **RETURN** (but not alternate return) statements.
 - If you enter a comment in an expression, precede the comment with an exclamation point (!).
 - You can use array sections within expressions. For more information, see “Array Slices and Array Sections” on page 289.
 - A **GOTO** statement can refer to a line number in your program. This line number is the number that appears in the Source Pane. For example, the following **GOTO** statement branches to source line number 432:

```
GOTO $432;
```

You must use a dollar sign (\$) before the line number so that TotalView knows that you’re referring to a source line number rather than a statement label.

You cannot branch to a label within your program. You can instead branch to a TotalView line number.

- The following expression operators are not supported: **CHARACTER** operators and the **.EQV.**, **.NEQV.**, and **.XOR.** logical operators.
- You can’t use subroutine function and entry definitions.
- You can’t use Fortran 90 pointer assignment (the **=>** operator).
- You can’t call Fortran 90 functions that require assumed shape array arguments.

Fortran Intrinsics

TotalView supports some Fortran intrinsics. You can use these supported intrinsics as elements in expressions. The classification of these intrinsics into groups is that contained within Chapter 13 of the *Fortran 95 Handbook*, by Jeanne C. Adams, *et al.*, published by the MIT Press.

TotalView does not support the evaluation of expressions involving complex variables (other than as the arguments for **real** or **aimag**). In addition, we do not support function versions. For example, you cannot use **dcos** (the double-precision version of **cos**).

The supported intrinsics are:

- Bit Computation functions: **btest**, **iand**, **ibclr**, **ibset**, **ieor**, **ior**, and **not**.
- Conversion, Null and Transfer functions: **achar**, **aimag**, **char**, **dbble**, **iachar**, **ichar**, **int**, and **real**.
- Inquiry and Numeric Manipulation Functions: **bit_size**.

-
- Numeric Computation functions: **acos**, **asin**, **atan**, **atan2**, **ceiling**, **cos**, **cosh**, **exp**, **floor**, **log**, **log10**, **pow**, **sin**, **sinh**, **sqrt**, **tan**, and **tanh**.

Complex arguments to these functions are not supported. In addition, on Macintosh and AIX, the **log10**, **ceiling**, and **floor** intrinsics are not supported.

The following are not supported:

- Array functions
- Character computation functions.
- Intrinsic subroutines

If you statically link your program, you can only use intrinsics that are linked into your code. In addition, if your operating system is Mac OS X, AIX, or Linux/Power, you can only use math intrinsics in expressions if you directly linked them into your program. The ****** operator uses the **pow** function. Consequently, it too must either be used within your program or directly linked. In addition, **ceiling** and **log10** are not supported on these three platforms.

Compiling for Debugging

- [Compiling with Debugging Symbols](#)
- [Maintaining Debug Information Separate from an Executable](#)

This appendix provides some important information on how to prepare your applications to be debugged with the TotalView debugger. The information covers only currently supported platforms. Please see the [Platforms Guide](#) for information.

Compiling with Debugging Symbols

The first step in getting a program ready for debugging is to add your compiler's **-g** debugging command line option. This option tells your compiler to generate symbol table debugging information; for example:

```
cc -g -o executable-name source-file
```

Here are a couple of general considerations about compiling your code:

Compiler option	What it does	When to use it
Debugging symbols option (usually -g)	Generates debugging information in the symbol table.	Before debugging <i>any</i> program with TotalView.
Optimization option (usually -O)	Rearranges code to optimize your program's execution. Some compilers let you use the -O option with the -g option, but we advise against doing this before using the debugger as unexpected results often occur.	After you finish debugging your program.

The following table lists the procedures to compile **C/C++ programs** on Linux platforms.

Compiler	Compiler Command Line
GCC C	gcc -g source.c
GCC C++	g++ -g source.cxx
clang C	clang -g source.c
clang C++	clang++ -gsource.cxx
Oracle Studio C	cc -g source.c
Oracle Studio C++	CC -g source.cxx
Intel C++ Compiler	icc -g source.cxx
PGI CC	pgcc -g source.c
PGI C++	pgc++ -gsource.cxx

The following table lists the procedures to compile **Fortran programs** on Linux platforms.

Compiler	Compiler Command Line
Absoft Fortran 77	f77 -g program .f f77 -g program.for
Absoft Fortran 90	f90 -g program.f90

Compiler	Compiler Command Line
Absoft Fortran 95	f95 -g program.f95
G77	g77 -g program.f
Intel Fortran Compiler	ifort -g program.f
Lahey/Fujitsu Fortran	lf95 -g program.f
PGI Fortran 77	pgf77 -g program.f
PGI Fortran 90	pgf90 -g program.f

The following table lists the procedures to compile programs on ARM64 platforms.

Compiler	Compiler Command Line
GCC C	gcc -g program.c
GCC C++	g++ -g program.cxx
G77	g77 -g program.f

The following table lists the procedures to compile programs on Mac OS platforms.

Compiler	Compiler Command Line
Absoft Fortran 77	f77 -gprogram .f f77 -gprogram.for
Absoft Fortran 90	f90 -gprogram.f90
Absoft Fortran 95	f95 -g program.f95
GCC C	gcc -g program.c
GCC C++	g++ -gprogram.cxx
GCC Fortran	gfortran -gprogram.f
clang C	clang -g source.c
clang C++	clang++ -gsource.cxx
Intel C++ Compiler	icc -g source.cxx
Intel Fortran Compiler	ifort -g program.f

Maintaining Debug Information Separate from an Executable

Because debug information embedded in an executable can be very large, some versions of Linux support stripping this information from the executable and placing it into a separate file. This file is then referenced within the executable using either a build ID section or a debug link section (or both) to identify the location and name of the separate debug file. The stripped image file will normally take up less space on the disk, and if you want the debug information, you can also install the corresponding **.debug** file.

The way this works with TotalView is controlled by a series of state variables and command line options discussed in [Controlling Separate Debug Files](#).

Create this file on Linux systems that have an **objcopy** that supports the **--add-gnu-debuglink** and **--only-keep-debug** command-line options. If **objcopy --help** mentions these options, creating this file is supported. See **man objcopy** for more details.

To create a separate file containing debug information:

1. Create a **.debug** copy of the executable or shared library. This second file is a regular executable but will contain only debugging symbol table information, with no code or data.
2. Create a stripped copy of the image file, and add to the stripped executable a **.gnu_debuglink** section that identifies the **.debug** file.

NOTE: The technique for creating a build ID separate debug information file is different and more complex than that for creating a debug link. Consult your system documentation for how to create a separate debug information file using the build ID method.

3. Distribute the stripped image and **.debug** files separately.

For example:

```
objcopy --only-keep-debug hello hello.debug
objcopy --strip-all hello
objcopy --add-gnu-debuglink=hello.debug hello
```

The code above uses **objcopy** to:

1. Create a separate debug file for an executable **hello** named **hello.debug**, containing only debug symbols and information.

-
- Strip the debug information from the **hello** executable.
 - Add a **.gnu_debuglink** section to the **hello** executable.

Controlling Separate Debug Files

The following command line options and CLI variables control how TotalView handles separate debug files.

- **Controls whether TotalView looks for either a build ID or a .gnu_debuglink section in image files:**
 - Command line options `-gnu_debuglink` and `-no_gnu_debuglink`
 - State variable `TV::gnu_debuglink`
This option basically turns on or off the functionality to support separate debug files.
- **Sets the search path to use when looking for debug files referenced by a .gnu_debuglink section:**
 - Command line option `-gnu_debuglink_search_path`
 - State variable `TV::gnu_debuglink_search_path`
- **Sets the search path to use when looking for debug files referenced by a .note.gnu.build-id section:**
 - Command line option `-gnu_debuglink_build_id_search_path`
 - State variable `TV::gnu_debuglink_build_id_search_path`
- **Specifies the global debug directory:**
 - Command line option `-gnu_debuglink_global_directory`
 - State variable `TV::gnu_debuglink_global_directory`
- **Validates the separate .gnu_debuglink debug file:**
 - ***Validate using a build ID, if available***
Command line option `-gnu_debuglink_check_build_id`
State variable `TV::gnu_debuglink_check_build_id`
 - ***Validate using a checksum***
Command line option `-gnu_debuglink_checksum`
State variable `TV::gnu_debuglink_checksum`

Searching for the Debug Files

If the `TV::gnu_debuglink` variable is **true** and if an image file contains either a `.note.gnu.build-id` or a `.gnu_debuglink` section, TotalView searches for a separate debug information file that matches the image file. TotalView will first search for the debug file using the `.note.gnu.build-id` section in the image file, if it exists. If that search fails, TotalView will search for the debug file using the `.gnu_debuglink` section in the image file, if it exists.

For the build ID method:

1. The `TV::gnu_debuglink_build_id_search_path` string is split at the colon (:) characters into a list of strings.
2. For each string on the list, "%D", "%G", and "%/" token expansion is performed to yield a list of directory names to search.
3. The list of directories is searched for the debug file path named by the `.note.gnu.build-id` section. The debug file path follows the pattern `".build-id/xx/yyy...yyy.debug"`, where `xx` are the first two hex characters of the build ID bit string, and `yyy...yyy` is the rest of the bit string. Build ID bit strings are at least 32 hex characters.

For separate debug files referenced by a `.gnu_debuglink` section:

1. The `TV::gnu_debuglink_search_path` string is split at the colon (:) characters into a list of strings.
2. For each string on the list, "%D", "%G", and "%/" token expansion is performed to yield a list of directory names to search.
3. The list of directories is searched for the debug file named in the `.gnu_debuglink` section. If the file is found, and the checksum matches or `TV::gnu_debuglink_checksum` is **false**, then the debug file is used.

For example, assume that the program's pathname is `/A/B/hello_world` and the debug filename stored in the `.gnu_debuglink` section of this program is `hello_world.debug`. If the `TV::gnu_debuglink_global_directory` variable is set to `/usr/lib/debug` and the `TV::gnu_debuglink_search_path` is set to its default value, TotalView searches for the following files:

1. `/A/B/hello_world.debug`
2. `/A/B/.debug/hello_world.debug`
3. `/usr/lib/debug/A/B/hello_world.debug`

Platform-Specific Topics

- Swap Space
- Shared Libraries

This appendix provides some platform-specific information that you may find useful. See [Platforms Guide](#) for specifics on platform support.

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
```

Another error you might see is:

```
job_t::launch, creating process: Operation failed
```

This error indicates that the `fork()` or `execve()` system call failed while TotalView was trying to create a process to debug.

To find out how much swap space has been allocated and is currently being used, use either the **swapon** or **top** commands.

To create additional swap space, use the **mkswap(8)** command.

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: `/lib/ld-linux.so.?`
 - Sets the PC to point to the location after the invocation of the dynamic loader but before the invocation of C++ static constructors or the `main()` routine.

When you attach to a process that uses shared libraries, TotalView takes the following actions:

- If you attached to the process after the dynamic loader ran, then TotalView loads the dynamic symbols for the shared library.
- If you attached to the process before it runs the dynamic loader, TotalView allows the process to run the dynamic loader to completion. Then, TotalView loads the dynamic symbols for the shared library.

If desired, you can suppress the recording and use of dynamic symbols for shared libraries by starting TotalView with the `-no_dynamic` option. Refer to the chapter “TotalView Command Syntax” in the *TotalView Reference Guide* for details on this startup option.

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 then modifies 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, 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.

You also need to enter `pxdb -s on`.

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.

Linux or Mac OS X

Add the following argument or command-line option to the command that you use to link your programs:

- `/usr/totalview/platform/lib/libdbfork_64.a`
- `-L/usr/totalview/platform/lib -ldbfork_64`

where *platform* is one of the following: **darwin-x86**, **linux-x86-64**, or **linux-arm64**.

Here is an example:

```
cc -o program program.c -L/usr/totalview/linux-x86-64/lib -ldbfork_64
```

Resources

- [Classic TotalView Documentation](#)
- [Conventions](#)
- [Contacting Us](#)

Classic TotalView Documentation

The following table describes all available documentation for Classic TotalView when using the classic UI. Much of the information is still applicable if you are using the new UI, although features not yet supported in the current interface must be accessed through the Command Line Interface (CLI) in the Command Line view.

Category	Title	Description	HTML	PDF	Print
General Classic TotalView Documentation					
	<i>Getting Started with Classic TotalView Products</i>	Introduces the basic features of Classic TotalView, MemoryScape, and ReplayEngine, with links for more detailed information	X	X	
	<i>Classic TotalView Platforms Guide</i>	Defines platform and system requirements for TotalView, MemoryScape, and ReplayEngine	X	X	
	<i>Classic TotalView Evaluation Guide</i>	Brochure that introduces basic Classic TotalView features		X	X
User Guides					
	<i>Classic TotalView User Guide</i>	Primary resource for information on using the Classic TotalView GUI and the CLI	X	X	
	<i>Debugging Memory Problems with MemoryScape</i>	How to debug memory issues, relevant to both Classic TotalView and the MemoryScape stand-alone product	X	X	
	<i>Reverse Debugging with Replay Engine</i>	How to perform reverse debugging using the embedded add-on ReplayEngine	X	X	
Reference Guides					
	<i>Classic TotalView Reference Guide</i>	A reference of CLI commands, how to run TotalView, and platform-specific detail	X	X	
New Features					

Conventions

Convention	Meaning
[]	Brackets are used when describing optional parts of a command.
<i>arguments</i>	In a command description, text in italics represents information you enter. Elsewhere, italics is used for emphasis.
Bold text	In a command description, bold text 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.
<code>Example text</code>	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.

Contacting Us

For technical support or suggestions for new features or improvements, contact TotalView:

- **By email:** support-TotalView@perforce.com
- **On the website:** Perforce Customer Support Portal at <https://portal.perforce.com>

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.

Open Source Software Notice

TotalView publishes the open source software products it uses in the `ATTRIBUTION.HTML` file located in the **doc** directory where you installed TotalView.

TotalView Glossary

This glossary defines terms specific to TotalView.

action point

A breakpoint. TotalView action points include standard breakpoints, watchpoints, eval points, and barriers.

action point identifier

A unique integer ID associated with an action point.

affected p/t set

The set of process and threads that are affected by the command. For most commands, this is identical to the target P/T set, but in some cases it might include additional threads. (See **p/t (process/thread) set** for more information.)

aggregated output

The CLI compresses output from multiple threads when they would be identical except for the P/T identifier.

arena

A specifier that indicates the processes, threads, and groups upon which a command executes. Arena specifiers are **p** (process), **t** (thread), **g** (group), **d** (default), and **a** (all).

array slice

A subsection of an array, which is expressed in terms of an upper bound, a lower bound, and a **stride**. Displaying a slice of an array can be useful when you are working with very large arrays.

autolaunching

When a process begins executing on a remote computer, TotalView can also launch a **tvdsvr** (TotalView Debugger Server) process on the computer that will send debugging information back to the TotalView process that you are interacting with.

automatic process acquisition

TotalView detects the many processes that parallel and distributed programs run in, and attaches to them automatically so you don't have to attach to them manually. If the process is on a remote computer, automatic process acquisition starts the -TotalView Debugger Server (**tvdsvr**).

barrier point

An action point specifying that processes reaching a particular location in the source code should stop and wait for other processes to catch up.

command history list

A debugger-maintained list that stores copies of the most recent commands issued by the user.

conditional breakpoint

A breakpoint containing an expression. If the expression evaluates to true, program stops. TotalView does not have conditional breakpoints. Instead, you must explicitly tell TotalView to end execution by using the `$stop` directive.

control group

All the processes that a program creates. These processes can be local or remote. If your program uses processes that it did not create, TotalView places them in separate control groups. For example, a client/server program has two distinct executables that run independently of one another. Each would be in a separate control group. In contrast, processes created by the **fork()** function are in the same control group.

debugger server

See **tvdsvr process**.

debugger state

Information that TotalView or the CLI maintains to interpret and respond to user commands. This includes debugger modes, user-defined commands, and debugger variables.

dpid

Debugger ID. The ID used for processes.

eval point

A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

expression system

A part of TotalView that evaluates C, C++, and Fortran expressions. An expression consists of symbols (possibly qualified), constants, and operators, arranged in the syntax of a source language. Not all Fortran 90, C, and C++ operators are supported.

focus

The set of groups, processes, and threads upon which a CLI command acts. The current focus is indicated in the CLI prompt (if you're using the default prompt).

gid

The TotalView group ID.

GOI

The group of interest. This is the group that TotalView uses when it is trying to determine what to step, stop, and so on.

group

When TotalView starts processes, it places related processes in families. These families are called "groups."

group of interest

The primary group that is affected by a command. This is the group that TotalView uses when it is trying to determine what to step, stop, and so on.

HIA

The Heap Interposition Agent, used when memory debugging. The HIA intercepts calls to heap library functions that allocate and deallocate memory by using the `malloc()` and `free()` functions and related functions such as `calloc()` and `realloc()`. In most cases, the HIA is loaded automatically when your program starts. For some platforms, however, the HIA needs to be explicitly linked to your application. See [Linking Your Application with the Agent](#) in the *TotalView User Guide*.

host computer

The computer on which TotalView is running.

initial process

The process created as part of a load operation, or that already existed in the runtime environment and was attached by TotalView or the CLI.

initialization file

An optional file that establishes initial settings for debugger state variables, user-defined commands, and any commands that should be executed whenever TotalView or the CLI is invoked. Must be called **.tvdrc**.

lockstep group

All threads that are at the same PC (program counter). This group is a subset of a workers group. A lockstep group only exists for stopped threads. All threads in the lockstep group are also in a workers group. By definition, all members of a lockstep group are in the same workers group. That is, a lockstep group cannot have members in more than one workers group or more than one control group.

manager thread

A thread created by the operating system. In most cases, you do not want to manage or examine manager threads.

native debugging

The action of debugging a program that is running on the same machine as TotalView.

pid

Depending on the context, this is either the process ID or the program ID. In most cases, this is the process ID.

POI

The process of interest. This is the process that TotalView uses when it is trying to determine what to step, stop, and so on.

process group

A group of processes associated with a multi-process program. A process group includes program control groups and share groups.

process/thread identifier

A unique integer ID associated with a particular process and thread.

process of interest

The primary process that TotalView uses when it is trying to determine what to step, stop, and so on.

program control group

A group of processes that includes the parent process and all related processes. A program control group includes children that were forked (processes that share the same source code as the parent), and children that were forked with a subsequent call to the **execve()** function (processes that don't share the same source code as the parent). Contrast this with **share group**.

program event

A program occurrence that is being monitored by TotalView or the CLI, such as a breakpoint.

p/t (process/thread) set

The set of threads drawn from all threads in all processes of the target program.

pthread ID

The ID assigned by the Posix pthreads package. If this differs from the system TID, it is a pointer value that points to the pthread ID.

satisfaction set

The set of processes and threads that must be held before a barrier can be satisfied.

satisfied

A condition that indicates that all processes or threads in a group have reached a barrier. Prior to this event, all executing processes and threads are either running because they have not yet hit the barrier, or are being held at the barrier because not all of the processes or threads have reached it. After the barrier is **satisfied**, the held processes or threads are released, which means they can be run. Prior to this event, they could not run.

serial line debugging

A form of remote debugging where TotalView and the **tvdsvr** communicate over a serial line.

service thread

A thread whose purpose is to *service* or manage other threads. For example, queue managers and print spoolers are service threads. There are two kinds of service threads: those created by the operating system or runtime system and those created by your program.

share group

All the processes in a control group that share the same code. In most cases, your program has more than one share group. Share groups, like control groups, can be local or remote.

single process server launch

A TotalView procedure that individually launches **tvdsvr** processes.

slice

A subsection of an array, which is expressed in terms of a lower bound, upper bound, and **stride**. Displaying a slice of an array can be useful when you are working with very large arrays.

stop set

A set of threads that TotalView stops after an action point -triggers.

stride

The interval between array elements in a slice and the order in which TotalView displays these elements. If the stride is 1, TotalView displays every element between the lower bound and upper bound of the slice. If the stride is 2, TotalView displays every other element. If the stride is -1, TotalView displays every element between the upper bound and lower bound (reverse order).

target computer

The computer on which the process to be debugged is running.

target process set

The target set for those occasions when operations can only be applied to entire processes, not to individual threads in a process.

target program

The executing program that is the target of debugger operations.

target p/t set

The set of processes and threads on which a CLI command acts.

thread of interest (TOI)

The primary thread affected by a command.

tid

The thread ID. On some systems (such as AIX where the threads have no obvious meaning), TotalView uses its own IDs.

trigger set

The set of threads that can trigger an action point (that is, the threads upon which the action point was defined).

triggers

The effect during execution when program operations cause an event to occur (such as arriving at a breakpoint).

tvdsvr process

The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.

type transformation facility (TTF)

Abbreviated as TTF. A TotalView subsystem that allows you to change the way information appears. For example, an STL vector can appear as an array.

user thread

A thread created by your program.

watchpoint

An action point that stops execution when the value of a memory location changes.

worker thread

A thread in a workers group. These are threads created by your program that perform the task for which you've written the program.

workers group

All the worker threads in a **control group**. Worker threads can reside in more than one **share group**.

Index

- -add-gnu-debuglink command-line option 587

Symbols

. (dot) current set indicator 372, 381

.rhosts file 324

/ slash in group specifier 375

& intersection operator 380

< first thread indicator (CLI) 372

- difference operator 380

| union operator 379

\$address data type 176

\$char data type 176

\$character data type 176

\$clid built-in variable 577

\$code data type 176

\$complex data type 176

\$complex_16 data type 176

\$complex_8 data type 176

\$count built-in function 106, 141, 578

\$countall built-in function 578

\$countthread built-in function 578

\$double data type 177

\$double_precision data type 177

\$duid built-in variable 577

\$extended data type 177

\$float data type 177

\$hold built-in function 579

\$holdprocess built-in function 579

\$holdprocessall built-in function 579

\$holdstopall built-in function 579

\$holdthread built-in function 579

\$holdthreadstop built-in function 579

\$holdthreadstopall built-in function 579

\$holdthreadstopprocess built-in

function 579

\$int data type 177

\$integer data type 177

\$integer_1 data type 177

\$integer_2 data type 177

\$integer_4 data type 177

\$integer_8 data type 177

\$logical data type 177

\$logical_1 data type 177

\$logical_2 data type 177

\$logical_4 data type 177

\$logical_8 data type 177

\$long data type 177

\$long_long data type 177

\$mpiexec variable 336

\$newval built-in function 116

\$newval built-in variable 577

\$newval intrinsic 114

\$nid built-in variable 577

\$oldval built-in function 116

\$oldval built-in variable 577

\$oldval intrinsic 114

\$oldval watchpoint variable 116

\$pid built-in variable 577

\$processduid built-in variable 577

\$real data type 177

\$real_16 data type 177

\$real_4 data type 177

\$real_8 data type 177

\$short data type 178

\$stop built-in function 117, 579

\$stop function 265

\$stopall built-in function 579

\$stopprocess built-in function 579

\$stopthread built-in function 579

\$string data type 178

\$systid built-in variable 577

\$tid built-in variable 577

\$visualize built-in function 579

\$void data type 178

\$wchar data type 178

\$wchar_s16 data type 178

\$wchar_s32 data type 178

\$wchar_u16 data type 178

\$wchar_u32 data type 178

\$wstring data type 178

\$wstring_s16 data type 178

\$wstring_s32 data type 178

\$wstring_u16 data type 178

\$wstring_u32 data type 178

Numerics

0xa110ca7f allocation pattern 545

0xdea110cf deallocation pattern 545

A

-a command-line option 54, 252
passing arguments to program 54

a width specifier 376

acquiring processes 326

action point

At Location 91

pending breakpoint 91

Action Point > Properties
command 97

Action Point ID attribute
definition 225

action points

and process/thread state 129

and templated code 89

CLI-available options 139

defined 86

disabling or deleting 66, 134

diving on 134

evaluation points 140

identifiers, never reused in a session 259

- loading 144
- properties 86
- saving 144
- setting 89
- sorting 133
- state 136
- suppressing and
 - unsuppressing 136
- tutorial 65
- unique IDs 86
- viewing variables at
 - breakpoint 69
 - width 128
- Action Points view 16
- adapter_use option 324
- adding command-line
 - arguments 45
- adding environment variables 45
- addr_not_at_start event 557
- \$address 176
- Address not at start of block
 - problems 519
- address space 508
- addresses
 - specifying in variable
 - window 210
- advancing
 - and holding processes 258
 - program execution 258
- agent, inseting with env 565
- agent. See heap debugging.
- agent's shared library 512
- aggregate presentation of process
 - and thread attributes 212
- aix_install_tvheap_mr.sh script 567
- aliases
 - built-in 255
 - group 255
 - group, limitations 255
- alloc_not_in_heap event 557
- alloc_null event 557
- alloc_returned_bad_alignment
 - event 557
- allocation
 - 0xa110ca7f pattern 545
- AMD GPU
 - process, defined 484
- AMD GPUs
 - GPU thread selector 491
- any_event event 557
- arena specifiers
 - inconsistent widths 378
- arenas
 - iterating over 371
- args command-line option 54
- ARGS variable 252
 - modifying 252
- ARGS_DEFAULT variable 252
 - clearing 252
- arguments
 - passing to program 54
 - replacing 252
- ARM64
 - compiling with debugging
 - symbols 586
- array of structure
 - dereferencing pointers 184
- array of structures
 - displaying 182
- array pointers 210
- array services handle (ash) 329
- Array Statistics view 187
- arrays
 - checksum statistic 190
 - count statistic 189
 - denormalized count
 - statistic 190
 - evaluating expressions 147
 - infinity count statistic 190
 - lower adjacent statistic 189
 - maximum statistic 189
 - mean statistic 189
 - median statistic 189
 - minimum statistic 189
 - NaN statistic 190
 - quartiles statistic 189
 - slice, initializing 261
 - slice, printing 262
 - slicing in the Array Statistics
 - view 196
 - standard deviation
 - statistic 189
 - sum statistic 189
 - upper adjacent statistic 190
- writing to file 263
 - zero count statistic 189
- ash (array services handle) 329
- ash (array services handle) 329
- ASM icon 368, 534, 545
- assigning output to variable 250
- assigning p/t set to variable 372
- At Location dialog 91
- attach options
 - enabling ReplayEngine 39
- Attach to a Running Program
 - dialog 35
- attaching
 - commands 42
 - configuring a debug
 - session 35
 - to job 326
 - to MPICH application 319
 - to MPICH job 319
 - to PE 326
 - to poe 326
 - to processes 35, 326
 - to RMS processes 328
 - to SGI MPI job 329, 330
- attaching to programs 565
- attaching to relatives 38
- attributes
 - defined for processes and
 - threads 225
 - different views of 217
 - selecting 214
 - setting order of 214
- auto_save_breakpoints
 - variable 144
- automatic dereferencing 210
- automatic process acquisition 319, 323
- automatic variables 515

B

- bad_alignment_argument
 - event 557
- barrier points 120, 122
 - defined (again) 120
 - satisfying 123
 - states 120
 - stopped process 125
- barrierpoints

- defined 259
- bit painting
 - 0xa110ca7f 545
 - 0xdea110cf 545
- branching around code 107
- breakpoint operator 380
- breakpoints
 - and MPI_Init() 326
 - automatically copied from master process 319
 - conditional 105, 578
 - copy, master to slave 319
 - countdown 106, 578
 - defined 259
 - entering 329
 - example setting in multiprocess program 98
 - fork() 97
 - loading 144
 - not shared in separated children 98
 - reloading 325
 - saving 144
 - set while running parallel tasks 325
 - setting 88, 89, 263, 325
 - shared by default in processes 97
 - sliding 90
 - thread-specific 577
- built-in aliases 255
- built-in functions
 - \$count 106, 141
 - \$countall 578
 - \$countthread 578
 - \$hold 579
 - \$holdprocess 579
 - \$holdprocessall 579
 - \$holdstopall 579
 - \$holdthread 579
 - \$holdthreadstop 579
 - \$holdthreadstopall 579
 - \$holdthreadstopprocess 579
 - \$stop 117, 579
 - \$stopall 579
 - \$stopprocess 579
 - \$stopthread 579
 - \$visualize 579
- built-in variables 577
 - \$clid 577

- \$duid 577
- \$newval 577
- \$nid 577
- \$oldval 577
- \$pid 577
- \$processduid 577
- \$string 176
- \$systid 577
- \$tid 577
- forcing interpretation 578

C

- C casting for Global Arrays 308, 309
- C control group specifier 376
- C/C++ statements
 - expression system 580
- C++
 - compiling on x86-64 585
- CAF (CoArray Fortran) 314
- Call Stack view 60, 67
 - responding to Processes and Threads view actions 220
- capture command 250
- casting Global Arrays 308, 309
- CGROUP variable 381
- ch_lfshmem device 317
- ch_mpl device 317
- ch_p4 device 317, 320
- ch_shmem device 317, 320
- changing command-line arguments 45
- changing process thread set 384
- changing program state 246
- \$char data type 176
- \$character data type 176
- chasing pointers 210
- check_guard_blocks action 558
- checksum array statistic 190
- CLI
 - components 244
 - in startup file 247
 - initialization 247
 - invoking program from shell
 - example 247
 - not a library 245
 - output 250
 - prompt 248

- setting action points 139
- starting 247
- starting from command prompt 247
- CLI commands
 - assigning output to variable 250
 - capture 250
 - dactions -load 144, 325
 - dactions -save 140, 144, 325
 - dattach 37, 42, 55, 319, 326, 327, 331
 - dattach mprun 331
 - dbarrier 120, 122, 123
 - dbreak 265
 - ddelete 332
 - ddisable 125
 - default focus 384, 385
 - dfocus 384
 - dga 308
 - dgo 325, 329, 379
 - dgroups -add 381
 - dhold 121
 - dkill 249
 - dload 42, 248, 258
 - dout 386
 - dprint 210, 262
 - drerun 248
 - drun 248, 252
 - dsession 41
 - dset 252, 254
 - dstatus 124
 - dstep 372, 373, 379, 386
 - dunhold 121
 - dunset 252
 - duntil 386
 - dup 210
 - dwhere 210, 373, 379
- CLI variables
 - ARGS 252
 - ARGS_DEFAULT 252
 - clearing 252
 - ARGS, modifying 252
 - auto_save_breakpoints 144
 - data format 209
 - EXECUTABLE_PATH 31, 34, 261
 - gnu_debuglink 588
 - LINES_PER_SCREEN 251
 - parallel_configs 334, 335
 - PROMPT 254

- SHARE_ACTION_POINT 97
- VERBOSE 246
- CLI view 16
- \$clid built-in variable 577
- cluster ID 577
- CoArray Fortran (CAF) 314
- \$code data type 176
- code constructs supported
 - C/C++ 580
 - Fortran 581
- code fragments 577
- code, branching around 107
- columns
 - ascending and descending order 216
- command arguments 252
 - clearing example 252
 - passing defaults 252
 - setting 252
- Command Line view 16, 60
- command prompts 254
 - default 254
 - format 254
 - setting 254
 - starting the CLI from 247
- command-line options 248
 - a 252
 - a 54
 - passing to debugger 54
 - passing to TotalView 54
 - s startup 247
- commands
 - Action Point > Properties 97
 - debugging, described 23
 - File -> Manage Sessions 49
 - Group > Attach 328, 329, 330
 - Group > Go 98, 325
 - Group > Kill 332
 - interrupting 246
 - mpirun 329
 - poe 319, 324
 - Process > Go 328, 329
 - Process > Out 386
 - Process > Run To 386
 - prun 328
 - rsh 324
 - Run To 391
 - scope 23
 - Tools > View Across 313
 - Tools > Visualize
 - Distribution 312
 - Tools > Watchpoint 116
 - totalview command 329
 - totalviewcli command 329
 - View > Dive In All 182
 - width 23
 - Window > Start Page 29
- compiling
 - CUDA programs. See CUDA, compiling.
 - on ARM64 586
 - on x86-64 585
 - programs 56
 - using -g option 56
- compiling programs 561
- \$complex data type 176
- \$complex_8 data type 176
- complex types
 - viewing in Data View 74
- \$complex_16 data type 176
- concealed allocation 521
- conditional breakpoints 105, 578
- configure new debugging session 31
- continuous execution 246
- Control Group attribute definition 225
- control groups
 - defined 364
- control in parallel environments 258
- control in serial environments 258
- controlling program execution 258
- core dump, naming the signal that caused 40
- core files
 - debug session in Sessions Editor 39
 - debugging 53, 54
 - multi-threaded 40
- correcting programs 108
- count array statistic 189
- \$countall built-in function 578
- countdown breakpoints 106, 578
- counter, loop 106
- \$countthread built-in function 578
- cpu_use option 324
- Cray
 - loading TotalView 305
 - starting the CLI 305
 - starting TotalView 305
- Cray XT, XE, and XK debugging 305
- creating new processes 248
- Ctrl+C 246
- CUDA
 - @parameter qualifier 457
 - @register storage qualifier 459
 - assigned thread IDs 450
 - CLI and operating on CUDA threads 457
 - compiling a program for debugging 447
 - compiling options 447
 - compiling Pascal GPU 447
 - compiling Tesla GPU 447
 - compiling Volta GPU 448
 - coordinate spaces, 4D and 5D 451
 - CUDA thread defined 450
 - data from CUDA thread, displaying 452
 - execution, viewing 450
 - g -G compiling option 447
 - GPU focus thread 450
 - GPU thread selector 451
 - host thread, viewing 451
 - logical coordinate space 451
 - MemoryChecker 44, 469
 - nvcc compiler 447
 - physical coordinate space 451
 - process, defined 443
 - PTX register, locations 458
 - ReplayEngine limitations 476
 - runtime variables, supported 455
 - sample program 477
 - single-stepping GPU code 451
 - starting TotalView for CUDA applications 448
 - storage qualifier, supported types 453
 - troubleshooting 475
 - type casting 456
 - variables from CUDA thread, displaying 452

variables, editing 456
current set indicator 372
custody changes 521

D

dactions command
-load 144, 325
-save 140, 144, 325
dangling pointer
problems 554
dangling pointers 517
dangling pointers and leaks
compared 518
data section 509
data types
see also data types
\$address 176
\$char 176
\$character 176
\$code 176
\$complex 176
\$complex_16 176
\$double_precision 177
\$extended 177
\$float 177
\$int 177
\$integer 177
\$integer_1 177
\$integer_2 177
\$integer_4 177
\$integer_8 177
\$logical 177
\$logical_1 177
\$logical_2 177
\$logical_4 177
\$logical_8 177
\$long 177
\$long_long 177
\$real 177
\$real_16 177
\$real_4 177
\$real_8 177
\$short 178
\$string 178
\$void 178
\$wchar 178
\$wchar_s16 178
\$wchar_s32 178
\$wchar_u16 178

\$wchar_u32 178
\$wstring 178
\$wstring_s16 178
\$wstring_s32 178
\$wstring_u16 178
\$wstring_u32 178
predefined 176
Data View 15
adding global variables 169
adding variables to 72
capabilities 15
responding to Processes and
Threads view actions 220
viewing updated values 74
viewing variables 71
data_format variables 209
dattach command 37, 42, 55, 319,
326, 327, 331
mprun command 331
dbarrier command 120, 122, 123
dbfork library 38
dbreak command 265
ddelete command 332
ddisable command 125
deadlocks 389
deallocate, defined 514
deallocation
0xdea110cf pattern 545
debug info in separate file 587
Debug Options
Reverse Debugging 32, 35
-debug, using with MPICH 332
debugger initialization 247
debugger PID 257
Debugger Unique ID (DUID) 577
debugging
attributes in Processes and
Threads view 214
basic tutorial 56
command descriptions 23
core file 53
PE applications 324
script 54
SHMEM library code 309
Start Page 21
starting a session 21
toolbar 23
UPC programs 310

debugging attributes
different views of 217
debugging command-line
option 523
debugging commands
Run To 391
debugging core files
in the Debug Core File
dialog 39
debugging session 258
configure new 31
debugging techniques 331
debugging toolbar 12
default control group specifier 374
default focus 383
deleting
action points 134
denormalized count array
statistic 190
dereferencing
automatic 210
pointers 210
-detect_leaks 558
dfocus command
example 384
dga command 308
dgo command 325, 329, 379
dgroups command
-add 381, 382
dhold command 121
dialogs
preferences 4
difference operator 380
disabling
action points 134
-display_specifiers command-line
option 556
displaying
areas of memory 210
local variables 146
memory 210
Dive In All command 180, 182
diving 326
on action points 134
dkill command 249, 258
dload command 42, 248, 258

- returning process ID 250
- dlopen
 - and RTLD_DEEPBIND on Linux 571
- dnx command 386
- docking and undocking views 5
- documentation
 - Help view 18
- Documents view 60, 83
- double_dealloc event 557
- \$double_precision data type 177
- dout command 386
- dpid 257
- dprint command 210, 262
 - using with CAF 315
- drawers
 - opening and closing 5
 - Processes and Threads
 - attributes 12, 212
- drerun command 248
- drun command 248, 252
- dsession command 41
- dset command 252, 254
- dstatus
 - thread options 223
- dstatus command 124
- dstep command 372, 373, 379, 386
- DUID 577
 - of process 577
- \$duid built-in variable 577
- dunhold command 121
- dunset command 252
- duntil command 386
- dup commands 210
- dwhere command 210, 373, 379
- dynamic linker 592
- dynamically allocate space 519
- dynamically linked program 592

E

- effects of parallelism on debugger
 - behavior 256
- enabling watchpoints 113
- entering input into the UI 61
- env, inserting agent 565

- environment variables
 - adding 45
 - before starting poe 324
 - for reverse connect 269
 - how to enter 45
 - MP_ADAPTER_USE 324
 - MP_CPU_USE 324
 - TOTALVIEW 318
- errors
 - using ReplayEngine with Infini-
band MPIs 333
- Evaluate Window
 - expression system 108
- evaluating state 259
- evaluation points
 - C constructs 580
 - defined 140, 259
 - examples 105
 - Fortran constructs 581
 - patching programs 106
 - setting 263
- evaluation system limitations 109
- evaluation, see also expression
system
- event_action command-line
option 557
- examples
 - basic debugging 56
 - expression 56
- exec() 38
- EXECUTABLE_PATH variable 31, 34,
261
 - setting 260
- executables
 - removing debug info 587
- execution
 - controlling 258
- execve() 97, 98
 - setting breakpoints with 98
- existent operator 380
- expression
 - basic debugging tutorial 56
- expression system
 - and arrays 147
 - array elements 147
 - C/C++ declarations 580
 - C/C++ statements 580
 - defined 147

- Fortran 581
- Fortran intrinsics 582
- functions and their issues 576
- structures 147
- templates and limitations 580
- Tools > Evaluate Window 108

- expressions 379
- \$extended data type 177
- extending a block 520

F

- F1 key for help 18
- Fermi GPU, compiling for. See CU-
DA, compiling for Fermi
- File > Manage Sessions
 - command 49
- File menu 28
- files
 - .rhosts 324
 - hosts.equiv 324
 - searching for 15
- finding deallocation problems 519
- finding heap heap allocation
problems 519
- first thread indicator of < 372
- \$float data type 177
- focus
 - as list 376
 - changing 384
 - pushing 384
 - restoring 384
- fork_loop.tvd example
program 247
- fork() 38, 97
 - setting breakpoints with 97
- Fortran
 - CoArray support 314
 - compiling on x86-64 585
 - expression system 581
 - in evaluation points 581
 - intrinsics in expression
system 582
- Fortran casting for Global
Arrays 308, 309
- Fortran, tracking memory 512
- free not allocated problems 519
- free_not_allocated event 557

Function attribute definition 225
function calls, in eval points 107
functions
 in expression system 576
 searching for 15

G

-g 523
-g -G option, for compiling CUDA program. See CUDA, -g -G option.
-g -G option, for compiling ROCm HIP program. See ROCm, -g -G option.
-g option
 compiling for debugging 56
g width specifier 376, 382
\$GA cast 308, 309, 308
\$ga cast 308, 309
Global Arrays 308
 casting 308, 309
 diving on type information 308
global variables
 adding to Data View 169
gnu_debuglink 587
gnu_debuglink command-line option 589
gnu_debuglink variable 588
gnu_debuglink_global_directory command-line option 588
gnu_debuglink_global_directory variable 588
Go command 24, 67, 325, 328, 329, 388
GOI
 defined 361
GPU. See CUDA.
Group > Attach Subset
 command 328, 329, 330
Group > Go command 98, 126, 325
Group > Kill command 332
group aliases 255
 limitations 255
Group by
 thread name 221
Group menu 23

group name 375
group number 375
Group of Interest (GOI) 361
group stepping 388
group syntax 374
 naming names 375
GROUP variable 382
group width
 action points 132
 of action points 128
group with the focus
 see GOI 361
groups
 adding an Attach to Program debug session 39
 behavior 388
 defined 364
 in Processes and Threads view 214
-guard_blocks command-line option 558
guard_corruption event 557
GUI namespace 253

H

Halt command 24, 388
header section 510
heap
 defined 519
heap allocation problems 519
heap API 519
 stopping allocation when misused 519
heap debugging
 attaching to programs 565
 environment variable 565
 IBM PE 569
 interposition
 defined 512
 linker command-line options 562
 MPICH 569
 tvheap_mr.a library 567
heap library functions 512
held operator 380
held processes, defined 120

Help menu 18
Help view 18
 F1 key 18
hexadecimal address, specifying in variable window 210
-hoard_freed_memory command-line option 558
hoarding
 finding a multithreading problem 547
 finding dangling pointer references 547
\$hold built-in function 579
hold state
 toggling 121
holding and advancing processes 258
holding problems 126
\$holdprocess built-in function 579
\$holdprocessall built-in function 579
\$holdstopall built-in function 579
\$holdthread built-in function 579
\$holdthreadstop built-in function 579
\$holdthreadstopall built-in function 579
\$holdthreadstopprocess built-in function 579
Hostname attribute definition 225
hosts.equiv file 324
hung processes
 debugging session 35

I

I state indicator 37
IBM MPI 323
IBM SP machine 318, 319
idle state indicator 37
IDs
 action points 86
image file, stripped copy 587
inconsistent widths 378
Infiniband MPIs
 possible errors 333
 settings 333

- with ReplayEngine 332
- infinity count array statistic 190
- Info view 71
- initial process 256
- initializing an array slice 261
- initializing the CLI 247
- Input/Output view 17, 61
- \$int data type 177
- \$integer_2 data type 177
- \$integer_4 data type 177
- \$integer_8 data type 177
- interactive CLI 244
- interface, new
 - docking and undocking 5
 - drawers 5
 - layout 4
 - main views 4
 - resizing behavior 5
 - Source views 6
 - Start Page 6
- internal counter 106
- interposition defined 512
- interrupting commands 246
- intersection operator 380
- intrinsics
 - \$newval 114
 - \$oldval 114
- invoking CLI program from shell
 - example 247
- invoking TotalView on UPC 310
- IP over the switch 324
- iterating
 - over a list 378
 - over arenas 371

K

- Kernel TID attribute definition 226
- Kill command 24, 388

L

- L lockstep group specifier 375, 376
- LD_BIND_NOW environment variable 592
- LD_LIBRARY_PATH environment variable 310
- LD_PRELOAD heap debugging en-

- vironment variable 565
- leaks
 - concealed ownership 521
 - custody changes 521
 - defined 520
 - orphaned ownership 521
 - underwritten destructors
 - leaks 521
 - why they occur 520
- leaks and dangling pointers
 - compared 518
- LIBPATH and linking 567
- libraries
 - debugging SHMEM library
 - code 309
 - shared 591
- limitations
 - AMD (ROCm) and ReplayEngine 500
 - CUDA and ReplayEngine 476
- limitations in evaluation
 - system 109
- LINES_PER_SCREEN variable 251
- Linux-PowerLE 118
- list_allocations action 558
- list_leaks action 558
- lists with inconsistent widths 378
- lists, iterating over 378
- load file 509
- load_session flag 54
- loading
 - action points 144
 - file into debugger 53
- loading sessions
 - Session Editor 41
- local variables
 - in the Local Variables view 70
- Local Variables view
 - adding variables to Data View 15
 - displaying 146
 - viewing variables at breakpoint 69
- lockstep group
 - defined 364
 - L specifier 375
- Logger 16, 60, 61

- \$logical data type 177
- \$logical_1 data type 177
- \$logical_2 data type 177
- \$logical_4 data type 177
- \$logical_8 data type 177
- \$long data type 177
- \$long_long data type 177
- Lookup File or Function view 81
- Lookup view 15
- loop counter 106
- lower adjacent array statistic 189

M

- machine code section 510
- make_actions.tcl sample
 - macro 263
- MALLOCTYPE heap debugging environment variable 565, 568
- Manage Debugging Sessions window
 - accessing 49
- managing sessions
 - from File menu 49
- maximum array statistic 189
- maxruntime command-line option 558, 559
- mean array statistic 189
- median array statistic 189
- memory
 - maps 508
 - pages 508
- Memory Debugger
 - functions tracked 512
- memory debugging
 - batch scripts 556
 - enable 523
 - event reports, generating 538
 - heap report 531
 - heap reports, generating 531
 - HIA 512
 - leak detection 525
 - leak reports 526, 532
 - leak reports, enabling 525
 - leak reports, MPI
 - programs 528
- memory leak
 - defined 520

- memory, displaying areas of 210
- memscript memory debugging
 - script 556
- menus
 - Group 23
 - Help 18
 - Local Variables context
 - menu 15
 - Process 23
 - show/hide views 282
 - Thread 23
- message queue display 329, 332
- messages from debugger,
 - saving 250
- minimum array statistic 189
- missing TID 372
- mixing arena specifiers 379
- more processing 251
- more prompt 251
- MP_ADAPTER_USE environment
 - variable 324
- MP_CPU_USE environment
 - variable 324
- MP_TIMEOUT 324
- MPI
 - attaching to 329, 330
 - Infiniband, using with
 - ReplayEngine 332
 - memory debugging leak
 - reports 528
 - on IBM 323
 - on SGI 328
 - on Sun 330
 - starting on Cray 323
 - starting on SGI 329
 - starting processes 328
 - starting processes, SGI 329
 - troubleshooting 331
- MPI_Init() 326
- MPICH 317, 319
 - and heap debugging 569
 - and SIGINT 332
 - and the TOTALVIEW environment variable 318
 - attach from TotalView 319
 - attaching to 319
 - ch_lfshmem device 317, 320
 - ch_mpl device 317

- ch_p4 device 317, 320
- ch_shmem device 320
- ch_smem device 317
- diving into process 319
- mpirun command 318
- naming processes 321
- obtaining 317
- P4 320
 - p4pg files 320
 - starting TotalView using 318
 - tv command-line option 318
 - using -debug 332
- mpirun command 318, 329
- mpirun process 329
- MPL_Init() 326
 - and breakpoints 326
- mprun command 330, 331
- multi-threaded core files 40

N

- namespaces 253
 - TV:: 253
 - TV::GUI:: 253
- naming MPICH processes 321
- naming threads 221
- NaNs
 - array statistic 190
- navigating
 - to functions 78
- navigation
 - diving and searching 61
 - using disabled action
 - points 136
- Next command 24, 64, 388
- \$nid built-in variable 577
- no_dynamic command-line
 - option 592
- node ID 577
- nodes, attaching from to poe 326
- nonexistent operators 380
- non-invasive 513
- non-sequential program
 - execution 246
- nvcc compiler, and CUDA. See CUDA, nvcc compiler.
- NVIDIA. See CUDA.

O

- \$oldval built-in variable 577
- omitting period in specifier 376
- omitting width specifier 376
- Open MPI
 - starting 327
- opening a core file 42
- operators
 - difference 380
 - & intersection 380
 - | union 379
 - breakpoint 380
 - existent 380
 - held 380
 - nonexistent 380
 - running 380
 - stopped 380
 - unheld 380
 - watchpoint 380
- options, for compiling CUDA. See CUDA, compiling options
- options, for compiling ROCm HIP programs. See ROCm, compiling options
- orphaned ownership 521
- Out command 24, 64, 388
- outliers 190
- output
 - assigning output to
 - variable 250
 - from CLI 250
 - only last command executed
 - returned 250
 - printing 250
 - returning 250
 - when not displayed 250

P

- p width specifier 377
- p.t notation 372
- p/t sets
 - expressions 379
 - syntax 372
- p/t syntax, group syntax 374
- p4 listener process 320
- p4pg files 320
- p4pg option 320

- parallel environments, execution
 - control of 258
- parallel program, defined 256
- parallel tasks, starting 325
- parallel_configs variable 334, 335
- parallel_support.tvd file 335
- parameter values
 - returning 515
- parameters
 - by reference 517
 - by value 517
- parsing comments example 263
- passing arguments 54
- passing default arguments 252
- passing pointer to memory to lower frames 516
- passing pointers 516
- patching
 - function calls 107
 - programs 106
- PATH environment variable
 - entering for debugging session 31
- pathnames, setting in procgroup file 320
- PC attribute definition 225
- PE 326
 - adapter_use option 324
 - applications 323
 - cpu_use option 324
 - from command line 324
 - from poe 324
 - options to use 324
 - switch-based
 - communication 324
- PE applications 324
- pending breakpoint, creating 91
- phase, UPC 313
- \$pid built-in variable 577
- pid specifier, omitting 376
- poe
 - and mpirun 319
 - and TotalView 325
 - arguments 324
 - attaching to 326
 - on IBM SP 320
 - placing on process list 327
 - required options to 324
 - running PE 324
 - TotalView acquires poe processes 326
- POI
 - defined 361
- pointers
 - as arrays 210
 - chasing 210
 - dangling 517
 - dereferencing 210
 - dereferencing an array of pointers 184
 - passing 516
 - realloc problem 520
- pointer-to-shared UPC data 312
- predefined data types 176
- preferences 4
- preferences toolbar 12
- preload variables, by platform 565
- preloading 512
- primary thread, stepping
 - failure 389
- printing an array slice 262
- Procedure Linkage Table (PLT) 592
- process
 - ID 577
 - numbers are unique 256
 - stepping 389
 - width specifier, omitting 376
- Process > Go command 126, 328, 329
- Process > Out command 386
- Process > Run To command 386
- process barrier breakpoint
 - changes when clearing 125
 - changes when setting 125
 - setting 121
- process DUID 577
- process groups
 - stepping 388, 389
- Process Held attribute
 - definition 225
- Process ID attribute definition 225
- Process menu 23
- Process of Interest (POI) 361
- Process State attribute
 - definition 225
- process width
 - action points 129
 - of action points 128
- process with the focus
 - see POI 361
- process_id.thread_id 372
- process/set threads
 - saving 372
- process/thread identifier 256
- process/thread notation 256
- process/thread sets 257
 - as arguments 371
 - changing focus 384
 - inconsistent widths 378
 - structure of 372
 - widths inconsistent 378
- \$processduid built-in variable 577
- processes
 - acquiring 319, 320
 - acquisition in poe 326
 - attaching to 35, 326
 - attribute definitions 225
 - barrier point behavior 125
 - copy breakpoints from master process 319
 - creating new 248
 - diving into 326
 - held defined 120
 - holding 120, 579
 - hung, debugging 35
 - in ptlist format 215
 - initial 256
 - released 120
 - releasing 120, 125
 - single-stepping 386
 - slave, breakpoints in 319
 - spawned 256
 - state and action points 129
 - stopped 120
 - stopped at barrier point 125
 - stopping intrinsic 579
 - stopping spawned 319
 - synchronizing 259
 - synchronizing with Run To 391
 - terminating 249
 - viewing and controlling 214
 - when stopped 389
- Processes and Threads view 12, 59

- ascending/descending column order 216
- attribute definitions 225
- attributes 12, 214
- different views of the attributes 217
- effect on other views 220
- main description 214
- tabular presentation 215
- tree presentation 212
- procgroup file 320
 - using same absolute path names 320
- Program arguments
 - in Debug New Program dialog 32
- program execution
 - advancing 258
 - controlling 258
- Program Session dialog 31, 33, 303
- program state, changing 246
- program, mapping to disk 508
- programs
 - compiling 56, 561
 - correcting 108
 - patching 106
- prompt and width specifier 378
- PROMPT variable 254
- prun command 328
- pthread ID 257
- ptlist format 215
- pushing focus 384

Q

- QSW RMS applications
 - attaching to 328
 - starting 327
- quartiles array statistic 189

R

- R state indicator 37
- RDMA optimizations
 - disabled with Infiniband 333
- \$real data type 177
- \$real_16 data type 177
- \$real_4 data type 177
- \$real_8 data type 177

- realloc not allocated problems 519
- realloc pointer problem 520
- realloc problems 520
- realloc_not_allocated event 557
- reallocation timing 517
- recording file
 - configuring with ReplayEngine 39
- reference counting 521
- relatives
 - attaching to 38
- reloading breakpoints 325
- remote login 324
- replacing default arguments 252
- Replay Mode attribute definition 226
- replay recording file debug session 39
- ReplayEngine
 - AMD limitations 500
 - and Infiniband MPIs 332
 - configuring in Session Manager 39
 - CUDA limitations 476
- ReplayEngine toolbar 12
- reports
 - when to create 524
- resetting command-line arguments 45
- resizing interface views 5
- Restart command 24, 388
- restarting
 - program execution 248
- restoring focus 384
- results, assigning output to variables 250
- returning parameter values 515
- reverse connect
 - concepts 267
 - environment variables 269
 - examples 273
 - starting a session 271
- reverse connections
 - tvconnect 266
- Reverse debugging
 - debug options 32, 35

- RMS applications
 - attaching to 328
 - starting 327
- ROCm
 - assigned thread IDs 491
 - compiling options 488
 - data from ROCm work-item, displaying 493
 - execution, viewing 490
 - g -G compiling option 488
 - host thread, viewing 492
 - limitations with ReplayEngine 500
 - starting TotalView for ROCm applications 488
 - troubleshooting 499
 - variables from ROCm work-item, displaying 493
- rsh command 324
- RTLD_DEEPCBIND
 - and dlopen on Linux 571
- Run To command 24, 64, 388
 - used to synchronize 391
- running operator 380
- running out of memory 520
- running state indicator 37

S

- s command-line option 247
- S share group specifier 374
- S state indicator 37
- S width specifier 376
- sane command argument 247
- satisfaction set 123
- satisfied barrier 123
- save_html_heap_status_
source_view action 558
- save_memory_debugging_file
action 558
- save_text_heap_status_
source_view action 558
- saving
 - action points 144
 - messages 250
- scope
 - of debugging commands 23
- scripting, memory debugging 556

- scrolling
 - output 251
- search
 - for an existing session 50
 - Lookup view 15
- searching for files and functions 15
- sections
 - data 509
 - header 510
 - machine code 510
 - symbol table 510
- Session Editor
 - recent sessions 41
- Session Manager
 - editing previous sessions 41
- sessions
 - configure new debugging 31
 - deleting duplicates 51
 - editing 41
 - loading an existing 51
 - loading using -load_session flag 54
 - loading using the CLI 41
 - managing 50
 - Start Page 21
 - starting 21, 58
- set expressions 379
- set indicator, uses dot 372, 381
- setting
 - breakpoints 88, 263, 325
 - breakpoints while running 88
 - thread specific
 - breakpoints 577
 - timeouts 324
- settings
 - for use of Infiniband MPis and ReplayEngine 333
- SGROUP variable 382
- Share Group attribute
 - definition 225
- share groups
 - defined 364
 - S specifier 374
- SHARE_ACTION_POINT variable 97
- shared libraries 591
- shell, example of invoking CLI program 247
- SHMEM library code
 - debugging 309
- \$short data type 178
- show_backtrace item 556
- show_backtrace_id item 556
- show_block_address item 556
- show_flags item 556
- show_guard_details item 556
- show_guard_id item 556
- show_guard_status item 556
- show_image item 556
- show_leak_stats item 556
- show_pc item 556
- show_pid item 556
- showing areas of memory 210
- side 576
- side-effects of functions in expression system 576
- SIGINT signal 332
- signals
 - that caused core dump 40
- single-stepping
 - on primary thread only 386
- slash in group specifier 375
- sleeping state indicator 37
- slices
 - UPC 310
- slicing arrays 196
- sliding breakpoints 90
- SLURM 304
- SMP machines 318
- sorting
 - action points 133
- Source Line attribute
 - definition 225
- Source views 6, 60
 - responding to Processes and Threads view actions 220
- source-level breakpoints 88
- space, dynamically allocating 519
- spawned processes 256
 - stopping 319
- specifier combinations 376
- specifiers
 - and dfocus 378
 - and prompt changes 378
 - example 382
 - examples 378, 393
- specifying groups 374
- stack frame 210
- stack frames 516
 - arranging 514
- stack memory 514, 517
- stack, compared to data section 514
- stack frame
 - data block 515
- standard deviation array
 - statistic 189
- standard I/O
 - altering in Session Manager 45
 - redirecting 58
- Start Page 6, 21, 28, 30
- starting 308
 - CLI 247
 - parallel tasks 325
 - TotalView 324
- starting a debugging session 21
- starting Open MPI programs 327
- state
 - of action points 136
 - process/thread state and action points 129
- static internal counter 106
- status bar, displaying state 63
- Step command 24, 62, 388
- stepping
 - see also* single-stepping
 - at process width 389
 - at thread width 389
 - primary thread can fail 389
 - process group 388, 389
 - target program 258
 - thread group 389
 - threads 392
 - tutorial 62
 - workers 392
- stepping a group 388
- \$stop built-in function 579
- stop function 265
- Stop Group
 - action point width setting 128
- Stop Process

- action point width setting 128
- Stop Reason attribute
 - definition 225
- Stop Thread
 - action point width setting 128
- stop, defined in a multiprocess
 - environment 258
- \$stopall built-in function 579
- stopped operator 380
- stopped process 125
- stopped state indicator 37
- stopping
 - spawned processes 319
- stopping execution on heap API
 - misuse 519
- \$stopprocess built-in function 579
- \$stopthread built-in function 579
- storage qualifier for CUDA. See CUDA, storage qualifier
- strdup allocating memory 519
- \$string data type 178
- stripped copy 587
- structures
 - expression evaluation 147
- stty sane command 247
- sum array statistic 189
- Sun MPI 330
- suppressing action points 136
- switch-based communication 324
 - for PE 324
- symbol table section 510
- synchronizing execution 390
- synchronizing processes 259
- system PID 257
- system TID 257
- System TID attribute definition 226
- systid 257
- \$systid built-in variable 577

T

- T state indicator 37
- t width specifier 377
- tabular presentation of process
 - and thread attributes 215
- target process/thread set 258

- target program
 - stepping 258
- target, changing 384
- tasks
 - starting 325
- Tcl
 - CLI and thread lists 245
 - version based upon 245
- templated code
 - and breakpoints 89
- templates
 - expression system 580
- terminal window
 - starting CLI from 247
- terminating processes 249
- termination_ notification event 557
- Tesla GPU, compiling for. See CUDA, Tesla GPU.
- thread
 - width specifier, omitting 376
- thread group
 - stepping 389
- Thread Held attribute
 - definition 226
- Thread ID
 - attribute definition 225
- thread ID
 - about 257
 - assigned to CUDA threads. See CUDA, assigned thread IDs.
 - assigned to ROCm threads. See ROCm, assigned thread IDs.
 - system 577
 - TotalView 577
- Thread Index attribute
 - definition 225
- Thread menu 23
- Thread Name attribute
 - definition 226
- thread names
 - setting 222
- thread numbers are unique 256
- thread of interest 371, 372
 - defined 372
- Thread of Interest (TOI) 361

- Thread State attribute
 - definition 225
- thread stepping 392
 - platforms where allowed 389
- thread width
 - of action points 128
- thread with the focus
 - see TOI 361
- thread_name properties 223
- threads
 - attribute definitions 225
 - displaying names 221
 - holding 122
 - in ptlist format 215
 - releasing 120
 - setting breakpoints in 577
 - single-stepping 386
 - state and action points 129
 - synchronizing with Run To 391
 - viewing and controlling 214
 - width specifier 373
- thread-specific breakpoints 577
- tid 257
- \$tid built-in variable 577
- TID missing in arena 372
- timeouts
 - during initialization 326
 - TotalView setting 324
- timeouts, setting 324
- timing of reallocations 517
- TOI
 - defined 361
- toolbars
 - described 12
 - showing 4
 - turning on text 4, 12
- Tools > Evaluate Window
 - expression system 108
- Tools > View Across command 313
- Tools > Visualize Distribution
 - command 312
- Tools > Watchpoint command 116
- tooltips 71
 - viewing variable values 152
- TotalView
 - and MPICH 318
 - invoking on CAF 314

- invoking on UPC 310
 - starting 324
- totalview command 329
- TOTALVIEW environment
 - variable 318, 319
- totalviewcli command 247, 248, 329
- tree presentation of process and
 - thread attributes 212
- troubleshooting
 - MPI 331
- TV
 - mrnet_super_bushy 403
 - tv command-line option 318
 - TV_REVERSE_CONNECT_DIR, env.
 - variable 269
 - TV:: namespace 253
 - TV::GUI:: namespace 253
 - tvconnect
 - reverse connection
 - command 266
 - TVCONNECT_OPTIONS, env.
 - variable 269
 - tvdsrv
 - editing command line for
 - poe 326
 - fails in MPI environment 332
 - TVHEAP_DEEPBIND
 - controlling RTLD_DEEPBIND on
 - Linux 572
 - tvheap_mr.a
 - aix_install_tvheap_mr.sh
 - script 567
 - and aix_malloctype.o 568
 - creating using poe 567
 - dynamically loading 567
 - libc.a requirements 567
 - pathname requirements 567
 - relinking executables on
 - AIX 568
 - tvheap_mr.a library 567
 - type strings
 - built-in 176

U

- underwritten destructors 521
- unheld operator 380

- union operator 379
- unique process numbers 256
- unique thread numbers 256
- unsuppressing action points 136
- UPC
 - assistant library 310
 - phase 313
 - pointer-to-shared data 312
 - shared scalar variables 310
 - slicing 310
 - starting 310
 - viewing shared objects 310
- UPC debugging 310
- upper adjacent array statistic 190
- User_TID attribute definition 226
- using env to insert agent 565

V

- values
 - updated in DataView 74
- variables
 - adding to Data View 72
 - assigning p/t set to 372
 - CGROUP 381
 - global, viewing 169
 - GROUP 382
 - setting command output
 - to 250
 - SGROUP 382
 - values update in Data View 72
 - viewing value at breakpoint 69
 - WGROUP 381, 382
- VERBOSE variable 246
- verbosity level 329
- View > Dive In All command 182
- viewing shared UPC objects 310
- Views
 - Documents 60
- views
 - Action Points 16
 - CLI 16
 - Command Line 16
 - Data 15
 - docking and undocking 5
 - Help 18
 - initial layout in interface 4
 - Logger 16
 - Lookup 15

- Processes and Threads 12, 214
 - showing and hiding 282
- \$visualize 579
- \$void data type 178
- Volta, compiling for 448

W

- W width specifier 376
- watching memory 114
- Watchpoint command 116
- watchpoint operator 380
- watchpoints
 - \$newval watchpoint
 - variable 116
 - \$oldval 116
 - alignment 117
 - conditional 116
 - copying data 116
 - defined 259
 - enabling 113
 - example of triggering when val-
 - ue goes negative 117
 - length compared to \$oldval or
 - \$newval 117
 - lowest address triggered 115
 - modifying a memory
 - location 110
 - monitoring adjacent
 - locations 115
 - multiple 115
 - on stack variables 112
 - PC position 115
 - platform differences 118
 - problem with stack
 - variables 114
 - supported platforms 118
 - triggering 110, 115
 - unconditional 111
 - watching memory 114
- WGROUP variable 381, 382
- When Done, Stop radio
 - buttons 123
- When hit
 - action point width
 - property 128
- When Hit, Stop radio buttons 122
- width
 - action points, default 129
 - action points, explained 128

- action points, group width 132
- action points, process
 - width 129
- action points, thread width 131
- in debugging commands 23
- width specifier 371
 - omitting 376
- workers group
 - defined 364
- workers group specifier 375
- writing array data to files 263

Z

- Z state indicator 37
- zero count array statistic 189
- Zombie state indicator 37

