ACKNOWLEDGMENTS

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

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

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

Rogue Wave Software, Inc.

Product Information: (303) 473-9118 (800) 487-3217
Fax: (303) 473-9137
Web: http://www.roguewave.com
Contents

Part 1: An Introduction to NextGen TotalView for HPC

Chapter 1  Getting Started  2
Introducing NextGen TotalView for HPC  ................................................ 2
An Initial Look at the Interface ................................................................. 4
  Customizing the Interface ................................................................. 4
  Preferences ................................................................. 4
  Resizing ................................................................. 4
  Drawers ................................................................. 5
  Undocking and Docking ............................................................. 5
A Tour of the Interface ................................................................. 6
  Central Window ................................................................. 6
  Toolbars ................................................................. 9
  Processes and Threads View ......................................................... 10
  Call Stack View and Local Variables (VAR) Drawer ...................... 11
  Data View ................................................................. 12
  Lookup View ................................................................. 12
  Action Points, CLI, and Logger Views ........................................ 13
  Help ................................................................. 14
Starting TotalView and Creating a Debugging Session .............. 17
  Debugging Commands ............................................................. 19

Chapter 2  Creating and Managing Sessions  21
Overview ................................................................. 21
Setting up Debugging Sessions ...................................................... 23
  Loading Programs from the Session Editor .................................... 23
    Starting a Debugging Session .................................................. 24
  Debug a Program ................................................................. 25
  Debug a Parallel Program ....................................................... 26
  Attach to Process ................................................................. 29
  Debug a Core or Replay Recording File ..................................... 33
  Load a Recent Session ............................................................. 34
  Editing a Previous Session ........................................................ 35
Chapter 3  

**Basic Debugging**  

Overview ........................................... 47  

Program Load and Navigation ..................... 49  

Load the Program to Debug ......................... 49  

Initial Display ........................................ 51  

Program Navigation ................................... 52  

Navigating from within the Source Pane ........... 53  

Searching with Highlighting and the Find Function 53  

Using the Lookup File or Function View ............ 55  

Stepping and Executing ............................ 58  

Simple Stepping ...................................... 58  

Setting and Running to a Breakpoint (Action Point) 61  

Set and Control Breakpoints ......................... 61  

Run Your Program and Observe the Call Stack .... 63  

Examining Data ...................................... 65  

Viewing Variables in the VAR Drawer .............. 65  

Viewing Variables in the Data View ................. 66  

Watching Data Values Update ....................... 67  

Moving On ........................................... 71  

Part 2: Debugging Tools and Tasks  

Chapter 4  

**Viewing and Organizing Processes and Threads**  

The Processes and Threads View .................... 73  

How the Processes and Threads View Works ........ 78  

Relationship of the Process and Threads View to Other Views 78  

Debugging Command Width ......................... 78  

Understanding Thread Width ....................... 79  

Understanding Process Width ...................... 80  

Understanding Group Width ......................... 80  

Other Considerations When Stepping Through Code 80
Known Limitations and Issues ................................................. 231
  Limitations ................................................................. 231
  Performance Issues ....................................................... 232

Chapter 11 Preferences ..................................................... 234
  About Preferences .......................................................... 234
  Display Settings ............................................................ 235
  Tool Bar ................................................................. 237
  Search Path ............................................................... 238
  Parallel Configuration .................................................. 241

Part 3: Using the CUDA Debugger

Chapter 12 About the TotalView CUDA Debugger .................. 244
  Overview ................................................................. 244
  Installing the CUDA SDK Tool Chain .............................. 246
  Directive-Based Accelerator Programming Languages .......... 247

Chapter 13 CUDA Debugging Model and Unified Display ........... 248
  The TotalView CUDA Debugging Model ............................ 249
  Pending and Sliding Breakpoints .................................... 251
  Unified Source View and Breakpoint Display ..................... 252

Chapter 14 CUDA Debugging Tutorial .................................. 254
  Compiling for Debugging .............................................. 254
    Compiling for Fermi .................................................. 254
    Compiling for Fermi and Tesla .................................... 255
    Compiling for Kepler ............................................... 255
    Compiling for Pascal ................................................ 255
    Compiling for Volta .................................................. 255
  Starting a TotalView CUDA Session ................................. 256
  Controlling Execution .................................................. 257
    Viewing GPU Threads ................................................. 257
    CUDA Thread IDs and Coordinate Spaces ......................... 258
    Single-Stepping GPU Code ......................................... 259
    Halting a Running Application ................................... 260
  Displaying CUDA Program Elements ............................... 261
    GPU Assembler Display ............................................. 261
    GPU Variable and Data Display ................................... 261
CUDA Built-In Runtime Variables ........................................ 262
Type Casting ................................................................. 263
PTX Registers ............................................................... 266
Enabling CUDA Memory Checker Feature .......................... 267
GPU Core Dump Support .................................................. 268
GPU Error Reporting ....................................................... 269

Chapter 15  CUDA Problems and Limitations ....................... 272
Hangs or Initialization Failures ............................................ 272
CUDA and ReplayEngine ................................................. 274

Chapter 16  Sample CUDA Program ................................. 275

Part 4: Appendices

Appendix A  More on Expressions ................................. 281
Overview ................................................................. 281
Calling Functions: Problems and Issues .............................. 281
Using Built-in Variables and Statements .............................. 283
  Using TotalView Variables ............................................. 283
  Using Built-In Statements ............................................. 284
Using Programming Language Elements ............................ 286
  Using C and C++ ...................................................... 286
  Using Fortran ......................................................... 287
    Fortran Statements ................................................ 287
    Fortran Intrinsics ................................................. 288

Appendix B  Compiling for Debugging ......................... 290
Overview ................................................................. 290
Compiling with Debugging Symbols .................................. 290
Maintaining Debug Information Separate from an Executable 293
  Controlling Separate Debug Files ................................ 294
  Searching for the Debug Files ..................................... 294

Appendix C  Platform-Specific Topics ......................... 296
Overview ................................................................. 296
Swap Space ............................................................. 296
Shared Libraries .......................................................... 297
  Changing Linkage Table Entries and LD_BIND_NOW .......... 298
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linking with the dbfork Library</td>
<td>298</td>
</tr>
<tr>
<td>Linux or Mac OS X</td>
<td>298</td>
</tr>
<tr>
<td>Appendix D</td>
<td></td>
</tr>
<tr>
<td><strong>Resources</strong></td>
<td>300</td>
</tr>
<tr>
<td>TotalView for HPC Documentation</td>
<td>300</td>
</tr>
<tr>
<td>Conventions</td>
<td>302</td>
</tr>
<tr>
<td>Contacting Us</td>
<td>302</td>
</tr>
<tr>
<td>Index</td>
<td>303</td>
</tr>
</tbody>
</table>
An Introduction to NextGen TotalView for HPC

“Getting Started” on page 2
An introduction to the primary features and interface of TotalView.

“Creating and Managing Sessions” on page 21
How to create a new session or load a previous session.

“Basic Debugging” on page 47
A tutorial based on a shipped example that illustrates basic debugging tasks.
Chapter 1

Getting Started

Introducing NextGen TotalView for HPC

Rogue Wave NextGen TotalView for HPC is the next generation user interface for the TotalView debugger. Behind the NextGen TotalView for HPC user interface is the TotalView debugging engine, the long standing industry leader for debugging massively parallel programs in the HPC arena.

Currently, the next generation UI is supported on Linux x86 64-bit, macOS, Linux PowerLE and Linux ARM64 platforms. It supports multi-process and multi-threaded debugging, and MPI and CUDA debugging.

Note that 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. Depending on the tools you use, the functionality may not appear in the user interface yet, however.

NextGen TotalView for HPC 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.

NextGen TotalView for HPC 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.

Each new release will include additional functionality based on a priority list that you can help influence. Please send email to tv-beta@roguewave.com with your feedback and feature priorities.

New UI Limitations

HPC functionality from the classic user interface continues to be added to the new UI. These are the major areas of CLI functionality not yet supported by the UI:
• Remote debugging
• Some advanced data debugging such as array manipulation and visualization
• Memory debugging with MemoryScape

UI Preferences

For new TotalView users, the new UI is the default. 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 Rogue Wave, conventions used in the documentation, and documentation for TotalView for HPC, see Appendix D, “Resources,” on page 300.
An Initial Look at the Interface

Starting NextGen TotalView for HPC 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 Chapter 11, “Preferences,” on page 234.

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.

<table>
<thead>
<tr>
<th>Drawer Open</th>
<th>Drawer Closed</th>
</tr>
</thead>
</table>

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.
To return a floating window to its default position, double-click on its banner or click the reattach icon \( \text{reattach} \) in the top-right corner next to the close (\( \times \)) 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 Window**

When you first start up NextGen TotalView for HPC, the central window contains either just the Start Page, or the Start Page and a Source view if you started NextGen TotalView for HPC with an executable name argument. This area is reserved for displaying the Start Page, Source views of code, and the Help view.

This central window cannot be closed, and other views cannot be brought into this area as tabs.
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.)

- 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.

Source view actions

- Create breakpoints by clicking on bold line numbers in the gutter.
• **See variable information** by hovering over a variable name.

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

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

If you highlight the function name and select “Navigate to File or Function” from the context menu, the NextGen UI finds and displays the source for the function, if the source is available. If there is more than one source location, the NextGen UI 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.
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” on page 87

Source views and their relationship to data display  
“Relationship of the Process and Threads View to Other Views” on page 78

**Toolbars**

The UI has the following toolbars:

**NOTE >>** The ReplayEngine toolbar appears only on supported platforms, which are Linux x86-32 and 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 “GPU Toolbars” on page 259.
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 6 – Processes and Threads View
Call Stack View and Local Variables (VAR) Drawer

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”.

Figure 7 – Call Stack View with Local Variables Drawer

RELATED TOPICS

Detailed information on this view
Chapter 4, “Viewing and Organizing Processes and Threads,” on page 73

The thread/process of focus and its effect on the display of data
“Relationship of the Process and Threads View to Other Views” on page 78

RELATED TOPICS

Detailed information on this view
“The Call Stack View and Local Variables” on page 143
**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

**Figure 8 – The Data View**

![Data View](image)

**RELATED TOPICS**

Exchanging, manipulating, and editing data  
*“The Data View” on page 154*

**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 9**, 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 9 – Lookup 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 TotalView for HPC debugging 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.
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.
In Figure 11, F1 was pressed with the cursor over the ReplayEngine toolbar.

The information displayed for context sensitive help is from the full product documentation for NextGen TotalView for HPC. If you move the Help into a separate window and increase its size, at some point navigation for the full product documentation appears.
Figure 12 – The Help Window with Full Product Documentation

Chapter 8 **ReplayEngine**

ReplayEngine is embedded functionality on Linux x86 and x86-64 platforms that allows you to go backwards in a debugging session. (Note that if you are working on a platform that does not support ReplayEngine, the ReplayEngine toolbar and ReplayEngine-related menu items do not appear on the interface.) To make best use of this functionality, it is important to understand precisely how ReplayEngine works and the implied limitations. So the first main section in this chapter describes the internals of this functionality.

For information focused on using ReplayEngine in a debugging session, see Using ReplayEngine.

ReplayEngine works hand in hand with TotalView, so the descriptions in this chapter assume you already have a good understanding of how the TotalView product works.
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 13 – Starting NextGen TotalView for HPC 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-newUI 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 NextGen TotalView for HPC UI with the program loaded and ready to debug.

**Figure 14** 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 14 – Starting NextGen TotalView for HPC with an Executable Name**


## Debugging Commands

The table below summarizes the behavior of the debugging commands available in NextGen TotalView for HPC. 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.

### Related Topics

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page Referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defining, Editing, and Managing Sessions</td>
<td>Chapter 2, “Creating and Managing Sessions,” on page 21</td>
</tr>
<tr>
<td>More on compiling programs for debugging</td>
<td>Appendix B, “Compiling for Debugging,” on page 290</td>
</tr>
<tr>
<td>More on starting NextGen TotalView for HPC</td>
<td>“Starting a Session from your Shell” on page 45</td>
</tr>
</tbody>
</table>

### Command Descriptions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>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.</td>
</tr>
<tr>
<td>Halt</td>
<td>Stops the thread at its current execution point.</td>
</tr>
<tr>
<td>Kill</td>
<td>Stops program execution. Existing breakpoints and other settings remain in effect.</td>
</tr>
<tr>
<td>Restart</td>
<td>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.</td>
</tr>
<tr>
<td>Next</td>
<td>Moves the thread to the next line of execution. If the line the thread was on includes one or more function calls, NextGen TotalView for HPC does not step into these functions but just executes them and returns.</td>
</tr>
<tr>
<td>Step</td>
<td>Like Next, except that NextGen TotalView for HPC does step into any function calls, so the thread stops at the first line of execution of the first function call.</td>
</tr>
<tr>
<td>Out</td>
<td>If the thread is in a block of execution, runs the thread to the first line of execution beyond that block.</td>
</tr>
<tr>
<td>Run To</td>
<td>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.</td>
</tr>
</tbody>
</table>
## RELATED TOPICS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controlling the scope (width) of debugging commands</td>
<td>“Debugging Command Width” on page 78</td>
</tr>
<tr>
<td>Controlling what happens when a thread reaches a breakpoint (action point)</td>
<td>“Controlling an Action Point’s Width” on page 123</td>
</tr>
<tr>
<td>Seeing the debugging commands in action</td>
<td>“Stepping and Executing” on page 58</td>
</tr>
</tbody>
</table>
Creating and Managing Sessions

Overview

There are two primary ways to load programs into NextGen TotalView for HPC 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 NextGen TotalView for HPC 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" on page 23
  - "Starting a Debugging Session" on page 24
  - "Debug a Program" on page 25
  - "Debug a Parallel Program" on page 26
  - "Attach to Process" on page 29
  - "Debug a Core or Replay Recording File" on page 33
  - "Load a Recent Session" on page 34
  - "Editing a Previous Session" on page 35
- "Loading Programs Using the CLI" on page 35
Additional Session Setup Options

- “Options: Reverse Debugging” on page 37
- “Program Environment: Environment Variables” on page 37
- “Standard Input and Output” on page 38

Managing Debug Sessions

- “Managing Sessions” on page 41

Starting NextGen TotalView for HPC with a Session Initiated

- “Starting a Session from your Shell” on page 45
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 NextGen TotalView for HPC. Alternatively, you can use the CLI.

“Loading Programs from the Session Editor” on page 23

“Loading Programs Using the CLI” on page 35

Loading Programs from the Session Editor

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” on page 24), while the File menu enables you to choose a specific debugging session, such as loading local programs, core files, and processes that are already running.

Figure 15 – Choosing a Specific Debug Session from the File Menu

Figure 16 – 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-newUI
```

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

**Figure 17 – Start Page Opening View**

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

- “Debug a Program” on page 25
- “Attach to Process” on page 29
- “Debug a Parallel Program” on page 26
- “Debug a Core or Replay Recording File” on page 33
- “Load a Recent Session” on page 34
Debug a Program

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

**Figure 18 – 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” on page 35.

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**: See “Options: Reverse Debugging” on page 37.

— **Program Environment: Environment variables for the program**: See “Program Environment: Environment Variables” on page 37.

— **Standard Input and Output**: See “Standard Input and Output” on page 38.

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 MPI and MPICH, OpenMP, ORNL SGI shared memory (shmem), Global Arrays, and UPC.

#### 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, you 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 case, you 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 the recommended method and 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 19 – Parallel Session: Session and Program Details

**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” on page 35

**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” on page 238.

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 20 – 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 the “MPI Startup” appendix in the *TotalView for HPC Reference Guide*.)

— **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 “Options: Reverse Debugging” on page 37.


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:

![Parallel Sanity Check Failed](image)

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Chapter/Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set up MPI debugging sessions for various environments and special use cases</td>
<td>Chapter 19, “Setting Up MPI Debugging Sessions” in <em>TotalView for HPC User Guide</em></td>
</tr>
<tr>
<td>Set up non-MPI parallel debugging sessions for applications that use the parallel execution models that TotalView supports</td>
<td>Chapter 20, “Setting Up Parallel Debugging Sessions” in <em>TotalView for HPC User Guide</em></td>
</tr>
<tr>
<td>Create MPI startup profiles for environments that TotalView doesn't define</td>
<td>Appendix A, “MPI Startup” in <em>TotalView for HPC Reference Guide</em></td>
</tr>
<tr>
<td>Tips for debugging parallel applications</td>
<td>Chapter 15, “Debugging Strategies for Parallel Applications” in <em>TotalView for HPC User Guide</em></td>
</tr>
</tbody>
</table>

### 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 21 – Attach to a running program

In the displayed list, processes to which NextGen TotalView for HPC 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” on page 35.

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.

### Field Definitions

The Processes section displays these fields:

- **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:

<table>
<thead>
<tr>
<th>Character and Meaning</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>I (Idle)</td>
<td>Process has been idle or sleeping for more than 20 seconds.</td>
</tr>
<tr>
<td>R (Running)</td>
<td>Process is running or can run.</td>
</tr>
<tr>
<td>S (Sleeping)</td>
<td>Process has been idle or sleeping for less than 20 seconds.</td>
</tr>
<tr>
<td>T (Stopped)</td>
<td>Process is stopped.</td>
</tr>
<tr>
<td>Z (Zombie)</td>
<td>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.</td>
</tr>
</tbody>
</table>

- **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 NextGen TotalView for HPC is running

### RELATED TOPICS

The CLI `dattach` command

`dattach` in the NextGen TotalView for HPC Reference Guide

The CLI `ddetach` command

`ddetach` in the NextGen TotalView for HPC Reference Guide
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 NextGen TotalView for HPC 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” on page 298.

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 “Options: Reverse Debugging” on page 37.)

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.
Creating and Managing Sessions / Setting up Debugging Sessions

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 | replayrecordingsessionfile` option

`dattach` in the *NextGen TotalView for HPC Reference Guide*

**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.
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.

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 -newUI 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 -newUI executable replay-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"
```
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: Reverse Debugging
- Environment Variables
- Standard input or output settings

See “Modifying Arguments in an Open Session” on page 39 for how to set options during an existing session.

Options: Reverse Debugging

Record all program state while running and then roll back your program to any point.

Figure 25 – Debug Options for Reverse Debugging

The Enable reverse debugging with ReplayEngine check box is visible only on Linux-x86 (32-bit) and 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.

RELATED TOPICS
Reverse Debugging
Chapter 10, “ReplayEngine,” on page 217

Program Environment: Environment Variables

When loading the program from within TotalView, add any necessary environment variables into the Environment variables for the program field.
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.
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” on page 45.)

**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**

![Image of Modify Arguments drop-down](image)

The Session Editor launches.
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” on page 35.

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**

What do you want to do today?

- Debug a Program
- Attach To Process
- Load Core or Replay Recording File

**Figure 31 – Accessing Manage Sessions from the File Menu**

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

- **Display data about a session** by selecting the session.

  **Manage Sessions**

  ![Manage Sessions window](image)

  **Session Name: tx_hello_sleep**
  
  **Type:** Attach
  **Program:** tx_hello_sleep
  **Path:** /nfs/fs-353/home/kduthie/test/c/tx_hello_sleep
  **Host:** fedora20-x8664.totalviewtech.com
  **Selected Host:** local
  **Last Run Time:** 06/03/15 08:51:25 am

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

  ![Search for a session](image)
• **Rename a session** by double-clicking on it and entering a new name.

![Sessions Table]

```
<table>
<thead>
<tr>
<th>Sessions</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wave</td>
<td>wave extended</td>
</tr>
</tbody>
</table>
```

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

![Session Name: tx_blocks]

```
Type: Program
Program: tx_blocks
Path: /home/schoose/tests
Selected Host: (local)
Last Run Time: 04/15/15 08:18:04 am
```

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

![Session Icons]

Table 1: Manage Sessions Icons

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>![New Session Icon]</td>
<td>Creates a new debugging session, opening a drop-down menu for selecting:</td>
</tr>
<tr>
<td>![Load Session Icon]</td>
<td>- Create a new Program Session</td>
</tr>
<tr>
<td>![Copy Icon]</td>
<td>- Set up an Attach Session</td>
</tr>
<tr>
<td>![Edit Icon]</td>
<td>- Create a session to load a Core or Replay Recording File</td>
</tr>
<tr>
<td>![Delete Icon]</td>
<td>- Duplicates a session, naming it &quot;&lt;Session Name&gt;Copy&quot;. You can rename and then edit this session.</td>
</tr>
</tbody>
</table>

[473x29] Creating and Managing Sessions / Managing Sessions  43
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 NextGen TotalView for HPC so a session is created and ready to begin when TotalView opens.

**NOTE >>** If you need features currently not supported in the NextGen TotalView for HPC UI (see “Introducing NextGen TotalView for HPC” on page 2), you can launch TotalView by invoking `totalview` without the flag `-newUI`. For example: `totalview`

**NOTE >>** If you have set the environment variable `TVNEWUI` to `True`, omit the flag `-newUI` in the commands below. See “Introducing NextGen TotalView for HPC” on page 2 for more information.

**Debugging a Program**

```
totalview -newUI executable
```
Starts TotalView and loads the `executable` program.

**Debugging a Parallel Program**

```
totalview -newUI -args mpirun -np 4 ./mpi_program
```
Starts TotalView and loads a four-process MPI program.

**Debugging a Core File**

```
totalview -newUI 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 -newUI 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 -newUI 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.

Loading a Session

```
$ totalview -newUI -load_session session-name
```

Starts TotalView and the named session.

RELATED TOPICS

Parallel preferences when debugging a parallel program “Parallel Configuration” on page 241
Chapter 3

Basic Debugging

Overview

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 NextGen TotalView for HPC 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 NextGen TotalView for HPC's primary tools, as follows:

- “Program Load and Navigation” on page 49
• “Stepping and Executing” on page 58
• “Setting and Running to a Breakpoint (Action Point)” on page 61
• “Examining Data” on page 65
Program Load and Navigation

This section discusses how to load a program and looks at the primary NextGen TotalView for HPC 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 TotalView for HPC Installation Guide.

1. Start NextGen TotalView for HPC.
   `totalview -newUI`
   The Start Page launches.
2. Select **Debug a Program** to open the Program Session window.

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-newUI -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” on page 23
- Starting a session from your shell
  - “Starting a Session from your Shell” on page 45
- Modifying arguments in an existing session
  - “Modifying Arguments in an Open Session” on page 39
**Initial Display**

At startup, TotalView displays your program's code in the Source pane, along with its default views: the Processes & Threads, Call Stack, Lookup File or Function, Action Points, 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 **Call Stack view** shows the backtrace of the thread that is currently in focus once the program is running. Part of the Call Stack is the **VAR drawer** which displays information on the current thread's variables.

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

- The **Source view** displays your source code's `main()` function before execution.
Three views are visible at the bottom: Action Points, Command Line and Logger.

- **Action Points**, which will display any breakpoints you set.

- **Command Line**, which, when selected, displays a prompt for entering CLI commands:

![Command Line](image1)

- **Logger** which, when selected, displays logging output from TotalView. The logger also displays any output from your program that is directed to stdout:

![Logger](image2)

- **Data View**, which enables you to evaluate expressions to observe your data while your program is running.

**RELATED TOPICS**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes &amp; Threads view</td>
<td>73</td>
</tr>
<tr>
<td>Call Stack panel</td>
<td>143</td>
</tr>
<tr>
<td>Action Points</td>
<td>85</td>
</tr>
<tr>
<td>The CLI</td>
<td>187</td>
</tr>
</tbody>
</table>

**Program Navigation**

TotalView provides several ways to search your applications for text strings, files or functions.

- **“Navigating from within the Source Pane”** on page 53: Navigate to a function from within the Source pane using the context menu.

- **“Searching with Highlighting and the Find Function”** on page 53. Search in the Source view by highlighting a string or using the Find function.

- **“Using the Lookup File or Function View”** on page 55. 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” on page 238.

**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.

![Source code example](image)

**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.

![Source code example](image)

**Searching with Highlighting 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.

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

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.

### Using 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.
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” on page 49.

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.

   ![Highlighting the PC](image)

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

   ![Processes & Threads](image)

   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 (_run_to_) in the toolbar. The PC advances to line 136. Blue highlighting denotes a “run to” location.
3. **Out**

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

```c
30     node t *node;
31     setup (content);
32     while (node = reexpr ())) {
33         print ("%8 1d (%03lx)\n", previous, (long) previous, (long) previous
34         fflush (stdout);
35         freetree (node);
```

4. **Next**

Click **Next** ( licences) 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:

```c
30     node t *node;
31     setup (content);
32     while (node = reexpr ())) {
33         print ("%8 1d (%03lx)\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” on page 47), but you must first load the program as described in “Load the Program to Debug” on page 49.

---

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.

   ![Source code snippet]

   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.

   ![Action Points](image1)

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

   ![Action Points](image2)

   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” on page 123 for more information.

### RELATED TOPICS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action points properties</td>
<td>85</td>
</tr>
<tr>
<td>“Introducing Action Points”</td>
<td></td>
</tr>
<tr>
<td>Enabling/disabling action points</td>
<td>127</td>
</tr>
<tr>
<td>“Managing and Diving on Action Points”</td>
<td></td>
</tr>
</tbody>
</table>

### 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:

```
node_t *expression ()
{
    node_t *node = term ();
    while (nextchar == ' ' || nextchar == ':') {
        node_class_t node_class = (node_class_t) readchar ();
        node = new_node (node_class, node_class_t); readchar ();
    }
    return (node);
}
```

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

The VAR drawer displays any local variables in scope:
If you move the focus back up the call stack, the local variables in the VAR drawer 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” on page 124
Examining Data

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

For primitive, built-in types, you can quickly view data local to the selected call stack frame from within the VAR drawer. 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.

For non-simple variables such as pointers, arrays or structures, add them to the Data View where you can drill down by clicking on the arrows to open up sections of data as needed.

This section discusses viewing local and scalar variables in the VAR drawer, 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” on page 49).

Viewing Variables in the VAR Drawer

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.

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 VAR drawer**

   The VAR drawer lists local variables. Scalar values, such as `result`, are displayed directly, while compound types, such as `node`, are identified with just type information.

   ```
   VAR
   Name | Type | Value |
   ---------------------
   node | node_t* | 0x09e0100 -> (node_t) |
   result | double | 2.0737690296563e-317 <denormalized> |
   ```

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

3. **View variables in a tooltip**

   In the Source pane or the VAR drawer, hover over the variable `result` to view a tool tip that displays its value:

   ```
   VAR
   Name | Type | Value |
   ---------------------
   node | node_t* | 0x09e0100 -> (node_t) |
   result | double | 2.0737690296563e-317 <denormalized> |
   ```

**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 scalar and other variables.
• Drill down into the nested structures of compound variables.
• 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 VAR Drawer.

1. Add a Variable to the Data View
   - From the VAR drawer, just drag a variable into the Data View.
   - Alternatively, right-click and select Add to Data View.
   - Add a global variable by clicking Add New Expression in the Data View and manually entering it:
Once entered, TotalView populates its type and value:

2. **View nested structures.**

   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`.

   — 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:

   — 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:

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

- \(2 + 3\)
- \(2 \times \frac{4}{5}\)
- \(\frac{1}{2} - \frac{3}{4}\)

At this point in the program’s execution, the second expression is being read in. In the Data View, note that the left variable has been assigned a value of 4. If you drill into the right variable, 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:

4. View the output in the logger.
The output of the program goes to `stdout` when `fflush()` is called.

As each expression is evaluated and printed to `stdout`, when the `stdout` buffer is flushed, the logger window will show the result of evaluating the expression. Run the program to the end to see the completed output.

```
while (node = r->node (true)) {
  previous = evaluate (node);
  print ("%g %s (0x%x):\n", previous, (long) previous, (long) previous);
  fflush (stdout);
}
freeTree (node);
return (0);
```

RELATED TOPICS

More on the Data View  “The Data View” on page 154
More on the VAR drawer  “The Call Stack View and Local Variables” on page 143
Moving On

• For an overview on NextGen TotalView for HPC’s new interface, see “An Initial Look at the Interface” on page 4.

• For more information on ways to start and manage sessions in TotalView, see “Starting TotalView and Creating a Debugging Session” on page 17 and Chapter 2, “Creating and Managing Sessions,” on page 21.

• To use the Command Line Interface, see Chapter 8, “Using the Command Line Interface (CLI),” on page 187.

• To run your program backward, starting from the point of failure and working back in time to find the cause, see Chapter 10, “ReplayEngine,” on page 217.
PART II

Debugging Tools and Tasks

“Viewing and Organizing Processes and Threads” on page 73
How to work with the Processes & Threads view, including process, thread and group width and process and thread attributes.

“Setting and Managing Action Points (Breakpoints)” on page 85
About TotalView’s four types of action points: breakpoints, evalpoints, watchpoints, and barrier points.

“Examining and Editing Data” on page 138
Using the Call Stack view, the VAR drawer, and the Data View.

“Debugging Python” on page 176
Using TotalView to debug Python extensions.

“Using the Command Line Interface (CLI)” on page 187
Using CLI commands via the Command Line view.

“ReplayEngine” on page 217
About TotalView’s ReplayEngine which allows you to go backwards in a debugging session.

“Controlling fork, vfork, and execve Handling” on page 213
Controlling how TotalView handles system calls to execve(), fork(), vfork(), and clone().
The Processes and Threads View

The Processes and Threads view allows you to examine all processes and threads in your debugging session, and to organize them into aggregate groupings based on attributes. In Figure 35, 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.
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.

**Customizing the Display using “Group By”**

If the “Select process or thread attributes to group by” panel is not visible, click the gear icon ( ) 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; redisplay it by double-clicking on the banner again or clicking the gear icon.

**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.

**Customizing the Process and Threads View Pane**

Control which columns to display by right-clicking in the banner and selecting column names to display or hide.

View a tabular representation of process and thread state by clicking the tabular icon ( ):  

**Figure 36 – Process and Threads Tabular View**
Sort the display by clicking 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 37 – Reordering the Process and Threads View**

Here, the table has been sorted by **Function**, identified by the down arrow in the Function column header.
Customize multiple views using the three View icons ( , , ). 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 38 – Process and Threads – Viewing Different Attributes
How the Processes and Threads View Works

The Processes and 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.

Relationship of the Process and Threads View to Other 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.

Debugging Command Width

The dropdown menu on the toolbar’s left determines the scope, or width, of the debugging commands. The action that NextGen TotalView for HPC takes when you select Step or any other debugging command differs considerably depending on the width you set.
This section uses the following three related acronyms:

- **TOI**—Thread of Interest, which is the thread that has the focus in the Processes and Threads view.
- **POI**—Process of Interest, which is the process to which the TOI belongs.
- **GOI**—Group of Interest, which is all processes and threads associated with the program you are debugging, including manager threads (threads that are part of the operating environment itself and that exist to help your program get its work done).

Another important concept to understand is the *lockstep group*. A lockstep group consists of any two or more threads stopped at the same line in the program. At any given time, there may be several lockstep groups. The most important lockstep group, if it exists, is that to which the TOI belongs, because stepping commands (Step, Next, and Out) then apply to that lockstep group, not just the TOI.

In the following discussions, there is also the concept of *goal*. The goal of an executing thread is the line where it will next stop. For the Next command, it is the next line in the program. For the Run To command, it is the line currently selected in the Source view. For the Run command, it is the next breakpoint, or possibly the end of the program.

**Understanding Thread Width**

At thread width, the debugging commands apply only to the TOI, the thread that has the focus, plus any threads that belong to its lockstep group, if one exists. NextGen TotalView for HPC does not allow other program threads to run, but manager threads run freely.

Debugging commands selected from the Thread menu also run at thread width.

Operating on a thread isn't the same as operating on a thread's process, because a process can have more than one thread.

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.
Understanding Process Width

At process width, the GOI consists of the process to which the TOI belongs, and all threads associated with that process, including manager threads. If the TOI happens to belong to a lockstep group that includes threads from other processes, those threads are included in the GOI as well. Execution continues until the TOI arrives at its goal, which can be the next statement, the next breakpoint, and so on. When the TOI reaches the goal, NextGen TotalView for HPC stops the other threads in the GOI.

Debugging commands selected from the Process menu also run at process width.

Understanding Group Width

At group width, the GOI is the control group, which is all processes associated with the program you are debugging. NextGen TotalView for HPC examines all of the processes and looks for processes that have a thread stopped at the same location as the TOI. These are deemed to be matching processes. TotalView runs these matching processes until one of their threads arrives at the goal. When this happens, TotalView stops that thread’s process. The command finishes when all matching processes have been stopped.

Debugging commands selected from the Group menu also run at group width.

Other Considerations When Stepping Through Code

Share Groups

A share group is all processes associated with a given executable. In most cases, your program will be defined by a single executable, in which case the control group, which is the group used at group width, consists of just one share group. If, however, your program uses one or more other executables, the control group will consist of two or more share groups. In this case, operations like stopping at a breakpoint do not apply to all processes and threads in the control group, but rather to all processes and threads in the share group to which the TOI belongs. This is because the processes and threads in the other share groups cannot stop at the breakpoint since it is not in the code they are executing.

Using Run To to Synchronize Processes and Threads

The Run To command, which runs to a selected line in the Source view code, is helpful in synchronizing processes and threads so you can step them together. The behavior differs depending on whether you are using this command at process width or group width.

At process width:

- If the Thread of Interest (TOI) is already at the goal location, NextGen TotalView for HPC 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, NextGen TotalView for HPC 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 POI at a source line.

At group width:

• NextGen TotalView for HPC 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.

**Remembering Breakpoint Width**

The width discussions here pertain to the menu on the left of the toolbar. If you are setting breakpoints and using the Go button to run to them, the width you set on the breakpoints themselves affects what NextGen TotalView for HPC does. Breakpoints can be set to, when hit, stop just the one thread, all threads in the process to which the thread belongs, or all threads. If, for example, you have selected Thread width from the toolbar menu, but the breakpoint you are running to is set to stop all threads in the process (the default), then all threads in the process will be stopped instead of just the one thread as you might otherwise expect.

For a complete discussion of breakpoint widths, see “Controlling an Action Point’s Width” on page 123.

**Setting the Program Counter**

To resume execution at a different statement than at stopped execution, reset the value of the program counter (PC). This is useful in the event that you want to skip over some code, execute some code again after changing certain variables, or restart a thread that is in an error state.

Setting the PC can be critical when restarting a thread that is in an error state. Although the PC symbol in the line number area points to the source statement that caused the error, the PC actually points to the failed machine instruction in the source statement. You need to explicitly reset the PC to the correct instruction. (You can verify the actual location of the PC before and after resetting it by displaying it in the Call Stack, or displaying the source code in the Source pane.)

Set the PC of a stopped thread to either a selected source line or a selected instruction. When you set the PC to a selected line, the PC points to the memory location where the statement begins. For most situations, setting the PC to a selected line of source code is the best choice.

To set the PC to a selected line:

1. Select the source line or instruction in the Source pane to highlight the line.
2. Select the **Thread > Set PC** command.

The PC moves to the selected line.

When resetting the PC to a selected line, TotalView attempts to force the thread to continue execution at that statement in the currently selected stack frame. If the currently selected frame is not the top stack frame, TotalView prompts to unwind the stack:

**This frame is buried. Should we attempt to unwind the stack?**

- For **Yes**, TotalView discards deeper stack frames (that is, all stack frames that are more deeply nested than the selected stack frame) and resets the machine registers to their values for the selected frame.

- For **No**, TotalView sets the PC to the selected line, but leaves the stack and registers in their current state. Since you can't assume that the stack and registers have the right values, selecting No is not recommended.
### Summary of Process and Thread Attributes

Table 2 describes the attributes you can select and what each one means.

**Table 2: Processes and Threads Attribute Descriptions**

<table>
<thead>
<tr>
<th>Property</th>
<th>Applies to</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Group</td>
<td>Processes</td>
<td>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.</td>
</tr>
<tr>
<td>Share Group</td>
<td>Processes</td>
<td>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.</td>
</tr>
<tr>
<td>Hostname</td>
<td>Processes</td>
<td>The hostname or IP address of where the process is running.</td>
</tr>
<tr>
<td>Process State</td>
<td>Processes</td>
<td>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.</td>
</tr>
<tr>
<td>Thread State</td>
<td>Threads</td>
<td>The thread execution state, e.g., Running, Stopped, Breakpoint, Watchpoint, etc.</td>
</tr>
<tr>
<td>Function</td>
<td>Threads</td>
<td>The function name of the location of the stopped thread. Displays the function name or “&lt;unknown address&gt;” if the thread is running or the function name is not known.</td>
</tr>
<tr>
<td>Source Line</td>
<td>Threads</td>
<td>The source code line of the location of the stopped thread. Displays the source file name and line number, or “&lt;unknown line&gt;” if the thread is running or the source line is not known.</td>
</tr>
<tr>
<td>PC</td>
<td>Threads</td>
<td>The program counter of the location of the stopped thread. Displays the program counter value, or “&lt;unknown address&gt;” if the thread is running.</td>
</tr>
<tr>
<td>Action Point ID</td>
<td>Threads</td>
<td>The action point ID (breakpoint or watchpoint) of the location of the stopped thread. Displays “ap(id)”, where id is the action point ID, or “none” if the thread is not stopped at an action point.</td>
</tr>
<tr>
<td>Stop Reason</td>
<td>Threads</td>
<td>More detailed information on why the process/thread has stopped.</td>
</tr>
<tr>
<td>Process ID</td>
<td>Processes</td>
<td>The debugger process ID (dpid) of the process. Displays dpid.</td>
</tr>
<tr>
<td>Thread ID</td>
<td>Threads</td>
<td>The dpid and debugger thread ID (dtid) of the thread. Displays dpid.dtid.</td>
</tr>
<tr>
<td>Thread Index</td>
<td>Threads</td>
<td>The thread index part of the Thread ID. For example, if the Thread ID is 3.2, the Thread Index is 0.2.</td>
</tr>
<tr>
<td>Process Held</td>
<td>Processes</td>
<td>The ‘hold’ state of the process. This may be because of an explicit hold request, or because the process is waiting at a barrierpoint.</td>
</tr>
<tr>
<td>Thread Held</td>
<td>Threads</td>
<td>The ‘hold’ state of the thread. This may be because of an explicit hold request, or because the thread is waiting at a barrierpoint.</td>
</tr>
<tr>
<td>Replay Mode</td>
<td>Processes</td>
<td>The Replay state (Replay or Record) of a process that has Replay enabled.</td>
</tr>
</tbody>
</table>
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 and Threads view.

<table>
<thead>
<tr>
<th>Button</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td>Moves the selected attribute up the list. If it passes another selected attribute, this changes the display in the Processes and Threads view.</td>
</tr>
<tr>
<td>↓</td>
<td>Moves the selected attribute down the list. If it passes another selected attribute, this changes the display in the Processes and Threads view.</td>
</tr>
<tr>
<td>⚙</td>
<td>Resets the initial order and the initial selection of attributes.</td>
</tr>
</tbody>
</table>
Chapter 5

Setting and Managing Action Points (Breakpoints)

Introducing Action Points

NextGen TotalView for HPC 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 41 – Action Points View

Each type of action point is identified with a distinctive icon, as displayed in Table 3.

Table 3: Action Point Types and Identifying Icons

<table>
<thead>
<tr>
<th>Action Points Icons</th>
<th>Type of Action Point</th>
<th>How to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Break</strong></td>
<td>Breakpoint</td>
<td>See “Breakpoints” on page 87</td>
</tr>
<tr>
<td><strong>Eval</strong></td>
<td>Evalpoint</td>
<td>See “Evalpoints” on page 98</td>
</tr>
<tr>
<td><strong>Watch</strong></td>
<td>Watchpoint</td>
<td>See “Watchpoints” on page 107</td>
</tr>
<tr>
<td><strong>Barr</strong></td>
<td>Barrier point</td>
<td>See “Barrier Points” on page 136</td>
</tr>
</tbody>
</table>

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 42 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” on page 91 and “Sliding Breakpoints” on page 88).

Figure 42 – Possible breakpoint locations in the Source view

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

```c
15 structData mystructArray[20];
16 int myArray[10];
17 structData mystruct;
18 mystruct.x = 10;
19 mystruct.y = 20;
```
• 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 43 – 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 44**, 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.
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” on page 91 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 45* illustrates a breakpoint set at line 91 that has slid to line 134:
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/codedynamics/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**” on page 91.

### 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:

(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” on page 132.

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.
• 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 multi-threaded 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.

### 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. (See “Share Groups” on page 80.) 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`

### Debugging Processes that Call the `execve()` Function

**NOTE >>** You can control how TotalView handles system calls to `execve()`. See “Exec Handling” on page 214.

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:
   Process name has exec’d name. Do you want to stop it now?

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:

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stops the parent process before it forks.</td>
</tr>
<tr>
<td>2</td>
<td>Stops both the parent and child processes.</td>
</tr>
<tr>
<td>3</td>
<td>Stops the parent process if the <code>fork()</code> function failed.</td>
</tr>
<tr>
<td>5</td>
<td>Stops the child process.</td>
</tr>
<tr>
<td>7</td>
<td>Stops the parent process.</td>
</tr>
</tbody>
</table>

**RELATED TOPICS**

- Linking with the dbfork library
- "Linking with the dbfork Library" on page 298
- Controlling system calls to `execve()`.
- "Exec Handling" on page 214
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” on page 102.
- Test potential fixes or patches for your program; see “Patching Programs” on page 103.
- 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” on page 283.
- 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“ on page 102 |
| Saving Action Points | “Saving Action Points to a File Using the CLI“ on page 136 |
| Using built-in TotalView statements to control execution | “Using Built-In Statements“ on page 284 |
| Writing code for an expression | “Using Programming Language Elements“ on page 286 |
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” on page 133).
- **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” on page 91.
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 \texttt{my_var} variable equals 4, the process stops the eighth time it executes the \texttt{$count} function. After the process stops, TotalView reevaluates the expression. If \texttt{my_var} equals 5, the process stops again after the process executes the \texttt{$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 \texttt{$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 \texttt{$count} is most often used within loops, you can use it outside of them as well.

For descriptions of the \texttt{$stop}, \texttt{$count}, and variations on \texttt{$count}, see “Using Built-in Variables and Statements” on page 283.

**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 \texttt{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 \texttt{goto} (C) or \texttt{GOTO} (Fortran) statement that jumps over the code you no longer want executed. For example, the evalpoint in Figure 46 executes three statements and then skips to line 656.
Branching Around Code

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

```c
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:

```c
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:

```c
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 47. The results are printed in the shell in which TotalView is running, displayed on the right.

**Figure 47 – 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” on page 108
- “Modifying Watchpoints” on page 110
- “Watching Memory” on page 111
- “Triggering Watchpoints” on page 112
- “Using Watchpoint Expressions” on page 113
- “Using Watchpoints on Different Architectures” on page 114

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**” on page 112 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**” on page 134.
Creating Watchpoints

Create a watchpoint in these ways:

- From the VAR panel 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” on page 110 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” on page 111 for more information.

**Displaying, Deleting, or Disabling Watchpoints**

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

![Watchpoint entry](image)

NOTE >> A watchpoint's width is set to Group by default. See “About an Action Point's Width: Group, Process or Thread” on page 123. See “Modifying Watchpoints” on page 110 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.

![Disabled watchpoint](image)

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

![Watchpoint in Process & Threads](image)
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.

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” on page 114.

- **When value changes**: The three width selections control how a breakpoint behaves in a multi-threaded 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” on page 113.

— **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:

![Image of a watchpoint triggering on `arg_count`]

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:

```java
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:

```java
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:

```java
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” on page 286.
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:

```c
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” on page 283.

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

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:

<table>
<thead>
<tr>
<th>Computer</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux x86-64</td>
<td>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.</td>
</tr>
<tr>
<td>Linux-PowerLE</td>
<td>On Linux-PowerLE platforms (but not Linux-Power big-endian 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, -(address % length) == 0.</td>
</tr>
<tr>
<td>Linux ARM64</td>
<td>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.</td>
</tr>
<tr>
<td>Mac OSX</td>
<td>Watchpoints are not supported.</td>
</tr>
</tbody>
</table>

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.

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.

![Figure 48 – Modify Barrier Point Property dialog](image)

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:
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:

<table>
<thead>
<tr>
<th>Scope</th>
<th>Stopped Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>Stops all threads in the current thread's control group.</td>
</tr>
<tr>
<td>Process</td>
<td>Stops all threads in the current thread's process.</td>
</tr>
<tr>
<td>Thread</td>
<td>Stops only this thread.</td>
</tr>
</tbody>
</table>

*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:

<table>
<thead>
<tr>
<th>Scope</th>
<th>Stopped Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>Stops all threads in the current thread's control group.</td>
</tr>
<tr>
<td>Process</td>
<td>Stops all threads in the current thread's process.</td>
</tr>
<tr>
<td>Thread</td>
<td>Stops only this thread.</td>
</tr>
</tbody>
</table>
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 49 – Satisfaction Set properties**

<table>
<thead>
<tr>
<th>Scope</th>
<th>Held Processes or Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control</strong></td>
<td>The default. Holds all processes.</td>
</tr>
<tr>
<td><strong>Share</strong></td>
<td>Holds all the processes that share the same image as the current executable where the barrier point is set.</td>
</tr>
<tr>
<td><strong>Workers</strong></td>
<td>Holds only this thread.</td>
</tr>
</tbody>
</table>

**Control** and **Share** settings hold at the process level. For multi-threaded 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 TotalView for HPC 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:

![Processes & Threads view](image)

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 multi-threaded 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 50 – Running to Barriers**

For more information on barriers, see "Barrier Points" on page 116.

**RELATED TOPICS**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dhold</code></td>
<td><em>CLI Commands</em> in the <em>NextGen TotalView for HPC Reference Guide</em></td>
</tr>
<tr>
<td><code>dunhold</code></td>
<td><em>CLI Commands</em> in the <em>NextGen TotalView for HPC Reference Guide</em></td>
</tr>
</tbody>
</table>
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 multi-threaded 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, multi-threaded 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 51 – Action Points width context menu
The Stop column displays the selected width:

![Stop column](image)

**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.

![Processes & Threads][1]

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.

<table>
<thead>
<tr>
<th>Thread State</th>
<th>Function</th>
<th>TID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint</td>
<td>snore_or_leave</td>
<td>4.3</td>
</tr>
<tr>
<td>Stopped</td>
<td>_select</td>
<td>1.1</td>
</tr>
<tr>
<td>Stopped</td>
<td>_select</td>
<td>1.2</td>
</tr>
<tr>
<td>Stopped</td>
<td>_select</td>
<td>2.1</td>
</tr>
<tr>
<td>Stopped</td>
<td>_select</td>
<td>2.2</td>
</tr>
<tr>
<td>Stopped</td>
<td>_select</td>
<td>3.1</td>
</tr>
<tr>
<td>Stopped</td>
<td>_select</td>
<td>3.2</td>
</tr>
<tr>
<td>Stopped</td>
<td>_select</td>
<td>4.1</td>
</tr>
<tr>
<td>Stopped</td>
<td>_select</td>
<td>4.2</td>
</tr>
<tr>
<td>Stopped</td>
<td>snore_or_leave</td>
<td>1.3</td>
</tr>
<tr>
<td>Stopped</td>
<td>snore_or_leave</td>
<td>2.3</td>
</tr>
<tr>
<td>Stopped</td>
<td>snore_or_leave</td>
<td>3.3</td>
</tr>
</tbody>
</table>

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:

![Action Points view example](image)
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:

![Action Points Table]

**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:

![Source View and Action Points View]

- **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:

![Save Breakpoints Dialog](image)

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

While this version of NextGen TotalView for HPC does not yet support all action point functionality in the UI, you have access to all features using the CLI through the command line view.

Creating Barrier Points, Evalpoints, and Watchpoints

The CLI supports multiple types of action points, some of which are not currently supported in the UI:

- A **breakpoint** stops execution of processes and threads that reach it. See “Breakpoints” on page 87 for UI functionality.
- An **evalpoint** executes a code fragment when it is reached. See “Evalpoints” on page 98 for UI functionality.
- An **watchpoint** monitors a location in memory and stops execution when a provided condition is met. See “Watchpoints” on page 107 for UI functionality.
- A **barrier point** synchronizes a set of threads or processes at a location. Not yet supported in the UI.

**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, e.g., and . 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:
**Breakpoints**

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

- **Set a breakpoint directly on a function:**
  
  
  
  \[
  \text{dbreak my\_function}
  \]

  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:**
  
  
  
  \[
  \text{dbreak -address 0x2b0b7aad1470}
  \]

  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, NextGen TotalView for HPC 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.

- **Print the value of result:**
  
  
  
  \[
  \text{dbreak 597 -e \{ printf("The value of result is \%d\n", result) \};}
  \]

  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:**
  
  
  
  \[
  \text{dbreak 50 -e \{goto 63\};}
  \]

  Any thread that reaches this breakpoint transfers to line 63.

- **Stop a loop after a certain number of iterations:**
  
  
  
  \[
  \text{dbreak 597 -e \{ if ( (i \% 100) == 0) \}
  \]

  \[
  \text{\{ printf("The value of i is \%d\n", i);}
  \]

  \[
  \text{
  \$stop;}
  \]
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 VAR panel that `arg_count` equals 1:

![VAR panel showing `arg_count` equals 1]
The watchpoint is now reflected in the Action Points view with the watchpoint icon:

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 **Processes & Threads view** reports the Process State stopped at this watchpoint.
- The **Source view** identifies the location of the PC.
- The **VAR panel** 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](#) in the NextGen TotalView for HPC Reference Guide
- **Evalpoints** [dbreak](#) in the NextGen TotalView for HPC Reference Guide
- **Barrier points** [dbarrier](#) in the NextGen TotalView for HPC Reference Guide
- **Watchpoints** [dwatch](#) in the NextGen TotalView for HPC Reference Guide
Barrier Points

In a multi-threaded, 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. NextGen TotalView for HPC 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 “NextGen TotalView for HPC Command Syntax” in the NextGen TotalView for HPC Reference Guide) to automatically save your breakpoints.

CLI: `dsetTV::auto_save_breakpoints`

At any time, you can restore saved action points.

CLI: `dactions -load filename`
Suppressing and Unsuppressing Action Points

Action points can be suppressed and unsuppressed as described in “Deleting, Disabling, and Suppressing” on page 128. The CLI commands for this are:

CLI: dactions --supress
CLI: dactions --unsuppress
Overview

NextGen TotalView for HPC is rich with features to analyze your program’s data.

The Call Stack, Local Variable (VAR drawer), and Data View

The Call Stack, the Local Variable (VAR) drawer, and the Data View all work together to provide views of your data at different points of your running program.

The VAR panel and the Data View work in concert to display your program’s data in detail.

The VAR panel 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 VAR panel update.

The VAR panel displays data only for scalar data. To see non-simple variables such as pointers, arrays or structures, add them to the Data View. 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 from the context menu, or by simply dragging the variable name from the VAR panel 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 TotalView for HPC.

Diving

Diving is integral to the NextGen TotalView for HPC 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. 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.

**Editing Data**

Edit a wide range of data while debugging your programs, such as variable type and value.

**Included in this chapter:**

“About Expressions” on page 140

“The Call Stack View and Local Variables” on page 143

“The Data View” on page 154

“Searching for Program Elements” on page 172

“Using the CLI to Examine Data” on page 173
About Expressions

Either directly or indirectly, accessing and manipulating data requires an evaluation system. When your program (and NextGen TotalView for HPC, 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[cntr]` or `my_var[cntr+3]`. In the latter case, TotalView must determine the value of `cntr+3` before it can access an array element.

Here is an illustration showing NextGen TotalView for HPC accessing the `my_var array` in the three ways discussed in this section:
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:

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

Figure 53 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 53 – Class Casting

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cylinder</td>
<td>struct Cylinder</td>
<td>( struct Cylinder)</td>
</tr>
<tr>
<td>Circle</td>
<td>struct Circle</td>
<td>( struct Circle)</td>
</tr>
<tr>
<td>m_height</td>
<td>double</td>
<td>50.5</td>
</tr>
<tr>
<td>cylinder</td>
<td>struct Shape</td>
<td>( struct Shape)</td>
</tr>
<tr>
<td>$table</td>
<td>int(...)**</td>
<td>0x00401730 -&gt; 0x00401464 : Shape::area(void) const</td>
</tr>
<tr>
<td>m_area</td>
<td>double</td>
<td>32046.64552</td>
</tr>
<tr>
<td>m_volume</td>
<td>double</td>
<td>809177.79938</td>
</tr>
</tbody>
</table>

[Add New Expression]
The Call Stack View and Local Variables

The **Call Stack View** consists of two panels, the **Call Stack** panel and the **Local Variables (VAR) Panel** (identified as “VAR” on the interface):

**Figure 54 – Call Stack and VAR Panels**

- **Call Stack panel** 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.

- **VAR panel**, or **drawer**, below the Call Stack panel displays all the arguments and local variables associated with the selected frame in the Call Stack view.
NOTE >> Global variables are not displayed in the VAR panel. To view them, add them to the Data View. See “Entering a New Expression into the Data View” on page 160.

The VAR drawer's three columns display each variable's Name, Type and Value at the time that the thread stopped. For example, in Figure 54, for selected function snore, the local variables display under the associated scope or program block.

NOTE >> The VAR drawer displays only scalar data; to drill down and view complex data, dive on the variable to add it to the Data View. See “The Data View” on page 154.
• The **Information tab** on the VAR panel displays additional detail about the location of the stopped thread and the selected frame in the stack trace.

   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.

• **Copy a variable definition** from the NextGen TotalView for HPC 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.

---

**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 VAR panel.

*Figure 55 – The VAR drawer displays local variables for the TOI*
The Call Stack View identifies `wait_a_while`, `snore`, and `snore_or_leave` as a C++ function. Other possible icons include Fortran or C. If a language is displayed, then there is debug information for that frame, so, for instance, `start_thread_select_nocancel`, and `_clone` have no debug information, and the source is not available. If selected, “Source not available” displays.

**Figure 56 – Source not available**

Refocusing the Source View updates the VAR panel

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 57** illustrates that moving down the stack to `snore_or_leave` updates the VAR panel with new local values and arguments.

**Figure 57 – 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 view and its VAR drawer 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 NextGen TotalView for HPC 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 NextGen TotalView for HPC 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, NextGen TotalView for HPC 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 58 illustrates some of the points made above.
The qualified subroutine name appears in the Call Stack view, and the qualified variable names appear in the VAR drawer. Note also that the \texttt{init()} routine can be called from \texttt{main} without qualification because of the \texttt{USE} statement.

**Common Blocks**

For each common block defined in the scope of a subroutine or function, NextGen TotalView for HPC 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 view, the common blocks and their variables appear in the VAR drawer. From there, of course, you can move those variables to the Data view to obtain more information.
Figure 59 illustrates the handling of common blocks.

**Fortran 90 User-Defined Types**

A Fortran 90 user-defined type is similar to a C structure. NextGen TotalView for HPC 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`:

```fortran
module common
  private
  real, allocatable :: ja(20)
end module common

module foobar
  private
  type foo
    integer ifoo
  end type foo
  type bar
    integer mdarray(2,3,4,5)
  end type bar
end module foobar
```

And this code creates variables of these types, one a simple type, one an 20-dimensional array, and one a pointer:

```fortran
use foobar

type (bar), target :: just_a_bar
type (foo), dimension(20), target :: foo_array
type (foo), pointer, dimension(:) :: foo_p
```
NextGen TotalView for HPC displays these in the VAR drawer, which you can then add to the Data View for more detail:

**Figure 60 – Fortran User-Defined Types**

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:

```fortran
real, allocatable, dimension(:) :: aal
```

Here is the unallocated array displayed in the Data View:

**Figure 61 – 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 62 – Fortran Deferred Shape Array**

---

**Fortran 90 Pointer Types**

A Fortran 90 pointer type points to scalar or array types.

NextGen TotalView for HPC implicitly handles slicing operations that set up a pointer or assumed shape subroutine argument so that the indices and values it displays in a VAR drawer 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:

```fortran
integer, target, dimension(5,2:10) :: ia21,ia22
integer, pointer, dimension(:,:) :: ip21, ip22

ip21 => ia21(:,2)
```

---
Figure 63 displays the original array ia21 and its pointer ip21 in the Data View.

**Figure 63 – Original Fortran Array**

![Data View](image)

Figure 64 illustrates the pointer ip21 representing a slice of the ia21 array after the assignment of the pointer.

**Figure 64 – Fortran Pointer Representing an Array Slice**

![Data View](image)

**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 NextGen TotalView for HPC 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:

```fortran
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 65 assigns a parameter to a local variable for display in the Data View:

**Figure 65 – 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 66 – Fortran Parameters, added to Data View**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>esc</td>
<td>character(len=1)</td>
<td>&quot;033&quot;</td>
</tr>
</tbody>
</table>
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” on page 154
- “Creating a New Expression by Diving on a Variable in the Data View” on page 157
- “Dereferencing a Pointer” on page 162
- “Entering a New Expression into the Data View” on page 160
- “Changing the Value of Data” on page 162
- “Casting to Another Type” on page 163
- “Displaying Arrays” on page 166
- “Customizing the Data View” on page 169, including Duplicating the Data View and The Data View Drawer
- “Controlling STL Data Transformation” on page 167
- “Using the CLI to Examine Data” on page 173

Adding Variables to the Data View

Once you have started your program and then it has stopped at a breakpoint or by using a stepping command, the VAR panel populates with local data from whatever stack frame is selected in the Call Stack view. The Source view is also refocused on the source file associated with the selected frame.

To add expressions, i.e. variables, to the Data View, use one of these methods:

- Add to the Data View from the VAR Panel using either the context menu or by dragging and dropping.
- Creating a New Expression by Diving on a Variable in the Data View.
- Entering a New Expression into the Data View.
- Move an Expression or Variable from the Source View to the Data View using the context menu.

NOTE >> Because global variables are not displayed in the VAR panel, add them to the Data View by typing them in directly. See “Entering a New Expression into the Data View” on page 160.
Add to the Data View from the VAR Panel

Use the VAR panel to add variables to the Data View, either by using the context menu or by dragging and dropping.

Figure 67 – Add to the Data View using the context menu

Figure 68 – Add to the Data View by dragging from the VAR panel

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.
Examining and Editing Data

The Data View

Figure 69 – Evaluated expressions use the original scope

This allows you to uniquely view the values of different variables with the same name in different scopes within your program.

If they go out of scope, an error is displayed:

Move an Expression or 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:
Creating a New Expression by Diving on a Variable in the Data View

The key to viewing your data is to dive on it, which creates a new expression in the Data View. 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.

Right-click on an expression in the Data view and select Dive.
The new expression is appended to the end of any existing expressions in the Data View, and the view automatically scrolls to this new position.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>twoDArray</td>
<td>int[20][20]</td>
<td>(int[20][20])</td>
</tr>
<tr>
<td>stringArray</td>
<td>sstring *[20]</td>
<td>(sstring *[20])</td>
</tr>
</tbody>
</table>

**Diving on a complex variable**

Select the right arrow to display the substructures in a complex variable.

If your complex variable has nested structures, these display in the Data View:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>myStruct</td>
<td>struct struct...</td>
<td>(struct structData)</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>0x00000000a (10)</td>
</tr>
<tr>
<td>y</td>
<td>int</td>
<td>0x000000014 (20)</td>
</tr>
<tr>
<td>b</td>
<td>float</td>
<td>777.2</td>
</tr>
<tr>
<td>nsa</td>
<td>struct nested...</td>
<td>(struct nestedStruct)</td>
</tr>
<tr>
<td>nsa</td>
<td>int[3]</td>
<td>(int[3])</td>
</tr>
</tbody>
</table>
View the nested structures by selecting the down arrow:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>myStruct</td>
<td>struct struct...</td>
<td>(struct strutData)</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>0x0000000a (10)</td>
</tr>
<tr>
<td>y</td>
<td>int</td>
<td>0x00000014 (20)</td>
</tr>
<tr>
<td>b</td>
<td>float</td>
<td>777.2</td>
</tr>
</tbody>
</table>

A new expression for this single element is added to the Data View:

NOTE >> You can also “dive” on a single element by adding a new expression. See “Entering a New Expression into the Data View” on page 160.
**Diving on multiple expressions**

You can select one or multiple expressions to dive on. Hold down the **Ctrl** key and select the expressions you want:

![Image of a data view with selected variables and sub-elements.]

Here we are selected a top-level complex variable as well as a sub-element of another structure.

**Deleting an expression**

Delete an expression using the context menu and selecting **Delete** or by clicking **Delete** on your keyboard.

![Image of a data view with delete options highlighted.]

**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.

**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:

  ![Data View](image)

  A new expression is added. This is the same as diving on the sub-element.

- **Increment a variable**:

  ![Data View](image)

- **Add or subtract variables**, assuming some relationship:

  ![Data View](image)

- **Add a global variable**. Type the variable name into the Name column and TotalView automatically enters the type and value.

  ![Code and Data View](image)

  The value updates in the Data View as you run through your program.
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:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>localString</td>
<td>string *</td>
<td>0x004009c8 -&gt; &quot;HelloHelloHelloHelloHelloHello...&quot;</td>
</tr>
</tbody>
</table>
```

Double-click in the Name column to make the text editable, and then dereference the pointer:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>localString</td>
<td>string</td>
</tr>
</tbody>
</table>
```

The Data View displays the variable’s value:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>localString</td>
<td>string</td>
<td>&quot;HelloHelloHelloHelloHelloHelloHello...&quot;</td>
</tr>
</tbody>
</table>
```

For `argv`, i.e. a pointer to a pointer, dereference it twice.

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>localString</td>
<td>string</td>
<td>&quot;HelloHelloHelloHelloHelloHelloHelloHello...&quot;</td>
</tr>
<tr>
<td>argv</td>
<td>string</td>
<td>*/home/bburns/sandbox/build/linux-x86...</td>
</tr>
</tbody>
</table>
```

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>myStruct</td>
<td>struct struct... (struct structData)</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>0x00000000 (100)</td>
</tr>
<tr>
<td>y</td>
<td>int</td>
<td>0x00000014 (20)</td>
</tr>
</tbody>
</table>

**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**

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:

Depending on the array declaration, NextGen TotalView for HPC displays arrays differently. See “Displaying Arrays” on page 166.

**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]`. 

| dynStrings | $string *[10] | 0x01884040 > 0x00000000 |
Then click the down arrow to display your array of 10 strings.

**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>
<thead>
<tr>
<th>Type String</th>
<th>Language</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$address</td>
<td>C</td>
<td>void*</td>
<td>Void pointer (address).</td>
</tr>
<tr>
<td>$char</td>
<td>C</td>
<td>char</td>
<td>Character.</td>
</tr>
<tr>
<td>$character</td>
<td>Fortran</td>
<td>character</td>
<td>Character.</td>
</tr>
<tr>
<td>$code</td>
<td>C</td>
<td>architecture-dependent</td>
<td>Machine instructions. The size used is the number of bytes required to hold the shortest instruction for your computer.</td>
</tr>
<tr>
<td>$complex</td>
<td>Fortran</td>
<td>complex</td>
<td>Single-precision floating-point complex number. The complex types contain a real part and an imaginary part, which are both of type real.</td>
</tr>
<tr>
<td>$complex_8</td>
<td>Fortran</td>
<td>complex*8</td>
<td>Areal<em>4-precision floating-point complex number. The complex</em>8 types contain a real part and an imaginary part, which are both of type real*4.</td>
</tr>
<tr>
<td>$complex_16</td>
<td>Fortran</td>
<td>complex*16</td>
<td>Areal<em>8-precision floating-point complex number. The complex</em>16 types contain a real part and an imaginary part, which are both of type real*8.</td>
</tr>
<tr>
<td>$double</td>
<td>C</td>
<td>double</td>
<td>Double-precision floating-point number.</td>
</tr>
</tbody>
</table>
### Table 4: TotalView Built-in Types

<table>
<thead>
<tr>
<th>Type String</th>
<th>Language</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$double_precision</td>
<td>Fortran</td>
<td>double</td>
<td>Double-precision floating-point number.</td>
</tr>
<tr>
<td>$extended</td>
<td>C</td>
<td>architecture-dependent; often long double</td>
<td>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.</td>
</tr>
<tr>
<td>$float</td>
<td>C</td>
<td>float</td>
<td>Single-precision floating-point number.</td>
</tr>
<tr>
<td>$int</td>
<td>C</td>
<td>int</td>
<td>Integer.</td>
</tr>
<tr>
<td>$integer</td>
<td>Fortran</td>
<td>integer</td>
<td>Integer.</td>
</tr>
<tr>
<td>$integer_1</td>
<td>Fortran</td>
<td>integer*1</td>
<td>One-byte integer.</td>
</tr>
<tr>
<td>$integer_2</td>
<td>Fortran</td>
<td>integer*2</td>
<td>Two-byte integer.</td>
</tr>
<tr>
<td>$integer_4</td>
<td>Fortran</td>
<td>integer*4</td>
<td>Four-byte integer.</td>
</tr>
<tr>
<td>$integer_8</td>
<td>Fortran</td>
<td>integer*8</td>
<td>Eight-byte integer.</td>
</tr>
<tr>
<td>$logical</td>
<td>Fortran</td>
<td>logical</td>
<td>Logical.</td>
</tr>
<tr>
<td>$logical_1</td>
<td>Fortran</td>
<td>logical*1</td>
<td>One-byte logical.</td>
</tr>
<tr>
<td>$logical_2</td>
<td>Fortran</td>
<td>logical*2</td>
<td>Two-byte logical.</td>
</tr>
<tr>
<td>$logical_4</td>
<td>Fortran</td>
<td>logical*4</td>
<td>Four-byte logical.</td>
</tr>
<tr>
<td>$logical_8</td>
<td>Fortran</td>
<td>logical*8</td>
<td>Eight-byte logical.</td>
</tr>
<tr>
<td>$long</td>
<td>C</td>
<td>long</td>
<td>Long integer.</td>
</tr>
<tr>
<td>$long_long</td>
<td>C</td>
<td>long long</td>
<td>Long long integer.</td>
</tr>
<tr>
<td>$real</td>
<td>Fortran</td>
<td>real</td>
<td>Single-precision floating-point number. When using a value such as real, be careful that the actual data type used by your computer is not real<em>4 or real</em>8, since different results can occur.</td>
</tr>
<tr>
<td>$real_4</td>
<td>Fortran</td>
<td>real*4</td>
<td>Four-byte floating-point number.</td>
</tr>
<tr>
<td>$real_8</td>
<td>Fortran</td>
<td>real*8</td>
<td>Eight-byte floating-point number.</td>
</tr>
<tr>
<td>$real_16</td>
<td>Fortran</td>
<td>real*16</td>
<td>Sixteen-byte floating-point number.</td>
</tr>
<tr>
<td>$short</td>
<td>C</td>
<td>short</td>
<td>Short integer.</td>
</tr>
<tr>
<td>$string</td>
<td>C</td>
<td>char</td>
<td>Array of characters.</td>
</tr>
<tr>
<td>$void</td>
<td>C</td>
<td>long</td>
<td>Area of memory.</td>
</tr>
</tbody>
</table>
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:

```c
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:

```fortran
integer a(1:7,1:8)
```

Since both dimensions of this array use Fortran’s default 1 lower bound, NextGen TotalView for HPC displays the data type using only the extent of each dimension, as follows:

```fortran
integer(7,8)
```

If an array declaration doesn’t use the default lower bound, NextGen TotalView for HPC displays both the lower and upper bound for each dimension. For example, this Fortran array is displayed in NextGen TotalView for HPC the same way it is declared:

```fortran
integer a(1:7,1:8)
```
integer a(-1:5,2:10)

Controlling STL Data Transformation

When a debugger displays a variable, it usually relies on the definitions of the data used by your compiler. Next-Gen TotalView for HPC's Data View, however, automatically transforms your data in an aggregated list or array view that makes it easier to examine.

This is particularly important for C++ STL (Standard Template Library) types that use abstractions such as structures, classes, and data types, including lists, maps, and vectors.

By default, NextGen TotalView for HPC transforms STL types, including strings, vectors, lists, maps, multimaps, sets, and multisets. This behavior is part of the NextGen TotalView for HPC Type Transformation Facility (TTF) that provides tools for customizing how you view data.

Viewing untransformed data

If you do need to look at the untransformed data structures, use the CLI's `dset` command to set the `TV::ttf` variable to `false`:

```
CLI: dset TV::ttf { true | false }
```

For example, here is how your compiler sees a vector compiled using the GNU C++ compiler (g++):
Most of the information is generated by the STL template and, in most cases, provides little value for analysis. In addition, the STL does not aggregate the information in a useful way.

NextGen TotalView for HPC solves this problem by transforming the data so that you can easily examine it. For example, here is the transformed vector, using TotalView’ default TTF settings:

### Figure 71 – A Transformed Vector

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>struct std::vector&lt;double&gt;</td>
<td>7.9</td>
</tr>
<tr>
<td>_M_start</td>
<td>double</td>
<td>0x00bab100 -&gt; 7.9</td>
</tr>
<tr>
<td>_M_finish</td>
<td>double</td>
<td>0x00bab118 -&gt; 0</td>
</tr>
<tr>
<td>_M_end_of_stor</td>
<td>double</td>
<td>0x00bab120 -&gt; 0</td>
</tr>
</tbody>
</table>
You can also create transformations for other STL containers.

<table>
<thead>
<tr>
<th>RELATED TOPICS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>General information on creating custom type</td>
<td>&quot;Creating Type Transformations&quot; in the NextGen TotalView for HPC Reference Guide</td>
</tr>
<tr>
<td>transformations</td>
<td></td>
</tr>
</tbody>
</table>

**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 duplicate the Data View for more options in viewing data. See “Duplicating the Data View” on page 169.

**Duplicating the Data View**

To examine variables in a dedicated window, duplicate the Data View using the view's "+" toolbar button.

There are multiple ways to create duplicated views.

- Create an empty copy of the Data View and then manually add variables to analyze in their own view.
• Create a copy of the Data View already populated with variables by selecting the variables and clicking the “+” button. A new Data View is created with the selected variables.

You can also drag/drop variables between data views.

When a new Data View is created, it appears next to the original Data View. If the original Data View is docked, the new Data View is tabbed next to the original view; if undocked, the new Data View is placed behind the original Data View.

Duplicated Data View windows have the same functionality as their original counterparts. You can undock or re-dock the newly created Data Views and move them to any location.

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.
Searching for Program Elements

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:

- Navigate to a function from within the Source pane using the context menu.
- Search in the Source view by highlighting a string or through the Find function.
- Use the Lookup view to search for files or functions.

See “Program Navigation” on page 52 for details on these features.
Using the CLI to Examine Data

To access all the functionality of TotalView for HPC, you can use the CLI.

**NOTE >>** For this release of NextGen TotalView for HPC, 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, NextGen TotalView for HPC transforms STL types. If you do need to look at the untransformed data structures, use the CLI's *dset* command to set the *TV::ttf* variable to *false*:

```
CLI: dset TV::ttf { true | false }
```

Following pointers in an STL data structure to retrieve values can be time-consuming. By default, NextGen TotalView for HPC only follows 500 pointers. You can change this by altering the value of the *TV::ttf_max_length* variable.

**RELATED TOPICS**

| General information on creating custom type transformations | "Creating Type Transformations" in the NextGen TotalView for HPC Reference Guide |
| Transforming C++ types | “Displaying C++ Types” in the chapter “Examining and Editing Data and Program Elements” in the TotalView for HPC User Guide |

**Changing Size and Precision**

You can change the format that NextGen TotalView for HPC 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*”.
```
Displaying Variables

Displaying Program Variables

Display local and global variables using `dprint`:

```
CLI: dprintvariable
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 NextGen TotalView for HPC 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.
Chapter 7

Debugging Python

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 \installdir\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:**
```bash
sudo yum install python-devel
sudo debuginfo-install glibc
sudo debuginfo-install python
```

**Ubuntu:**
```bash
sudo apt-get install python-dev
sudo apt-get install python-db
```

Python can also be built to support debugging by configuring it with the `--with-pydebug` flag. See Building and Using a Debug Version of Python for details on building a debug version of the interpreter.

Limitations

The following functionality and limitations exist:

- **Python version**: Python 2.7, and Python 3.5 and above.
- **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 and ctypes to call functions in DLLs or shared libraries. 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 support@roguewave.com.
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 72 – Set up a Python Debugging Session**

   ![Program Session](image)

   **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 -newUI --args /usr/bin/python test.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 73 – Create a Breakpoint on a Python Extension Function
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 74 – Stopped at Python extension function and clean integrated call stack**

Note the Call Stack in Figure 74 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 75 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 75 – 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 *NextGen TotalView for HPC Reference Guide*
| General information on creating custom type transformations | "Creating Type Transformations" in the *NextGen TotalView for HPC Reference Guide*
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 76 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 76 – Comparing Python and C/C++ variables in the Data View
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 ReplayEngine in this user guide to learn more about reverse debugging. For information on MemoryScape, see the book Debugging Memory Problems with MemoryScape available on the Rogue Wave documentation page on the website.
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>
<thead>
<tr>
<th>Python C/C++ &quot;Glue&quot; Technology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctypes</td>
<td>A foreign function library for Python. <a href="https://docs.python.org/3/library/ctypes.html">https://docs.python.org/3/library/ctypes.html</a></td>
</tr>
<tr>
<td>Cython</td>
<td>A superset of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes. <a href="https://cython.org/">https://cython.org/</a></td>
</tr>
<tr>
<td>SWIG</td>
<td>A software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python. <a href="http://www.swig.org/Doc3.0/Python.html">http://www.swig.org/Doc3.0/Python.html</a></td>
</tr>
<tr>
<td>PyQt/PySide and SIP</td>
<td>SIP is a tool that makes it easy to create Python bindings for C and C++ libraries. <a href="https://www.riverbankcomputing.com/software/sip/intro">https://www.riverbankcomputing.com/software/sip/intro</a> <a href="https://www.riverbankcomputing.com/static/Docs/sip/">https://www.riverbankcomputing.com/static/Docs/sip/</a></td>
</tr>
</tbody>
</table>
**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 6: Python Extension Filters Supported by TotalView**

<table>
<thead>
<tr>
<th>Python C/C++ &quot;Glue&quot; Technology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctypes</td>
<td>A foreign function library for Python. <a href="https://docs.python.org/3/library/ctypes.html">https://docs.python.org/3/library/ctypes.html</a></td>
</tr>
<tr>
<td>SWIG</td>
<td>A software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python. <a href="http://www.swig.org/Doc3.0/Python.html">http://www.swig.org/Doc3.0/Python.html</a></td>
</tr>
</tbody>
</table>

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.
Access to the CLI

NextGen TotalView for HPC's default settings display the Command Line view, which provides access to the CLI.
If the Command Line view is not present when you start NextGen TotalView for HPC, 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” on page 193.

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 77 for **dhistory**.
The Command Line view has two main uses, as represented in Figure 78.

Figure 78 – Uses of the Command Line View

- 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 NextGen TotalView for HPC 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 78, 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
• Examples of Using the CLI
About the CLI and Tcl

The CLI is built in version 8.0 of Tcl, so NextGen TotalView for HPC 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 NextGen TotalView for HPC

Figure 79 illustrates the relationships between the CLI, the GUI, the TotalView core, and your program:

Figure 79 – How the CLI Interacts with NextGen TotalView for HPC

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 NextGen TotalView for HPC displays by setting the `VERBOSE` variable to `WARNING` or `ERROR`. (For more information, see the “TotalView Variables” chapter in the NextGen TotalView for HPC 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 NextGen TotalView for HPC 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 NextGen TotalView for HPC, except those that have to do with the GUI. (In some cases, NextGen TotalView for HPC 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 *NextGen TotalView for HPC 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 NextGen TotalView for HPC 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 `srun` 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*.

```bash
% 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 real16_array (i, j) = 4.093215 * j+2
76 endif
77 26 continue
78 27 continue
79
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 `drerun` command. The following example continues where the previous example left off:

```
d1.<> dkill # kills process
79
80@> do 40 i = 1, 500
81 denorms(i) = x'00000001'
82
83@> 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
```

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 GUI, NextGen TotalView for HPC also writes this information to the Logger 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 *NextGen TotalView for HPC 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. NextGen TotalView for HPC 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`, `drerun`, `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 NextGen TotalView for HPC 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 NextGen TotalView for HPC state variables. (The variables that appear in other namespaces are usually related to NextGen TotalView for HPC preferences.) NextGen TotalView for HPC 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 *NextGen TotalView for HPC Reference Guide*, lets you create your own aliases.

For example, the following command halts the current group:

```sh
dfocus g dhalt
```

Using an abbreviation is easier. The following command does the same thing:

```sh
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 NextGen TotalView for HPC), or one that was created as NextGen TotalView for HPC loaded the program.

- **Spawned process**
  A new process created by a process executing under CLI control.

NextGen TotalView for HPC 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 NextGen TotalView for HPC organizes them into groups

“Share Groups” on page 80

Information on the NextGen TotalView for HPC thread/process width

“Debugging Command Width” on page 78

**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.

NextGen TotalView for HPC thread ID
This is usually identical to the system TID. On some systems (such as AIX) where the threads have no obvious meaning, NextGen TotalView for HPC 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 NextGen TotalView for HPC 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 NextGen TotalView for HPC PID does not change. The system PID changes, however, since the operating system has created a new target process.
Controlling Program Execution

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, NextGen TotalView for HPC must also determine how other threads and processes respond, stopping some and letting others continue to run.

### RELATED TOPICS

<table>
<thead>
<tr>
<th>Stepping commands</th>
<th>&quot;Other Considerations When Stepping Through Code&quot; on page 80</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <strong>dload</strong> command</td>
<td><strong>dload</strong> in &quot;CLI Commands&quot; in the <em>NextGen TotalView for HPC Reference Guide</em></td>
</tr>
<tr>
<td>The <strong>dattach</strong> command</td>
<td><strong>dattach</strong> in &quot;CLI Commands&quot; in the <em>NextGen TotalView for HPC Reference Guide</em></td>
</tr>
<tr>
<td>The <strong>drun</strong> command</td>
<td><strong>drun</strong> in &quot;CLI Commands&quot; in the <em>NextGen TotalView for HPC Reference Guide</em></td>
</tr>
<tr>
<td>The <strong>dkill</strong> command</td>
<td><strong>dkill</strong> in &quot;CLI Commands&quot; in the <em>NextGen TotalView for HPC Reference Guide</em></td>
</tr>
</tbody>
</table>

### 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 *NextGen TotalView for HPC 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 different 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 NextGen TotalView for HPC Reference Guide) stops the process when the program reaches a location in the source code.

- A **watchpoint** (see `dwatch` in the NextGen TotalView for HPC Reference Guide) stops the process when the value of a variable is changed.

- A **barrierpoint** (see `dbarrier` in the NextGen TotalView for HPC 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 then on individual threads.)

- An **evalpoint** (see `dbreak` in the NextGen TotalView for HPC 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 NextGen TotalView for HPC. You can use it and never use the NextGen TotalView for HPC 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 NextGen TotalView for HPC' built-in CLI commands.

This section contains macros that show how the CLI programmatically interacts with your program and with NextGen TotalView for HPC. 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 NextGen TotalView for HPC Reference Guide's entry for the EXECUTABLE_PATH variable

```tcl
# 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 NextGen TotalView for HPC 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 (0x0000001)
        (2) = 2 (0x0000001)
        (3) = 3 (0x0000001)
    }
    d1.<> array_set list 2 3 99
    d1.<> dprint list3
list3 = {
    (1) = 1 (0x00000001)
    (2) = 99 (0x0000063)
    (3) = 99 (0x0000063)
}

For more information on slices, see the section “Displaying Array Slices,” in the TotalView for HPC User Guide, available on the Rogue Wave web site.

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 pf2Dslice {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.<> pf2Dslice a 1 4 1 4
    0.841470956802 0.909297406673 0.141120001673-0.756802499294
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:

```tcl
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.

```tcl
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” on page 87.

Following this macro is an excerpt from a program that uses it.

```bash
#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 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 NextGen TotalView for HPC functions, variables, and statements. Fortunately, you only use this kind of mixture in a few places and you'll know what you're doing.
Chapter 9

Controlling fork, vfork, and execve Handling

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 7: exec_handling and fork_handling Command Options and State Variables

<table>
<thead>
<tr>
<th>Command Options</th>
<th>CLI State Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>-exec_handling</td>
<td>TV::exec_handling exec-handling-list</td>
</tr>
<tr>
<td>-fork_handling</td>
<td>TV::fork_handling fork-handling-list</td>
</tr>
</tbody>
</table>

The lists `exec-handling-list` and `fork-handling-list` are Tcl lists of `regexp` and `action` pairs. Each `regexp` 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

| The state variables | TV::exec_handling exec-handling-list and TV::fork_handling fork-handling-list |

*totalview* command options

| -exec_handling exec-handling-list and -fork_handling fork-handling-list |

Setting breakpoints when using *fork()* and *execve()*

| “Setting Breakpoints When Using the fork()/execve() Functions” on page 96 |

Linking with the dbfork library

| “Linking with the dbfork Library” on page 298 |

---

**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:

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

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:

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

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'
d1.> 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
d1.> ST
1   (0)        Nonexistent [bash]
2  (20053)    Stopped [bash<tx_fork_exec><tx_hello>.1]
   2.1 (20053/20053) Stopped PC=0x7f70517dd210
```
ReplayEngine is embedded functionality on Linux x86 and x86-64 platforms that allows you go to backwards in a debugging session.

**Note:** If your platform does not support ReplayEngine, the ReplayEngine toolbar and ReplayEngine-related menu items do not display.

To make best use of this functionality, see “How ReplayEngine Works” on page 217.

For information on using ReplayEngine in a debugging session, see “Using ReplayEngine” on page 223.

ReplayEngine complements NextGen TotalView for HPC, so this discussion assumes a working knowledge of how the NextGen TotalView for HPC product works.

## How ReplayEngine Works

### Play It Backwards

The hardest step in locating software bugs involves working backward from a failure to identify the error that caused it. Conventional debugging techniques don’t make it easy to find the cause of an error, as they allow you to control program execution only in a forward direction.

Instead of going back to the beginning to try to recreate the conditions of a problem, ReplayEngine starts from the point of failure and works backward in time to find the cause. Recreating the conditions of a crash, sometimes the hardest problem in conventional forward debugging, is no longer necessary. You can now move to locate errors that occurred long before the failure they caused.
The Process of Recording and Playback

In order to move backward in your program, ReplayEngine saves state information as your program executes. This information includes the order in which your program executes and any changes to its data. When ReplayEngine is saving state information, it’s in record mode.

The saved state information is the program’s execution history. You can save the execution history at any time and then reload the recording when debugging the executable in a subsequent session.

Using a ReplayEngine command, either by clicking a toolbar button or by entering a command into the CLI, shifts ReplayEngine into replay mode. In this mode, you can move to any previously executed statement, at which point ReplayEngine displays its saved state information. The information displayed in replay mode is identical to the information displayed in record mode.

Most debugging commands work the same in replay mode as in record mode. Commands such as viewing a variable or setting a breakpoint work as you would expect. Debugging commands that do not work as expected are those that change or alter a recorded state. Typically, these are commands that:

- Change a variable’s value.
- Call functions that alter memory.
- Run threads asynchronously.

If your program calls a routine that displays information, in replay mode the routine will not display this information. For example, suppose your program calls printf(). When the printf() is executed in record mode, it writes text. However, when the printf() is replayed, the text is not rewritten. Similarly, if your program unlinks a file in record mode, the file remains always unlinked in replay mode. That is, in replay mode, the file will be unlinked even if you move back to before the unlink statement.

When executing in record mode, your program runs more slowly than without ReplayEngine turned on. Usually, you will not notice the extra execution time. However, when you are in replay mode, the computational overhead required to recreate the program’s state may be noticeable. When it needs extra time, ReplayEngine displays a dialog box to cancel the operation.

System Resource Issues

ReplayEngine writes internal information in /tmp. Normally, this uses very little space, but in some situations /tmp can grow large, and if your system has a small /tmp area, ReplayEngine may fill it up. If this occurs, you can:

- Increase the amount of storage allocated to /tmp.
- Use the TMPDIR environment variable to point to another disk location.
- Define a special TotalView variable, TVD_REPLAY_TMPDIR, for ReplayEngine to use as the base directory for writing its temporary information. For example:
ReplayEngine also changes the amount of memory your program uses because it keeps history and state information in memory.

While in replay mode, ReplayEngine creates extra processes (usually around 10) but depending on the complexity of the application you are debugging, you may see more. Ignore these processes as they are used only by ReplayEngine.

### RELATED TOPICS

Controlling the history and state information storage   “Setting Preferences for ReplayEngine” on page 227

Saving and reloading execution history   “Saving and Loading the Execution History” on page 221

### Replaying Your Program

Before you can replay your program’s statements, you must stop your program’s execution. Either halt your program, or TotalView can stop execution when your program encounters a breakpoint.

### Figure 80 – ReplayEngine Toolbar in Active State

The ReplayEngine commands include:

- **Record**, a toggle that enables and disables ReplayEngine. See “Enabling and Disabling ReplayEngine” on page 223.

- **Go Back** displays the state that existed at the last action point. If no action point is encountered, ReplayEngine displays the state that existed at the start of its recorded history.

- **Prev** displays the state that existed when the previous statement executed. If that line had a function call, Prev skips over the call.

- **Unstep** displays the state that existed when the previous statement executed. If that line had a function call, ReplayEngine moves to the last statement in that function.

- **Caller** displays the state that existed before the current routine was called.

- **Back To** displays the program’s state for the line you select. This line must have executed prior to the currently displayed line. If you wish to move forward within replay mode, select a line and select the **Run To** button from the main debugging toolbar.

- **Live** shifts from replay mode to record mode. It also displays the statement that would have executed had you not moved into ReplayMode.
• **Bookmark** ( ) creates a ReplayEngine bookmark at a selected location.

• **Save** ( ) saves the current replay recording session to a file.

The above commands are also available on the Process menu, with indications of the keyboard shortcuts.

---

**NOTE >>** The ReplayEngine toolbar commands appear only if you are using TotalView on a Linux-x86 (32-bit) or Linux-x86-64 machine. On these platforms, these buttons are permanently disabled if you do not have a ReplayEngine license.

---

To move forward within the program’s history, use the **Step**, **Next**, **Run To**, and **Out** buttons. These commands do the same thing in replay or record modes.

You can also set breakpoints in previously executed statements. After setting a breakpoint, pressing the **Go** button moves you to that statement. You can transform a breakpoint to an evalpoint if the evalpoint uses simple expressions such as “**if (x==y+z) $stop**”. You cannot, however, create barrierpoints.

If you reach the line that would have been executed if you hadn’t gone into replay mode, you are automatically switched back to record mode and you can then resume program execution. You can also switch back to record mode by clicking the **Live** button.

---

**Threads and Processes**

When recording, ReplayEngine runs and records one thread at a time. In a multi-threaded or multi-process program, normally ReplayEngine decides the order in which threads are run and recorded. ReplayEngine runs all threads in succession and saves state information for each thread as it executes.

If you need to control the way threads execute, use the TotalView asynchronous threading commands while in record mode. With these commands you can:

• Single-step a process or lockstep group.

• Hold threads so they do not run.

In replay mode, all actions must occur in the same order as recorded. This implies that you cannot influence the order in which threads execute, and you cannot hold a thread.

---

**Attaching to Running Programs**

If you attach to a program, ReplayEngine begins recording that program’s execution at the time you attached to it. This implies that you cannot go back further than when you attached to the process.
Saving and Loading the Execution History

TotalView can save the current ReplayEngine execution history to a recording file at any time. The saved recording file can then be loaded into TotalView where all the replay options are available to go back and forth within the time boundaries of the saved recording.

ReplayEngine recording is also available as standalone functionality through Rogue Wave's Replicator product. Using Replicator, you can embed the recording engine directly into your application and record its execution history without the debugger involved. Replicator saves the recorded execution history to a Replay file whenever an error occurs or when you want to trigger the saving of execution history.

Once saved, load these Replay recording files into TotalView so you can easily examine and understand the execution of your program. See the Rogue Wave website (https://www.roguewave.com) for details about the Replicator product.

To save a recording, either select the **Save** button on the toolbar:

or the **Save Recording File …** menu item in the File menu.

Requesting to save a recording file results in a Save dialog:

**Figure 81 – Save ReplayEngine recording dialog**

![Save ReplaySession dialog](image)

This dialog has the expected functionality to move around the file system and save the recording file. By default, the name of the recording file is `replay_<executable-name>_<date>_<time>.recording`, as shown above.

In addition, you can use the CLI **dhistory** command:

```
dhistory -save filename
```
The *filename* can be either a path or a simple file name, in which case it is saved into the current working directory. If no *filename* is specified, the recording is saved in the current working directory as `replay_pid-hostname.recording`.

The saved recording can be loaded into NextGen TotalView for HPC as follows:

- At startup, using the same syntax as when opening a core file:
  ```
  totalview -newUI executable recording-file
  ```
  TotalView recognizes the recording file for what it is and acts appropriately.

- After TotalView is running, using the `dattach` command with option `-c`:
  ```
  dattach executable -c recording-file
  ```

- On the Start Page view by selecting **Load Core File or Replay Recording File**. (See the section “Debug a Core or Replay Recording File” on page 33.)

Performing any of the above displays the dialog for selecting the record file and application used during the recording session when the recording session was saved.

Again, TotalView recognizes it is dealing with a recording file.
Using ReplayEngine

There is very little difference between running NextGen TotalView for HPC and running ReplayEngine. With ReplayEngine enabled on a running program, use the special buttons Go Back, Prev, Unstep, Caller, or Back To to go back in your program's history to the statement you wish to examine.

Enabling and Disabling ReplayEngine

You can prepare a program for replay when you first load it into NextGen TotalView for HPC. Once the program is loaded, there are a number of ways to enable replay.

Enabling ReplayEngine at Program Load

To enable ReplayEngine when loading a program into NextGen TotalView for HPC, select the checkbox Enable reverse debugging with Replay Engine in either the Debug a Program or Attach to Process dialogs available on the Start Page.

Figure 82 – Enabling ReplayEngine from the Start Page
For a program already under TotalView control, you can use the Command Line view to enter the `dattach` command with the `-replay` option.

```bash
dattach -replay program-path
```

For a new program, ReplayEngine begins recording instructions as soon as you begin executing the program. For a running process to which you have attached, ReplayEngine starts recording the next time you restart the process.

## Enabling and Disabling ReplayEngine for a Loaded Program

Once a program is loaded into NextGen TotalView for HPC, enable and disable replay via several options.

### Enabling Replay

Replay behavior differs depending on whether or not program execution has begun.

**The program is not yet executing**

If the program is loaded but has not started executing, enable ReplayEngine in any of the following ways:

- Click the Record toolbar button
- Select the Debug > Enable ReplayEngine menu item
- Execute the CLI command `dhistory -enable`

ReplayEngine begins recording when the process starts executing. If you restart the process, ReplayEngine begins recording from the beginning of process execution.

To stop recording, exit the program and explicitly disable ReplayEngine. You cannot turn replay off while a process is executing.

**The program is executing but halted**

If a process is already executing and stopped, you can immediately enable replay with any of the methods used when your program is not yet executing — but replay will then be enabled only while the program executes that single time. At process exit and restart, ReplayEngine will no longer be enabled unless you explicitly re-enable it.

Enabling ReplayEngine during program execution also means that you cannot step backward beyond the point at which ReplayEngine was enabled.

### Disabling Replay

You cannot disable ReplayEngine for a process that is executing. You must:
1. Kill the executing process.

2. Disable ReplayEngine either by
   - Clicking the **Record** button or de-selecting the **Debug > Enable ReplayEngine** menu item, both of which are toggles.
   - Entering `dhistory -disable` in a CLI prompt focused on the process.

If you now restart the process, ReplayEngine will be disabled for the executing process.

After killing the process, you can also return to the Start Page, click the **Edit** option for your most recent session (the pencil icon), and then de-select the **Enable reverse debugging with Replay Engine** option in the resulting dialog.

**Examining Program State and History**

After enabling ReplayEngine, you can begin controlling your program's execution using the same execution commands used when ReplayEngine is not enabled. For example, you might set a breakpoint and press the **Go** button, or select a line and press the **Run To** button.

When you wish to review the program's state at some previous point, halt your program and use the **Go Back**, **Prev**, **Unstep**, **Caller**, or **Back To** buttons to go to the statement you wish to examine. These four buttons are similar to the **Next**, **Step**, **Out**, and **Run To** toolbar buttons, differing only in that the Replay buttons go backwards in the program's history. The **Process** pull-down menu contains the menu bar equivalents to these commands.

While you are in replay mode, note that the **Next**, **Step**, **Out**, and **Run To** toolbar buttons are still active to move forward in the history.

When you're in replay mode, TotalView changes the highlight line from yellow to orange within the Source Pane.

**Figure 83 – Source View with ReplayEngine**

```c
my_pid = getpid();

if (argc > arg_count && (arg = argv[arg_count]) && isdigit(*arg))
    fork_count = atoi (arg);

if (argc_ok && argc > arg_count && (arg = argv[arg_count]) && isdigit(*arg))
    threads_per_copy = atoi (arg);

if (threads_per_copy > FORK_LOOP_MAX_THREADS)
    fprintf(stderr, "fork loop: 'threads_per_copy' limited to %d", FORK_LOOP_MAX_THREADS);  
    threads_per_copy = FORK_LOOP_MAX_THREADS;

fork_count++;
```
The Source View always shows the last line executed – the “Live” location – within record mode using the symbol. When you are in replay mode, this symbol is where ReplayEngine shifts from replay mode back to record mode.

**NOTE >>** ReplayEngine supports process width only; therefore, the scoping commands at the far left side of the main toolbar have no effect in replay mode.

## Replay Bookmarks

Replay bookmarks mark a point in the execution of a program, allowing you to quickly jump back to that point in time.

**NOTE >>** Bookmarks are set at a specific point during the program's execution history. If you restart your program, it is not guaranteed that the execution history will be the same. Even if you did not recompile your application, data inputs and other factors may alter the execution paths taken and, as a result, a bookmark might not map to the same point in execution history as when you initially created it.

### Creating bookmarks

Create bookmarks at any point while stepping through the code of your program by:

- Selecting the **Bookmarks > Create Replay Bookmark...** menu item.
- Using the **Ctrl+Shift+D** keyboard shortcut.
- Clicking the **Bookmark** icon ( ) on the ReplayEngine toolbar.

You can add an optional comment from the **Create Replay Bookmark** dialog.

**Figure 84 – Create Replay Bookmark dialog**
Once the bookmark is created it displays in the **Replay Bookmarks view**.

**Figure 85 – Replay Bookmarks view**

---

**Activating bookmarks**

Activate a bookmark by double-clicking on the bookmark in the **Replay Bookmarks view**. ReplayEngine takes you to that point in your program’s execution history where you have the full power of NextGen TotalView for HPC and ReplayEngine to examine the state of your program, run forward or backward, set breakpoints, and so forth.

**Figure 86 – Activated Replay bookmark**

---

Return to the live point in your program by clicking the **Live** icon on the ReplayEngine toolbar, selecting **Live** from the **Process** menu, or using the **Alt+Shift+L** keyboard shortcut.

**Setting Preferences for ReplayEngine**

You can set these preferences for ReplayEngine:
The maximum amount of memory to allocate to ReplayEngine

The preferred behavior when the memory maximum is reached

Set memory size preference in the CLI, like so:

```bash
dset TV::replay_history_size value
```

The value can be a number, or a number followed by ‘K’ or ‘M’ for kilobytes or megabytes. The default value ‘0’ specifies to limit the maximum size by available memory only.

For example:

```bash
dset TV::replay_history_size 1024M
```

Sets the maximum history size to 1024 megabytes.

```bash
dset TV::replay_history_size 1000000
```

Sets the maximum history size to 1000000 bytes.

The behavior preference defines ReplayEngine behavior when the maximum memory size is reached. By default, the oldest history is discarded so that recording can continue. Alternatively, you can specify that the recording process simply stops when the allocated memory is used up.

For example:

```bash
dset TV::replay_history_mode 1
```

Discard oldest history and continue recording (the default).

```bash
dset TV::replay_history_size 1000000
```

Stop the process being recorded and stop recording.

**CLI Support**

CLI support is accessed through the Command Line view on the interface. If that view is not currently displayed, you can show it either by right-clicking in the menu and toolbar area and selecting the Command Line view in the context menu, or through the Window > Views menu.

- The `dload` and `dattach` CLI commands have the `-replay` option for enabling and disabling ReplayEngine. For example:
  ```bash
dload -replay myProgram
  ```
- The `dgo`, `dnext`, `dnexti`, `dout`, `dstep`, `dstepi`, and `duntil` commands let you step or run backwards by using the `-back` option. For example:
  ```bash
dnext -back
duntil -back 22
  ```
The `dhistory` command has the following options:

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

- **–enable**
  If the program has not been started, ReplayEngine is enabled when the program is started. If the program is already running, ReplayEngine is enabled immediately. Recording begins at the point that ReplayEngine was enabled and moving back beyond that point is not possible.

- **–disable**
  Disables ReplayEngine for the next restart for the process.

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

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

- **–go_live**
  Resets the process back to record mode.

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

- **–delete_bookmark ID**
  Deletes the bookmark with the given ID.

- **–clear_bookmarks**
  Deletes all Replay bookmarks.

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

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

These CLI commands are explained in detail in the `NextGen TotalView for HPC Reference Guide`. 
Known Limitations and Issues

Limitations

• **Obscure instructions:** Use of AMD 3DNow! and other extended AMD instructions is not supported (though Intel SSE, SSE2, SSE3 and SSE4 instructions are supported). Instructions that modify CS, DS, ES or SS registers are also not supported.

• **AsyncIO:** ReplayEngine does not support asynchronous IO operations. `io_cancel`, `io_destroy`, `io_getevents`, `ioperm`, `iopl`, `io_setup`, and `io_submit` system calls are all unsupported.

• **Exec:** ReplayEngine does not support the `execve` syscall, as used by libc's `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`, and `execve()` functions. If the target program attempts to issue this system call, forward execution will not be possible beyond this point (though reverse execution is still possible).

• **Obscure system calls:** Certain rarely used system calls are not supported. If the target program attempts to issue an unsupported system call, forward execution will not be possible beyond this point (though reverse execution is still possible). The following system calls are either esoteric or obsolete, and only maintained in the kernel for backward compatibility with binaries written for early 2.x series kernels: `ssetmask`, `modify_ldt`, `pivot_root`, `vm86`, and `unshare`.

• **Use of setrlimit():** If the target program uses `setrlimit` to reduce the amount of memory, processes, or other resources consumed, ReplayEngine may not be able to operate properly due to lack of resources.

• **Use of x86 inter-segment (aka 'far') jumps/calls:** ReplayEngine does not support the use of far jumps/calls in the target program. Any such attempt will result in forward execution not being able to continue from the point at which the far jump/call instruction is issued.

• **Non-executable memory:** ReplayEngine ignores the executable status of memory when running code, so code that would usually fail because it is in non-executable memory will run successfully.

• **Disk usage:** Depending on the target program, ReplayEngine can create large temporary files within `/tmp`. See System Resource Issues for information on how to use alternative temporary directories.

• **Self-modifying code:** ReplayEngine mostly works with self-modifying code, but in some situations the effects of writing into the currently executing “basic block” may be delayed (that is, writing instructions just ahead of the current program counter such that the processor executes the newly written code by virtue of “running in to” rather than “jumping to” it).

• **Shared memory accesses straddling valid and invalid pages:** Accessing shared memory where the instruction's operand straddles a page boundary such that the first part of the operand is in accessible shared memory, but the second part is in mapped shared memory which is not backed
by a valid shared object (e.g. because the file which is mapped has been truncated) should receive signal SIGBUS. Under ReplayEngine, a target program making such an access will not receive SIGBUS but will read zeros for the part of the operand that straddles into unbacked memory. Note that normal attempted access to shared memory not backed by a shared object will generate a SIGBUS as normal; the issue applies only when a single instruction's access that lies half in valid memory and half in invalid memory that should generate a SIGBUS.

- **Breakpoints**: All breakpoints used with ReplayEngine work like hardware breakpoints. In particular, if the code where the breakpoint resides is not modified, writing to that code will not remove the breakpoint, and setting a breakpoint that is not at the first byte of an instruction will have no effect.

- **System call output buffers**: Any system calls that write to memory must be passed a buffer entirely within writable memory. For example, if read() is passed an 8k buffer of which only the first 4k is in user-writable memory, if that read() would normally return 4k or fewer characters then natively it may succeed, but on ReplayEngine it will fail with EFAULT. If a system call that writes to memory is passed a buffer which is not in writable memory at all, but fails for some other reason before the kernel tries to write to the buffer, then natively it may fail with some error other than EFAULT, but on ReplayEngine it may fail with EFAULT. If two buffers which overlap are passed to a system call which writes to both of them or reads from one and writes to the other, the behavior in ReplayEngine may differ from the native behavior (although behavior in such cases is liable to vary between kernel versions, too.)

- **Adjust Flag**: According to the Intel manuals, the state of the Adjust Flag (AF) after some instructions is “undefined.” On some processor models, different executions of the same code can produce different states of AF. If the behavior of a program depends on the state of AF when it is supposed to be undefined, the program may not run correctly with ReplayEngine.

- **SIGCHLD while attaching**: If a SIGCHLD arrives for a process while ReplayEngine is in the middle of attaching to the process, the SIGCHLD may be silently lost. Once the process has been attached to, SIGCHLD is handled normally.

- **Loading a previous recording session**: The successful reloading and debugging of a previously saved replay recording session requires that both the environment that saved the session and the environment replaying the recording session be exactly the same.

---

**Performance Issues**

**High TLB rates with certain multi-threaded target programs**

When reverse debugging an application in which many threads make frequent system calls on a multi-processor platform, binding the application process to a single processor can improve performance. This is because such applications put stress on ReplayEngine's heap management, which in turn stresses the processor's TLB (transla-
tion lookaside buffer). If the application is bound to a single processor, it is less likely to suffer TLB misses caused by process migration. Since user threads are automatically serialized during reverse debugging, there is no loss of concurrency due to binding.

If the application is to be launched under NextGen TotalView for HPC, one way to accomplish binding is to preface the totalview command with a taskset(1) command specifying a single processor. For example:

taskset --cpu-list 3 totalview -newUI -replay myapp

To accomplish binding when NextGen TotalView for HPC is to be attached to a running application, find the PID (process identifier) of the application process, and use taskset to bind that process to a single processor before attaching to it with NextGen TotalView for HPC. For example:

taskset --pid --cpu-list 3 <PID of myapp>

We have noticed the need for such binding when debugging MySQL applications with ReplayEngine.
Preferences

About Preferences

Customize aspects of TotalView using the Settings toolbar or by selecting File Preferences, including:

- “Display Settings” on page 235
- “Tool Bar” on page 237
- “Search Path” on page 238
- “Parallel Configuration” on page 241
Display Settings

- **User Interface Style**, NextGen or Classic

**NOTE** >> These preference settings require a restart.

**Figure 87 – 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.
Tool Bar

Figure 88 – Preferences: Tool Bar Visibility

This dialog displays or hides the toolbars, which you can also do 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.
Search Path

The Search Path tab customizes the locations in which NextGen TotalView for HPC 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:

```dlang
dl.<> dset *PATH*
EXECUTABLE_PATH {}
EXECUTABLE_SEARCH_PATH ${EXECUTABLE_PATH}:${PATH}:
OBJECT_SEARCH_PATH ${COMPILATION_DIRECTORY}:${EXECUTABLE_PATH}:${EXECUTABLE_DIRECTORY}:
$links(${EXECUTABLE_DIRECTORY}):.:${TOTALVIEW_SRC}
SHARED_LIBRARY_SEARCH_PATH ${EXECUTABLE_DIRECTORY}
SOURCE_SEARCH_PATH ${COMPILATION_DIRECTORY}:${EXECUTABLE_PATH}:${EXECUTABLE_DIRECTORY}:
$links(${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 89 – 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 90 – 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 91 – 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 menu 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 92 – Parallel Configuration Preferences
• **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**

<table>
<thead>
<tr>
<th>Linking with the dbfork Library</th>
<th>“Linking with the dbfork Library” on page 298 and the TV::dbfork variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>More about controlling how TotalView attaches to processes</td>
<td>The TV::parallel_attach variable in the TotalView Reference Guide</td>
</tr>
<tr>
<td>More about how TotalView stops processes.</td>
<td>The TV::parallel_stop variable in the TotalView Reference Guide</td>
</tr>
</tbody>
</table>
This part introduces the TotalView CUDA debugger and includes the following chapters:

“About the TotalView CUDA Debugger” on page 244
   Introduces the CUDA debugger, including features, requirements, installation and drivers.

“CUDA Debugging Model and Unified Display” on page 248
   Explores setting and viewing action points in CUDA code.

“CUDA Debugging Tutorial” on page 254
   Discusses how to build and debug a simple CUDA program, including compiling, controlling execution, and analyzing data.

“CUDA Problems and Limitations” on page 272
   Issues related to limitations in the NVIDIA environment.

“Sample CUDA Program” on page 275
   Compilable sample CUDA program.
About the TotalView CUDA Debugger

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 hosts, see the NextGen TotalView for HPC Supported Platforms Guide.
• Tesla, Fermi, Kepler, Pascal, or Volta hardware supported by NVIDIA
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:
- 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.
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” on page 249.
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 93 illustrates how TotalView would group the Linux and CUDA threads.

**Figure 93 – 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 for HPC 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” on page 87.

RELATED TOPICS

| Sliding breakpoints | “Sliding Breakpoints” on page 88 |
| Pending breakpoints | “Pending Breakpoints” on page 91 |
| Pending evalpoints | “Creating a Pending Evalpoint” on page 100 |
| How the unified Source view displays breakpoints in dynamically-loaded code | “Unified Source View and Breakpoint Display” on page 252 |
| 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 |
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” on page 7.

This is particularly visible when breakpoints are set. For example, Figure 94 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 94 – Source view before CUDA kernel launch

![Figure 94 - Source view before CUDA kernel launch](image)

After CUDA kernel launch, Figure 95 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 95 – Source view after CUDA kernel launch

![Figure 95 - Source view after CUDA kernel launch](image)

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>More on the unified Source view display</td>
<td>“Unified Source View Display” on page 8</td>
</tr>
<tr>
<td>The CUDA share group model</td>
<td>“The TotalView CUDA Debugging Model” on page 249</td>
</tr>
<tr>
<td>Using <strong>dactions</strong> to display pending and mixed breakpoint</td>
<td>“Examples of Actions Points in Both Host and</td>
</tr>
<tr>
<td>detail before and after CUDA code has loaded.</td>
<td>Dynamically Loaded Code” in the <strong>dactions</strong> entry</td>
</tr>
<tr>
<td></td>
<td>in the <em>TotalView Reference Guide</em></td>
</tr>
</tbody>
</table>
This chapter discusses how to build and debug a simple CUDA program using TotalView.

### 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” on page 275.

### Compiling for Fermi

To compile for Fermi, use the following compiler option:
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 96.

**Figure 96 – Source view opened on CUDA host code**

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” on page 88.)

Viewing GPU Threads

Once the CUDA kernel starts executing, it will hit the breakpoint planted in the GPU code, as shown in Figure 97.

Figure 97 – CUDA thread stopped at a breakpoint, focused on GPU thread &lt;&lt;(0,0,0),(0,0,0)&gt;&gt;

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 Var panel 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:

![Call Stack](image)

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 98 – CUDA Thread IDs**

![Processes & Threads](image)

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:
**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: \( \text{grid} = \text{grid index} \) and block indices: \( \text{block index} = \text{block index} \). 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 100 – 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 101. 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 101 – 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 8.

Table 8: Supported Type Storage Qualifiers

<table>
<thead>
<tr>
<th>Storage Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@code</code></td>
<td>An offset within executable code storage</td>
</tr>
<tr>
<td><code>@constant</code></td>
<td>An offset within constant storage</td>
</tr>
</tbody>
</table>
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.

### 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 102 shows the result of casting a float in generic storage to a 2x2 array of floats in generic storage.

Figure 102 – 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:
```bash
d1.<> dfocus .-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 {*}(@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 {*(@parameter Matrix*)0x10}
*(@parameter 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:

```c
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:

```c
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:

```c
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 103.

Figure 103 – PTX register variables: one live, one dead
Enabling CUDA Memory Checker Feature

You can detect global memory addressing violations and misaligned global memory accesses by enabling the CUDA Memory Checker feature.

To enable the feature, use one of the following:

- 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:

```bash
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 9: CUDA Exception Codes**

<table>
<thead>
<tr>
<th>Exception code</th>
<th>Error Precision</th>
<th>Error Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_EXCEPTION_0: “Device Unknown Exception”</td>
<td>Not precise</td>
<td>Global error on the GPU</td>
<td>An application-caused global GPU error that does not match any of the listed error codes below.</td>
</tr>
<tr>
<td>CUDA_EXCEPTION_1: “Lane Illegal Address”</td>
<td>Precise (Requires memcheck on)</td>
<td>Per lane/thread error</td>
<td>A thread has accessed an illegal (out of bounds) global address.</td>
</tr>
<tr>
<td>CUDA_EXCEPTION_2: “Lane User Stack Overflow”</td>
<td>Precise</td>
<td>Per lane/thread error</td>
<td>A thread has exceeded its stack memory limit.</td>
</tr>
<tr>
<td>CUDA_EXCEPTION_3: “Device Hardware Stack Overflow”</td>
<td>Not precise</td>
<td>Global error on the GPU</td>
<td>The application has triggered a global hardware stack overflow, usually caused by large amounts of divergence in the presence of function calls.</td>
</tr>
<tr>
<td>CUDA_EXCEPTION_4: “Warp Illegal Instruction”</td>
<td>Not precise</td>
<td>Warp error</td>
<td>A thread within a warp has executed an illegal instruction.</td>
</tr>
<tr>
<td>CUDA_EXCEPTION_5: “Warp Out-of-range Address”</td>
<td>Not precise</td>
<td>Warp error</td>
<td>A thread within a warp has accessed an address that is outside the valid range of local or shared memory regions.</td>
</tr>
</tbody>
</table>
CUDA Debugging Tutorial / GPU Error Reporting

### Table 9: CUDA Exception Codes

<table>
<thead>
<tr>
<th>Exception code</th>
<th>Error Precision</th>
<th>Error Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_EXCEPTION_6:</td>
<td>Not precise</td>
<td>Warp error</td>
<td>A thread within a warp has accessed an incorrectly aligned address in the local or shared memory segments.</td>
</tr>
<tr>
<td></td>
<td>“Warp Misaligned Address”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA_EXCEPTION_7:</td>
<td>Not precise</td>
<td>Warp error</td>
<td>A thread within a warp has executed an instruction that attempts to access a memory space not permitted for that instruction.</td>
</tr>
<tr>
<td></td>
<td>“Warp Invalid Address Space”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA_EXCEPTION_8:</td>
<td>Not precise</td>
<td>Warp error</td>
<td>A thread within a warp has advanced its PC beyond the 40-bit address space.</td>
</tr>
<tr>
<td></td>
<td>“Warp Invalid PC”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA_EXCEPTION_9:</td>
<td>Not precise</td>
<td>Warp error</td>
<td>A thread within a warp has triggered a hardware stack overflow.</td>
</tr>
<tr>
<td></td>
<td>“Warp Hardware Stack Overflow”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA_EXCEPTION_10:</td>
<td>Not precise</td>
<td>Global error</td>
<td>A thread has accessed an illegal (out of bounds) global address. For increased precision, enable memcheck.</td>
</tr>
<tr>
<td></td>
<td>“Device Illegal Address”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA_EXCEPTION_11:</td>
<td>Precise (Requires memcheck on)</td>
<td>Per lane/thread error</td>
<td>A thread has accessed an incorrectly aligned global address.</td>
</tr>
<tr>
<td></td>
<td>“Lane Misaligned Address”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA_EXCEPTION_12:</td>
<td>Precise</td>
<td>Per warp</td>
<td>Any thread in the warp has hit a device side assertion.</td>
</tr>
<tr>
<td></td>
<td>&quot;Warp Assert&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA_EXCEPTION_13:</td>
<td>Precise (Requires memcheck on)</td>
<td>Per lane/thread error</td>
<td>A thread has corrupted the heap by invoking free with an invalid address (for example, trying to free the same memory region twice)</td>
</tr>
<tr>
<td></td>
<td>&quot;Lane Syscall Error&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA_EXCEPTION_14:</td>
<td>Not precise</td>
<td>Per warp</td>
<td>A thread has accessed an illegal (out of bounds) global/local/shared address. For increased precision, enable the CUDA memcheck option. See “Enabling CUDA Memory Checker Feature” on page 267.</td>
</tr>
<tr>
<td></td>
<td>&quot;Warp Illegal Address&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA_EXCEPTION_15:</td>
<td>Precise</td>
<td>Per host thread</td>
<td>A host thread has attempted to access managed memory currently used by the GPU.</td>
</tr>
<tr>
<td></td>
<td>&quot;Invalid Managed Memory Access&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CUDA Problems and Limitations

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:

```bash
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_SIZE x BLOCK_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 Bs 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;
}
PART IV

Appendices

Appendix A, “More on Expressions,” on page 281
Provides examples of expressions, using program elements in expressions, and a list of TotalView's built-in variables and statements for use in expressions.

Appendix B, “Compiling for Debugging,” on page 290
Specifics on how to compile your programs for debugging in TotalView.

Appendix C, “Platform-Specific Topics,” on page 296
Issues specific to individual platforms.

Appendix D, “Resources,” on page 300
Lists conventions used in this document, contact information, and documentation available for TotalView for HPC, useful if you are using features not yet supported in the TotalView UI by invoking commands through the Command Line Interface (CLI).
Appendix A

More on Expressions

Overview

This appendix includes detail and troubleshooting discussions on using expressions in TotalView.

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: \texttt{my\_var[\text{cntr}]} and \texttt{my\_var[++\text{cntr}]} If \texttt{cntr} equals 3, you’ll see the values of \texttt{my\_var[3]} and \texttt{my\_var[4]}. However, since \texttt{cntr} 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” on page 283
- “Using Built-In Statements” on page 284

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>$clid</td>
<td>The cluster ID. (Interpreted expressions only.)</td>
</tr>
<tr>
<td>$duid</td>
<td>The TotalView-assigned Debugger Unique ID (DUID). (Interpreted expressions only.)</td>
</tr>
<tr>
<td>$newval</td>
<td>The value just assigned to a watched memory location. (Watchpoints only.)</td>
</tr>
<tr>
<td>$nid</td>
<td>The node ID. (Interpreted expressions only.)</td>
</tr>
<tr>
<td>$oldval</td>
<td>The value that existed in a watched memory location before a new value modified it. (Watchpoints only.)</td>
</tr>
<tr>
<td>$pid</td>
<td>The process ID.</td>
</tr>
<tr>
<td>$processduid</td>
<td>The DUID (debugger ID) of the process. (Interpreted expressions only.)</td>
</tr>
<tr>
<td>$systid</td>
<td>The thread ID assigned by the operating system. When this is referenced from a process, TotalView throws an error.</td>
</tr>
<tr>
<td>$tid</td>
<td>The thread ID assigned by TotalView. When this is referenced from a process, TotalView throws an error.</td>
</tr>
</tbody>
</table>

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:

```c
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:

```plaintext
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`.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$count expression</code></td>
<td>Sets a process-level countdown breakpoint.</td>
</tr>
<tr>
<td><code>$countprocess expression</code></td>
<td>When any thread in a process executes this statement for the number of times specified by <code>expression</code>, the process stops. The other processes in the control group continue to execute.</td>
</tr>
<tr>
<td><code>$countall expression</code></td>
<td>Sets a program-group-level countdown breakpoint. All processes in the control group stop when any process in the group executes this statement for the number of times specified by <code>expression</code>.</td>
</tr>
<tr>
<td><code>$countthread expression</code></td>
<td>Sets a thread-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by expression, 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 <code>$countprocess</code>. A thread evaluates expression when it executes <code>$count</code> for the first time. This expression must evaluate to a positive integer. When TotalView first encounters this variable, it determines a value for <code>expression</code>. TotalView does not reevaluate until the expression actually stops the thread. This means that TotalView ignores changes in the value of <code>expression</code> until it hits the breakpoint. After the breakpoint occurs, TotalView reevaluates the expression and sets a new value for this statement. The internal counter is stored in the process and shared by all threads in that process.</td>
</tr>
<tr>
<td><code>$hold</code></td>
<td>Holds the current process.</td>
</tr>
<tr>
<td><code>$holdprocess</code></td>
<td>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.</td>
</tr>
<tr>
<td>Statement</td>
<td>Use</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>$holdstopall</td>
<td>Like $hold, except that any processes in the group which are running are stopped. The other processes in the group are not automatically held by this call—they are just stopped.</td>
</tr>
<tr>
<td>$holdprocessstopall</td>
<td></td>
</tr>
<tr>
<td>$holdthread</td>
<td>Freezes the current thread, leaving other threads running.</td>
</tr>
<tr>
<td>$holdthreadstop</td>
<td>Like $holdthread, except that it stops the process. The other processes in the group are left running.</td>
</tr>
<tr>
<td>$holdthreadstopprocess</td>
<td>Like $holdthreadstop, except that it stops the entire group.</td>
</tr>
<tr>
<td>$holdthreadstopall</td>
<td>Sets a process-level breakpoint. The process that executes this statement stops; other processes in the control group continue to execute.</td>
</tr>
<tr>
<td>$stop $stopprocess</td>
<td>Sets a process-level breakpoint. The process that executes this statement stops; other processes in the control group continue to execute.</td>
</tr>
<tr>
<td>$stopall</td>
<td>Sets a program-group-level breakpoint. All processes in the control group stop when any thread or process in the group executes this statement.</td>
</tr>
<tr>
<td>$stopthread</td>
<td>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 $stopprocess.</td>
</tr>
<tr>
<td>$visualize(expression[,slice])</td>
<td>Visualizes the data specified by expression and modified by the optional slice value. Expression and slice must be expressed using the code fragment's language. The expression must return a dataset (after modification by slice) that can be visualized. slice 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.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO 100 I=1,10</td>
<td>DO100I=1,10</td>
</tr>
<tr>
<td>CALL RINGBELL</td>
<td>CALL RING BELL</td>
</tr>
<tr>
<td>X .EQ. 1</td>
<td>X.EQ.1</td>
</tr>
</tbody>
</table>

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, dble, 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

Overview

This appendix provides some important information on how to prepare your applications to be debugged with the NextGen TotalView for HPC debugger. The information covers only currently supported platforms. Please see the Platforms Guide for information.

The two topics in the appendix are:

- "Compiling with Debugging Symbols" on page 290
- "Maintaining Debug Information Separate from an Executable" on page 293

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:

```
c -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 any program with NextGen TotalView for HPC.
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.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC C</td>
<td><code>gcc -g source.c</code></td>
</tr>
<tr>
<td>GCC C++</td>
<td><code>g++ -g source.cxx</code></td>
</tr>
<tr>
<td>clang C</td>
<td><code>clang -g source.c</code></td>
</tr>
<tr>
<td>clang C++</td>
<td><code>clang++ -gsource.cxx</code></td>
</tr>
<tr>
<td>Oracle Studio C</td>
<td><code>cc -g source.c</code></td>
</tr>
<tr>
<td>Oracle Studio C++</td>
<td><code>CC -g source.cxx</code></td>
</tr>
<tr>
<td>Intel C++ Compiler</td>
<td><code>icc -g source.cxx</code></td>
</tr>
<tr>
<td>PGI CC</td>
<td><code>pgcc -g source.c</code></td>
</tr>
<tr>
<td>PGI C++</td>
<td><code>pgc++ -gsource.cxx</code></td>
</tr>
</tbody>
</table>

The following table lists the procedures to compile **Fortran programs** on Linux platforms.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absoft Fortran 77</td>
<td><code>f77 -g program.f f77 -g program.for</code></td>
</tr>
<tr>
<td>Absoft Fortran 90</td>
<td><code>f90 -g program.f90</code></td>
</tr>
<tr>
<td>Absoft Fortran 95</td>
<td><code>f95 -g program.f95</code></td>
</tr>
<tr>
<td>G77</td>
<td><code>g77 -g program.f</code></td>
</tr>
<tr>
<td>Intel Fortran Compiler</td>
<td><code>ifort -g program.f</code></td>
</tr>
<tr>
<td>Lahey/Fujitsu Fortran</td>
<td><code>lf95 -g program.f</code></td>
</tr>
<tr>
<td>PGI Fortran 77</td>
<td><code>pgf77 -g program.f</code></td>
</tr>
<tr>
<td>PGI Fortran 90</td>
<td><code>pgf90 -g program.f</code></td>
</tr>
</tbody>
</table>
The following table lists the procedures to compile programs on ARM64 platforms.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC C</td>
<td>gcc -g program.c</td>
</tr>
<tr>
<td>GCC C++</td>
<td>g++ -g program.cxx</td>
</tr>
<tr>
<td>G77</td>
<td>g77 -g program.f</td>
</tr>
</tbody>
</table>

The following table lists the procedures to compile programs on Mac OS platforms.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absoft Fortran 77</td>
<td>f77 -gprogram.f f77 -gprogram.for</td>
</tr>
<tr>
<td>Absoft Fortran 90</td>
<td>f90 -gprogram.f90</td>
</tr>
<tr>
<td>Absoft Fortran 95</td>
<td>f95 -g program.f95</td>
</tr>
<tr>
<td>GCC C</td>
<td>gcc -g program.c</td>
</tr>
<tr>
<td>GCC C++</td>
<td>g++ -gprogram.cxx</td>
</tr>
<tr>
<td>GCC Fortran</td>
<td>gfortran -gprogram.f</td>
</tr>
<tr>
<td>clang C</td>
<td>clang -g source.c</td>
</tr>
<tr>
<td>clang C++</td>
<td>clang++ -gsource.cxx</td>
</tr>
<tr>
<td>Intel C++ Compiler</td>
<td>icc -g source.cxx</td>
</tr>
<tr>
<td>Intel Fortran Compiler</td>
<td>ifort -g program.f</td>
</tr>
</tbody>
</table>
Maintaining Debug Information Separate from an Executable

Because debug information embedded in an executable can be very large, some versions of Linux support stripping this information from the executable and placing it into a separate file. This file is then referenced within the executable using either a build ID section or a debug link section (or both) to identify the location and name of the separate debug file. The stripped image file will normally take up less space on the disk, and if you want the debug information, you can also install the corresponding .debug file.

The way this works with TotalView is controlled by a series of state variables and command line options discussed in Controlling Separate Debug Files.

Create this file on Linux systems that have an objcopy that supports the -add-gnu-debuglink and -only-keep-debug command-line options. If objcopy -help mentions these options, creating this file is supported. See man objcopy for more details.

To create a separate file containing debug information:

1. Create a .debug copy of the executable or shared library. This second file is a regular executable but will contain only debugging symbol table information, with no code or data.
2. Create a stripped copy of the image file, and add to the stripped executable a .gnu_debuglink section that identifies the .debug file.

   **NOTE >>** The technique for creating a build ID separate debug information file is different and more complex than that for creating a debug link. Consult your system documentation for how to create a separate debug information file using the build ID method.

3. Distribute the stripped image and .debug files separately.

   For example:

   ```
   objcopy --only-keep-debug hello hello.debug
   objcopy --strip-all hello
   objcopy --add-gnu-debuglink=hello.debug hello
   ```

   The code above uses objcopy to:

   1. Create a separate debug file for an executable hello named hello.debug, containing only debug symbols and information.
   2. Strip the debug information from the hello executable.
   3. Add a .gnu_debuglink section to the hello executable.
Controlling Separate Debug Files

The following command line options and CLI variables control how TotalView handles separate debug files.

- **Controls whether TotalView looks for either a build ID or a .gnu_debuglink section in image files:**
  - Command line options `-gnu_debuglink` and `-no_gnu_debuglink`
  - State variable `TV::gnu_debuglink`
  - This option basically turns on or off the functionality to support separate debug files.

- **Sets the search path to use when looking for debug files referenced by a .gnu_debuglink section:**
  - Command line option `-gnu_debuglink_search_path`
  - State variable `TV::gnu_debuglink_search_path`

- **Sets the search path to use when looking for debug files referenced by a .note.gnu.build-id section:**
  - Command line option `-gnu_debuglink_build_id_search_path`
  - State variable `TV::gnu_debuglink_build_id_search_path`

- ** Specifies the global debug directory:**
  - Command line option `-gnu_debuglink_global_directory`
  - State variable `TV::gnu_debuglink_global_directory`

- **Validates the separate .gnu_debuglink debug file’s checksum:**
  - Command line option `-gnu_debuglink_checksum`
  - State variable `TV::gnu_debuglink_checksum`

Searching for the Debug Files

If the `TV::gnu_debuglink` variable is `true` and if an image file contains either a `.note.gnu.build-id` or a `.gnu_debug_link` section, TotalView searches for a separate debug information file that matches the image file. TotalView will first search for the debug file using the `.note.gnu.build-id` section in the image file, if it exists. If that search fails, TotalView will search for the debug file using the `.gnu_debuglink` section in the image file, if it exists.

For the build ID method:

1. The `TV::gnu_debuglink_build_id_search_path` string is split at the colon (:) characters into a list of strings.
2. For each string on the list, "%D", "%G", and "%/" token expansion is performed to yield a list of directory names to search.

3. The list of directories is searched for the debug file path named by the .note.gnu.build-id section. The debug file path follows the pattern "\.build-id/xx/yyyy...yyy.debug", where xx are the first two hex characters of the build ID bit string, and yyyy...yy 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
Overview

This appendix provides some platform-specific information that you may find useful. The sections are:

- “Swap Space” on page 296
- “Shared Libraries” on page 297

See the NextGen 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, Next-Gen TotalView for HPC 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

NextGen TotalView for HPC supports dynamically linked executables, that is, executables that are linked with shared libraries.

When you start NextGen TotalView for HPC 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, NextGen TotalView for HPC 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 “NextGen TotalView for HPC Command Syntax” in the NextGen TotalView for HPC 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 NextGen TotalView for HPC 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:

```bash
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 NextGen TotalView for HPC dbfork library so that NextGen TotalView for HPC 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
TotalView for HPC Documentation

The following table describes all available documentation for TotalView for HPC when using the legacy UI. Much of the information is still applicable if you are using the new UI, although features not yet supported in the interface must be accessed through the Command Line Interface (CLI) in the Command Line view.

<table>
<thead>
<tr>
<th>Category</th>
<th>Title</th>
<th>Description</th>
<th>HTML</th>
<th>PDF</th>
<th>Print</th>
</tr>
</thead>
<tbody>
<tr>
<td>General TotalView for HPC Documentation</td>
<td><strong>Getting Started with TotalView for HPC Products</strong></td>
<td>Introduces the basic features of TotalView for HPC, MemoryScape, and ReplayEngine, with links for more detailed information</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>TotalView for HPC Platforms Guide</strong></td>
<td>Defines platform and system requirements for TotalView, MemoryScape, and ReplayEngine</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>TotalView for HPC Evaluation Guide</strong></td>
<td>Brochure that introduces basic TotalView for HPC features</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>User Guides</td>
<td><strong>TotalView for HPC User Guide</strong></td>
<td>Primary resource for information on using the TotalView for HPC GUI and the CLI</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
### Category

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>HTML</th>
<th>PDF</th>
<th>Print</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Debugging Memory Problems with MemoryScape</strong></td>
<td>How to debug memory issues, relevant to both TotalView for HPC and the MemoryScape standalone product</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Reverse Debugging with Replay Engine</strong></td>
<td>How to perform reverse debugging using the embedded add-on ReplayEngine</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

### Reference Guides

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>HTML</th>
<th>PDF</th>
<th>Print</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TotalView for HPC Reference Guide</strong></td>
<td>A reference of CLI commands, how to run TotalView, and platform-specific detail</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

### New Features

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>HTML</th>
<th>PDF</th>
<th>Print</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What’s new in this release</strong></td>
<td>On the landing page of the HTML documentation, lists new features for the documented release</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TotalView for HPC New Features and Change Log</strong></td>
<td>Details the changes to the product from release to release</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

### Installation Guide

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>HTML</th>
<th>PDF</th>
<th>Print</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TotalView for HPC Install Guide</strong></td>
<td>Installing TotalView for HPC and the FLEXlm license manager</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
## Conventions

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>Brackets are used when describing optional parts of a command.</td>
</tr>
<tr>
<td>arguments</td>
<td>In a command description, text in italics represents information you enter. Elsewhere, italics is used for emphasis.</td>
</tr>
<tr>
<td>Bold text</td>
<td>In a command description, 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.</td>
</tr>
<tr>
<td>Example text</td>
<td>In program listings, this represents a program or something you’d enter in response to a shell or CLI prompt. Bold text here indicates exactly what you should type. If you’re viewing this information online, example text is in color.</td>
</tr>
</tbody>
</table>

## Contacting Us

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

- **By email**: support@roguewave.com
- **By phone**: See Rogue Wave’s Technical Support page (https://www.roguewave.com/help-support/technical-support) for support numbers.

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

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

Symbols
/tmp
  use with ReplayEngine 218
$address data type 164
$char data type 164
$character data type 164
$clid built-in variable 283
$code data type 164
$complex data type 164
$complex_16 data type 164
$complex_8 data type 164
$count built-in function 102, 134, 284
$countall built-in function 284
$countthread built-in function 284
$double data type 164
$double_precision data type 165
$duid built-in variable 283
$extended data type 165
$float data type 165
$hold built-in function 284
$holdprocess built-in function 284
$holdprocessall built-in function 285
$holdstop built-in function 285
$holdstopall built-in function 285
$holdthread built-in function 285
$holdthreadstop built-in function 285
$holdthreadstopall built-in function 285
$holdthreadstopprocess built-in function 285
$int data type 165
$integer data type 165
$integer_1 data type 165
$integer_2 data type 165
$integer_4 data type 165
$integer_8 data type 165
$logical data type 165
$logical_1 data type 165
$logical_2 data type 165
$logical_4 data type 165
$logical_8 data type 165
$long data type 165
$long_long data type 165
$newval built-in function 113
$newval built-in variable 283
$newval intrinsic 111
$nid built-in variable 283
$oldval built-in function 113
$oldval built-in variable 283
$oldval intrinsic 111
$oldval watchpoint variable 113
$pid built-in variable 283
$processduid built-in variable 283
$real data type 165
$real_16 data type 165
$real_4 data type 165
$real_8 data type 165
$short data type 165
$stop built-in function 114, 285
$stop function 212
$stopall built-in function 285
$stopprocess built-in function 285
$stopthread built-in function 285
$string data type 165
$systid built-in variable 283
$tid built-in variable 283
$visualize built-in function 285
$void data type 165
$wchar data type 166
$wchar_s16 data type 166
$wchar_s32 data type 166
$wchar_u16 data type 166
$wchar_u32 data type 166
$wstring data type 166
$wstring_s16 data type 166
$wstring_s32 data type 166
$wstring_u16 data type 166
$wstring_u32 data type 166

A
-a command-line option 46, 198
  passing arguments to program 46
action point
  At Location 90
  pending breakpoint 90
Action Point > Properties
  command 96
Action Point ID attribute
  definition 83
action points
  and process/thread state 124
  and templated code 88
  CLI-available options 132
  defined 85
  disabling or deleting 62, 128
  diving on 128
  evaluation points 133
  identifiers, never reused in a session 205
  loading 136
  properties 85
  saving 136
  setting 87
  sorting 127
  state 130
  suppressing and unsuppressing 130
  tutorial 61
  unique IDs 86
  viewing variables at breakpoint 65
  width 123
Action Points view 13
adding command-line arguments 38
Index
304
adding environment variables 38
$address 164
addresses
specifying in variable window 175
advancing
and holding processes 204
program execution 204
aggregate presentation of process and thread attributes 74
aliases
built-in 201
group 201
group, limitations 201
ARGS variable 198
modifying 198
ARGS_DEFAULT variable 198
clearing 198
arguments
passing to program 46
replacing 198
ARM64
compiling with debugging symbols 292
array pointers 174
arrays
evaluating expressions 140
slice, initializing 207
slice, printing 208
writing to file 209
assigning output to variable 196
At Location dialog 90
attach options
enabling ReplayEngine 33
Attach to a Running Program dialog 29
attaching
commands 36
configuring a debug session 29
to processes 29
attaching to relatives 32
attributes
defined for processes and threads 83
different views of 77
selecting 74
setting order of 74
auto_save_breakpoints variable 136
automatic dereferencing 174
B
BackTo toolbar button (ReplayEngine) 219
barrier points 116, 118
defined (again) 116
satisfying 119
states 116
stopped process 120
barrieroers defined 205
barriers with ReplayEngine 220
branching around code 104
breakpoints
and ReplayEngine 220
conditional 102, 284
countdown 102, 284
defined 205
eample setting in multipro-
cess program 97
fork() 96
loading 136
not shared in separated children 96
saving 136
setting 87, 210
shared by default in processes 96
sliding 88
thread-specific 283
width, interaction with com-
mand width 81
built-in aliases 201
built-in functions
$count 102, 134
$countall 284
$countthread 284
$hold 284
$holdprocess 284
$holdprocessall 285
$holdstopall 285
$holdthread 285
$holdthreadstop 285
$holdthreadstopall 285
$holdthreadstopprocess 28
$hold 5
$stop 114, 285
$stopall 285
$stopprocess 285
$stopthread 285
$visualize 285
built-in variables 283
$clid 283
$duid 283
$newval 283
$nid 283
$soldval 283
$pid 283
$processduid 283
$string 164
$systid 283
$tid 283
$visualize 285
C
C/C++ statements
expression system 286
C++
compiling on x86-64 291
Call Stack view 11, 51, 63
responding to Processes and Threads view actions 78
Caller toolbar button (ReplayEngine) 219
capture command 196
changing command-line arguments 38
changing program state 192
$char data type 164
$character data type 164
chasing pointers 174
CLI commands supporting ReplayEngine 228
components 190
in startup file 193
initialization 193
invoking program from shell example 193
not a library 191
output 196
prompt 194

RogueWave
relationship to NextGen UI 191
setting action points 132
starting 193
starting from command prompt 193

CLI commands
assigning output to variable 196
-back option for replay debugging 228
capture 196
dactions-load 136
dactions-save 133, 136
dattach 30, 36
dattach-replay 228
dbarrier 116, 118, 119
dbreak 211
ddisable 120
dhistory 229
dhistory-disable 225
dhistory-replay 224
dhold 117
dkill 195
dload 35, 194, 204
dload-replay 228
dnext and dnexti -back commands 228
dprint 174, 208
drun 194
drun 194, 198
dsview 35
dset 198, 200
dstatus 120
dunhold 117
dunset 198
dup 174
dwhere 174

CLI variables
ARGS 198
ARGS_DEFAULT 198
ARGV, modifying 198
auto_save_breakpoints 136
data format 173
EXECUTABLE_PATH 25, 27, 207
gnu_debuglink 294
LINES_PER_SCREEN 197
PROMPT 200
SHARE_ACTION_POINT 96
ttf 167, 173
ttf_max_length 173
VERBOSE 192

CLI view 13
$clid built-in variable 283
cluster ID 283
$code data type 164
code constructs supported
	C/C++ 286
	Fortran 287
code fragments 283
code highlighting during ReplayEngine replay 225
code marker for end of replay (ReplayEngine) 226
code, branching around columns ascending and descending order 76
command arguments 198
clearing example 198
passing defaults 198
setting 198
Command Line view 13, 52
command prompts 200
default 200
format 200
setting 200
starting the CLI from 193
command-line options 194
-a 198
-a 46
-newUI 3
passing to debugger 46
-s startup 193
-sb 136
commands
Action Point > Properties 96
at group width 80
at process width 80
at thread width 79
debugging, described 19
File -> Manage Sessions 41
Group > Go 97
interaction with breakpoint width 81
interrupting 192
Run To 80
scope 19, 78
Tools > Watchpoint 113
width 19, 78
Window > Start Page 24
compiling
CUDA programs. See CUDA, compiling.
on ARM64 292
on x86-64 291
programs 47
using -g option 47
$complex data type 164
$complex_8 data type 164
complex types
in the VAR drawer 66
viewing in Data View 69
$complex_16 data type 164
conditional breakpoints 102, 284
configure new debugging session 25
continuous execution 192
Control Group attribute definition 83
control groups and share groups 80
control in parallel environments 204
control in serial environments 204
controlling program execution 204
core dump, naming the signal that caused 34
core files
debug session in Sessions Editor 33
debugging 45
multi-threaded 34
correcting programs 105
$countall built-in function 284
countdown breakpoints 102, 284
counter, loop 103
$countthread built-in function 284
Cray XT and temporary space 218
creating new processes 195
Ctrl+C 192
CUDA
@parameter qualifier 264
@register storage qualifier 266
assigned thread IDs 258
CLI and operating on CUDA threads 263
compiling a program for debugging 254
compiling options 254
compiling Pascal GPU 255
compiling Tesla GPU 255
compiling Volta GPU 255
coordinate spaces, 4D and 5D 259
CUDA thread defined 258
data from CUDA thread, displaying 261
execution, viewing 258
features 244
-g -G compiling option 254
GPU focus thread 258
GPU thread selector 258
host thread, viewing 259
installing 246
Linux-PowerLE, supported platform 244
Linux-x86_64, supported platform 244
logical coordinate space 259
MemoryChecker 267
nvcc compiler 254
physical coordinate space 259
process, defined 249
PTX register, locations 266
ReplayEngine limitations 274
requirements 244
runtime variables, supported 262
sample program 275
single-stepping GPU code 259
starting TotalView for CUDA applications 256
storage qualifier, supported types 261
troubleshooting 272
type casting 263
variables from CUDA thread, displaying 261
variables, editing 263
D
dactions command
-load 136
-save 133, 136
data types
see also data types
$address 164
$char 164
$character 164
$code 164
$complex 164
$complex_16 164
$double_precision 165
$extended 165
$float 165
$int 165
$integer 165
$integer_1 165
$integer_2 165
$integer_4 165
$integer_8 165
$logical 165
$logical_1 165
$logical_2 165
$logical_4 165
$logical_8 165
$long 165
$long_long 165
$real 165
$real_4 165
$real_8 165
$short 165
$string 165
$void 165
$wchar 166
$wchar_s16 166
$wchar_s32 166
$wchar_u16 166
wchar_u32 166
$wstring 166
$wstring_s16 166
$wstring_s32 166
$wstring_u16 166
$wstring_u32 166
predefined 164
Data View 12, 52
adding global variables 161
adding variables to capabilities 12
responding to Processes and Threads view actions 78
viewing updated values 69
viewing variables 66
data_format variables 173
dattach command 30, 36
-replay option for ReplayEngine 228
dbarrier command 116, 118, 119
dbfork library 32
dbreak command 211
disable command 120
deadlocks 79
Debug > Enable ReplayEngine menu item
disabling replay 224
enabling replay 224
debug info in separate file 293
Debug menu 224
Debug New Program dialog enabling ReplayEngine 223
Debug Options
Reverse Debugging 26, 28
debugger initialization 193
debugger PID 203
Debugger Unique ID (DUID) 283
debugging
arriving at goals 79
attributes in Processes and Threads view 74
basic tutorial 47
CLI commands supporting replay debugging 228
command descriptions 19
core file 45
Start Page 17
starting a session 17
switching between replay and live debugging 220
toolbar 19
debugging attributes
different views of 77
debugging behavior in Replay-Engine replay mode 218, 220, 225
debugging commands
at group width 80
at process width 80
at thread width 79
Group menu 80
interaction with breakpoint width 81
Process menu 80
Run To 80
scope 78
Thread menu 79
width 78
width selection menu in toolbar 79
debugging commands in ReplayEngine 218
debugging core files
in the Debug Core File dialog 33
debugging engine
unsupported features 2
debugging session 204
configure new 25
debugging toolbar 9
deleting
action points 128
derereferencing
automatic 174
pointers 174
dhistory command 229
dhold command 117
dialogs
drawers
preferences 4
disabling
action points 128
disabling ReplayEngine 224
Debug > Enable Replay-Engine menu item 224
with CLI dhistory -disable command 225
with Process > Startup Parameters dialog 225
with Record button 224
displaying
areas of memory 175
local variables 138
memory 175
diving
on action points 128
dkill command 195, 204
dload command 35, 194, 204
-replay option for ReplayEngine 228
returning process ID 196
dload -replay and -noreplay options 228
dnext and dnexti -back command-line options 228
docking and undocking views 5
documentation
Help view 14
$double_precision data type 165
dpid 203
dprint command 174, 208
drawers
Local Variables (VAR) 11
opening and closing 5
Processes and Threads attributes 10, 73
drerun command 194
drun command 194, 198
dssession command 35
dset command 198, 200
dstatus command 120
DUID 283
of process 283
$duid built-in variable 283
dunhold command 117
dunset command 198
dup commands 174
dwhere command 174
dynamic linker 298
dynamically linked program 298
E
effects of parallelism on debugger behavior 202
embed recordings
directly in a program using Replicator 221
enabling ReplayEngine with CLI dhistory command 224
with Debug > Enable Replay-Engine menu item 224
enabling watchpoints 110
environment variables
adding 38
how to enter 38
in Session Manager 26
eval breakpoints with ReplayEngine 220
Evaluate Window
documentation
limitations 106
evaluation system
examples
basic debugging 47
expression 47
exec() 32
EXECUTABLE_PATH variable 25, 27, 207
setting 206
executables
as definers of share groups 80
removing debug info 293
execution
controlling 204
execve() 96
setting breakpoints with 96
expression
basic debugging tutorial 47
expression system and arrays 140
array elements 140
C/C++ declarations 286
C/C++ statements 286
defined 140
Fortran 287
Fortran intrinsics 288
functions and their issues 281
structures 140
templates and limitations 286
Tools > Evaluate Window 105
$extended data type 165

F
F1 key for help 14
features of CUDA debugger. See CUDA, features.
Fermi GPU, compiling for. See CUDA, compiling for Fermi
File > Manage Sessions command 41
File menu 23
files searching for 12
$float data type 165
fork_loop.tvd example program 193
fork() 32, 96
setting breakpoints with 96
Fortran compiling on x86-64 291
expression system 287
in evaluation points 287
intrinsics in expression system 288
Function attribute definition 83
function calls, in eval points 104
functions in expression system 281
searching for 12

G
-g -G option, for compiling CUDA program. See CUDA, -g -G option.
-g option compiling for debugging 47
global variables adding to Data View 161
gnu_debuglink 293
gnu_debuglink command-line option 294
gnu_debuglink variable 294
gnu_debuglink_global_directory command-line option 294
gnu_debuglink_global_directory variable 294
Go command 19, 63
goal defined 79
GoBack toolbar button (ReplayEngine) 219
GOI defined 79
GPU. See CUDA.
Group > Go command 97, 121
group aliases 201
limitations 201
Group menu 19, 80
Group of Interest (GOI) 79
group width action points 126
of action points 123
group with the focus see GOI 79
groups adding an Attach to Program debug session 33
group width 80
in Processes and Threads view 73
GUI namespace 199

H
Halt command 19
held processes, defined 116
Help menu 14
Help view 14
F1 key 14
hexadecimal address, specifying in variable window 175
highlighting during ReplayEngine replay 225
$hold built-in function 284
hold state toggling 117
holding and advancing processes 204
holding problems 121
$holdprocess built-in function 284
$holdprocessall built-in function 285
$holdstopall built-in function 285
$holdthread built-in function 285
$holdthreadstop built-in function 285
$holdthreadstopall built-in function 285
$holdthreadstopprocess built-in function 285
Hostname attribute definition 83
hung processes debugging session 29

I
I state indicator 31
idle state indicator 31
IDs action points 86
image file, stripped copy 293
Info view 66
initial process 202
initializing an array slice 207
initializing the CLI 193
$int data type 165
$integer_2 data type 165
$integer_4 data type 165
$integer_8 data type 165
interactive CLI 190
interface, new docking and undocking 5
drawers 5
Index

layout 4
main views 4
resizing behavior 4
Source views 6
Start Page 6
internal counter 103
interrupting commands 192
intrinsics
$newval 111
$oldval 111
invoking CLI program from shell example 193

K
Kill command 19

L
launching NextGen UI 24
LD_BIND_NOW environment variable 298
libraries
shared 297
library and system call behavior during ReplayEngine replay 218
limitations
CUDA and ReplayEngine 274
limitations in evaluation system 106
LINES_PER_SCREEN variable 197
Linux-PowerLE 115
Linux-PowerLE, supported CUDA platform. See CUDA, Linux-PowerLE.
Linux-x86_64, supported CUDA platform. See CUDA, Linux-x86_64.
Live toolbar button
(ReplayEngine) 219, 220
-load_session flag 46
loading
action points 136
file into debugger 45
loading sessions
Session Editor 34
local variables
in the VAR drawer 66

Local Variables (VAR) drawer 11
Local Variables view
adding variables to Data View 12
lockstep group
defined 79
Logger 13, 52
$logical data type 165
$logical_1 data type 165
$logical_2 data type 165
$logical_4 data type 165
$logical_8 data type 165
$long data type 165
$long_long data type 165
Lookup File or Function view 55
Lookup view 12
loop counter 103

M
make_actions.tcl sample macro 210
Manage Debugging Sessions window
accessing 41
managing sessions from File menu 41
map templates 167
matching processes 81
memory, displaying areas of 175
MemoryScape debugging not supported 3
menus
Debug 224
documenting command width 79
Group 19
Group menu 80
Help 14
Local Variables context menu 12
Process 19, 220, 225
Process menu 80
save ReplayEngine recording file 221
show/hide views 237
Thread 19
Thread menu 79
messages from debugger, saving 196
more processing 196
more prompt 196
multi-threaded core files 34

N
namespaces 199
TV:: 199
TV::GUI:: 199
navigating
to functions 53
navigation
diving and searching 52
using disabled action points 130
Next command 19, 60
NextGen UI
newUI, command line option 3
overview 2
relationship to CLI 191
starting 18
unsupported features of TotalView debugging engine 2

$nid built-in variable 283
no_dynamic command-line option 297
node ID 283
non-sequential program execution 192
nvcc compiler, and CUDA. See CUDA, nvcc compiler.
NVIDIA. See CUDA.

O
$oldval built-in variable 283
opening a core file 36
options, for compiling CUDA. See CUDA, compiling options
Out command 19, 60
output
assigning output to variable 196
from CLI 196
only last command executed returned 196
printing 196
returning 196
when not displayed 196

P
parallel environments, execution control of 204
parallel program, defined 202
parsing comments example 210
passing arguments 46
passing default arguments 198
patching
  function calls 104
  programs 103
PATH environment variable
  entering for debugging session 25
PC attribute definition 83
pending breakpoint, creating 90
performance with
  ReplayEngine 218
$pid built-in variable 283
POI
  defined 79
  pointers
    as arrays 174
    chasing 174
dereferencing 174
predefined data types 164
preferences 4
preferences toolbar 9
Prev toolbar button
  (ReplayEngine) 219
primary thread, stepping failure 79
printing an array slice 208
Procedure Linkage Table (PLT) 298
process
  ID 283
  numbers are unique 202
Process > Go command 121
process barrier breakpoint
  changes when clearing 120
  changes when setting 120
  setting 117
process creation by
  ReplayEngine 219
process DUID 283
Process Held attribute
  definition 83
Process ID attribute
  definition 83
Process menu 19, 80, 220
  ReplayEngine commands 225
Process of Interest (POI) 79
Process State attribute
  definition 83
process width
  action points 124
  of action points 123
process with the focus
  see POI 79
process/thread identifier 202
process/thread notation 202
process/thread sets 202
$processduid built-in variable 283
processes
  attaching to 29
  attribute definitions 83
  barrier point behavior 120
  creating new 195
  held defined 116
  holding 116, 284
  hung, debugging 29
  in ptlist format 75
  in share groups 80
  initial 202
  process width 80
  released 116
  releasing 116, 120
  spawned 202
  state and action points 124
  stopped 116
  stopped at barrier point 120
  stopping intrinsic 285
  synchronizing 205
  synchronizing with Run
    To 80
  terminating 195
  viewing and controlling 73
  when stopped 80
Processes and Threads view 10, 51
  ascending/descending column order 76
attribute definitions 83
attributes 10, 74
different views of the attributes 77
effect on other views 78
main description 73
tabular presentation 75
tree presentation 74
Program arguments
  in Debug New Program dialog 25
program execution
  advancing 204
  controlling 204
Program Session dialog 25, 27
program state in
  ReplayEngine 218
program state, changing 192
programs
  compiling 47
  correcting 105
  patching 103
PROMPT variable 200
pthread ID 203
ptlist format 75

R
R state indicator 31
$real data type 165
$real_16 data type 165
$real_4 data type 165
$real_8 data type 165
Record button (ReplayEngine)
  disabling replay 224
record mode 218
Record toolbar button
  (ReplayEngine) 219
recording file
  configuring with
    ReplayEngine 33
recording program history 218
relatives
  attaching to 32
Release-specific
  The CLI supports multiple types of action points,
  some of which are not
Index 311

currently supported in the UI 132
remote debugging not supported 3
replacing default arguments 198
replay mode 218
Replay Mode attribute definition 83
Replay recording file and Replicator 221
replay recording file debugging session 33
ReplayEngine breakpoints, evalpoints, and barrierpoints 220
canceling operations in replay mode 218
CLI command support 228
CLI commands for debugging during replay 228
configuring in Session Manager 33
CUDA limitations 274
displaying commands 218
displaying commands during replay 220, 225
debugging through the CLI 228
dhistory CLI command 229
disabling 224
enabling 223
enabling in Session Manager 37
end of replay marker 226
highlighting in code during replay 225
library and system call behavior during replay 218
performance when using 218
process creation 219
program state 218
record mode 218
replay mode 218
saving a recording file 221
scoping disabled during replay 226
supports only process width 226
switching between replay and live debugging 220
system and library call behavior during replay 218
thread execution in replay mode 220
ReplayEngine commands 220
ReplayEngine toolbar 9
Replicator standalone product 221
resetting command-line arguments 38
resizing interface views 4
Restart command 19
restarting program execution 194
results, assigning output to variables 196
Reverse debugging debug options 26, 28
reverse debugging Session Manager option 37
Run To command 19, 59
used to synchronize 80
running state indicator 31
S -s command-line option 193
S state indicator 31
sane command argument 193
satisfaction set 119
satisfied barrier 119
saving action points 136
messages 196
-sb option 136
scalar values in the VAR drawer 66
scope of debugging commands 19
scoping disabled during replay (ReplayEngine) 226
scoping output 196
search for an existing session 42
Lookup view 12
searching for files and functions 12
Session Editor recent sessions 34
Session Manager editing previous sessions 35
reverse debugging 37
sessions configure new debugging 25
deleting duplicates 43
editing 35
loading an existing 43
loading using -load_session flag 46
loading using the CLI 35
managing 42
Start Page 17
starting 17, 50
setting breakpoints 87, 210
breakpoints while running 87
thread specific breakpoints 283
Share Group attribute definition 83
share groups as members of control groups 80
defined 80
SHARE_ACTION_POINT variable 96
shared libraries 297
shell, example of invoking CLI program 193
$short data type 165
showing areas of memory 175
side 281
side-effects of functions in expression system 281
signals that caused core dump 34
sleeping state indicator 31
sorting
action points 127
Source Line attribute definition 83
Source views 6, 51
responding to Processes and Threads view actions 78
source-level breakpoints 87
space requirements with ReplayEngine 218
spawned processes 202
stack frame 174
standard I/O
altering in Session Manager 38
redirecting 50
standard template library, see STL
Start Page 6, 17, 23, 24
starting CLI 193
starting a debugging session 17
starting NextGen UI 18
state
of action points 130
process/thread state and action points 124
state of program in ReplayEngine 218
static internal counter 103
status bar, displaying state 59
Step command 19, 58
stepping
at thread width 79
lockstep groups 79
primary thread can fail 79
target program 204
tutorial 58
STL
map transformation 167
$stop built-in function 285
stop function 212
Stop Group
action point width setting 123
Stop Process
action point width setting 123
Stop Reason attribute definition 83
Stop Thread
action point width setting 123
stop, defined in a multiprocess environment 204
$stopall built-in function 285
stopped process 120
stopped state indicator 31
$stopprocess built-in function 285
$stopthread built-in function 285
storage qualifier for CUDA. See CUDA, storage qualifier
$string data type 165
stripped copy 293
structures
expression evaluation 140
stty sane command 193
suppressing action points 130
synchronizing processes 205
system and library call behavior during ReplayEngine replay 218
system PID 203
system TID 203
$systid built-in variable 283
T
T state indicator 31
tabular presentation of process and thread attributes 75
target process/thread set 204
target program stepping 204
Tcl
CLI and thread lists 191
version based upon 191
temp space requirements with ReplayEngine 218
temp space with Cray XT 218
templated code
and breakpoints 88
templates
expression system 286
vectors 167
terminal window
starting CLI from 193
terminating processes 195
Tesla GPU, compiling for. See CUDA, Tesla GPU.
thread execution during ReplayEngine replay 220
thread groups 79
Thread Held attribute definition 83
Thread ID
attribute definition 83
thread ID about 203
assigned to CUDA threads.
See CUDA, assigned thread IDs.
system 283
TotalView 283
Thread Index attribute definition 83
Thread menu 19, 79
thread numbers are unique 202
Thread of Interest (TOI) 79
Thread State attribute definition 83
thread width
of action points 123
thread with the focus see TOI 79
threads
arriving at a goal 79
attribute definitions 83
holding 118
in ptlist format 75
in share groups 80
releasing 116
setting breakpoints in 283
state and action points 124
synchronizing with Run To 80
thread width 79
viewing and controlling 73
threads in lockstep 79
thread-specific breakpoints 283
$tid built-in variable 283
TMPDIR environment variable and ReplayEngine 218
TOI defined 79
toolbars
described 9
ReplayEngine 225
showing 4
turning on text 4, 9
width menu in debugging toolbar 79
Tools > Evaluate Window expression system 105
Tools > Watchpoint command 113
tooltips 66
viewing variable values 144
TotalView debugging engine unsupported features 2
totalviewcli command 193, 194
transformations of STL types 167
tree presentation of process and thread attributes 74
troubleshooting 302
TTF and STL types 167
tff variable 167, 173
tff_max_length variable 173
TV:: namespace 199
TV::GUI:: namespace 199
TVD_REPLAY_TMPDIR variable 218
type strings
built-in 164
type transformation variable 167, 173
type transformations, creating 167
U
unique process numbers 202
unique thread numbers 202
Unstep toolbar button (ReplayEngine) 219
unsuppressing action points 130
using ReplayEngine 223

V
values
updated in DataView 69
VAR drawer 11, 63
viewing variables at breakpoint 65
VAR panel
displaying. See also VAR drawer. 138
variables
adding to DataView 67
global, viewing 161
setting command output to 196
tff 167, 173
values update in DataView 67
viewing value at breakpoint 65
vector transformation, STL 167
VERBOSE variable 192
views
Action Points 13
Call Stack 11
CLI 13
Command Line 13
Data 12
docking and undocking 5
Help 14
initial layout in interface 4
Local Variables drawer 11
Logger 13
Lookup 12
Processes and Threads 10, 73
showing and hiding 237
$visualize 285
$void data type 165
Volta, compiling for 255

W
watching memory 111
Watchpoint command 113
watchpoints
$newval watchpoint variable 113
$oldval 113
alignment 114
conditional 113
copying data 113
defined 205
enabling 110
example of triggering when value goes negative 113
length compared to $oldval or $newval 114
lowest address triggered 112
modifying a memory location 107
monitoring adjacent locations 112
multiple 112
on stack variables 109
PC position 112
platform differences 115
problem with stack variables 111
supported platforms 114
triggering 107, 112
unconditional 108
watching memory 111
When Done, Stop radio buttons 118
When hit action point width property 123
When Hit, Stop radio buttons 118
width
action points, default 124
action points, explained 123
action points, group width 126
action points, process width 124
action points, thread width 125
behavior of group width 80
behavior of process width 80
behavior of thread width 79
command and breakpoint interaction 81
in debugging commands 19, 78
menu in debugging toolbar 79
writing array data to files 209

Z
Z state indicator 31
Zombie state indicator 31