

# TotalView User's Guide

---

TotalView Version 4.0  
DOC-0516-00

Product names mentioned in *TotalView User's Guide* are trademarks of their respective manufacturers and are used here only for identification purposes.

TotalView, TimeScan and Gist are trademarks of Dolphin Interconnect Solutions, Inc.

FLEXlm is a registered trademark of Globetrotter Software, Inc.

Copyright © 1996 - 1998 by Dolphin Interconnect Solutions, Inc. All rights reserved.

Copyright © 1996 by BBN Systems and Technologies, a division of BBN Corporation.

Copyright © 1987 - 2002, LynuxWorks, Inc. All rights reserved.

Printed in the United States of America.

All rights reserved. No part of *TotalView User's Guide* may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photographic, magnetic, or otherwise, without the prior written permission of LynuxWorks, Inc.

LynuxWorks, Inc. makes no representations, express or implied, with respect to this documentation or the software it describes, including (with no limitation) any implied warranties of utility or fitness for any particular purpose; all such warranties are expressly disclaimed. Neither LynuxWorks, Inc., nor its distributors, nor its dealers shall be liable for any indirect, incidental, or consequential damages under any circumstances.

(The exclusion of implied warranties may not apply in all cases under some statutes, and thus the above exclusion may not apply. This warranty provides the purchaser with specific legal rights. There may be other purchaser rights which vary from state to state within the United States of America.)

# Contents

## CHAPTER 1:

### **Introduction 11**

- TotalView's Advantages 12
- TotalView's Windows 14
- Examining Source and Machine Code 16
- Controlling Processes and Threads 16
- Using Action Points 17
- Examining and Manipulating Data 18
- Distributed Debugging 19
- Multiprocess Programs 20
- Multithreaded Programs 22
- Context-Sensitive Help 23

## CHAPTER 2:

### **TotalView Basics 25**

- Starting TotalView 26
- Using the Primary Windows 26
- Using the Mouse Buttons and Menus 33
- Scrolling Windows and Fields 34
- Getting Help 37
- Diving into Objects 37
- Editing Text 38
- Searching for Text 41
- Using the Spelling Corrector 42
- Saving the Contents of Windows 42

Exiting from the TotalView Debugger 43

## **CHAPTER 3:**

### **Setting Up a Debugging Session 45**

Loading Executables 48

Attaching to Processes 50

Debugging Remote Processes 53

Detaching from Processes 55

Examining a Core File 56

Starting the Debugger Server for Remote Debugging 57

Determining the Status of Processes and Threads 68

Setting Search Paths 75

Setting Command Arguments 77

Specifying Environment Variables 78

Setting Input and Output Files 80

Monitoring TotalView Sessions 80

## **CHAPTER 4:**

### **Debugging Programs 83**

Finding the Source Code for Functions 86

Editing Source Text 87

Changing the Editor Launch String 88

Interpreting Status and Control Registers 89

Controlling Program Execution 90

Starting Processes and Threads 91

Examining Process Groups 92

Setting a Breakpoint 97

Continuing with a Specific Signal 102

Setting the Program Counter 103

Stopping Processes and Threads 105

Deleting Processes 106

Restarting Processes 106

## **CHAPTER 5:**

### **Examining and Changing Data 107**

Changing the Values of Variables 115

Changing the Data Type of Variables 116  
Displaying Array Slices 125  
Changing Type Strings to Display Machine Instructions 130

**CHAPTER 6:**

**Setting Action Points 133**

Defining Evaluation Points and Conditional Breakpoints 140  
Controlling Action Points 145  
Saving Action Points in a File 150  
Evaluating Expressions 150  
Writing Code Fragments 152

**CHAPTER 7:**

**Troubleshooting 159**

**CHAPTER 8:**

**X Resources 163**

**CHAPTER 9:**

**TotalView Command Syntax 175**

**CHAPTER 10:**

**TotalView Debugger Server Command Syntax 181**



# About This Guide

This guide describes how to use the TotalView Multiprocess Debugger, a source-level and machine-level debugger with an easy-to-use interface (based on the X Window System) and support for debugging multiprocess and multithreaded programs. The guide assumes that you are familiar with the C programming language, UNIX operating systems, the X Window System, and the processor architecture of the platform on which you're running TotalView.

This guide covers the general use of TotalView on any platform. Most of the examples and illustrations in this guide show TotalView running on in a generic UNIX environment. To learn about the specifics of running TotalView on your LynxOS platform, refer to the *TotalView Supplement for LynxOS Users*.

---

## Getting Started

To get started quickly with TotalView:

- Install the TotalView software as described in “Installing TotalView” on page 1.
- Read “Introduction” on page 11 for an overview of TotalView.
- Learn the basics by reading “TotalView Basics” on page 25 and “Compiling Programs” on page 46.

---

# Typographical Conventions

This guide uses the following conventions to present information:

<b>bold</b>	An exact filename, command, or user input.
<i>italic</i>	A variable or a value that you supply.
<code>typewriter</code>	Computer output; C and Fortran programs.
Control-Z	Press the keys simultaneously; for example, hold down the Control key and press the Z key.
Esc Z	Press the first key and then the second; for example, press the Escape key and then press the Z key.
^Z	Shorthand for Control-Z.
M-I	Shorthand for Meta-I. (The Meta key varies with your platform; usually it is the Alt key.)
menu ? submenu	Shorthand for a popup menu name and submenu name.
[ ]	Optional items in command syntax descriptions.
...	Repetition of the previous command or input.



---

## Reporting Problems

If you experience any problems with TotalView, please contact LynuxWorks Technical Support Monday–Friday (holidays excluded) between 8:00 AM and 5:00 PM Pacific Time (LynuxWorks U.S. Headquarters) or between 9:00 AM and 6:00 PM Central European Time (LynuxWorks Europe).

*LynuxWorks U.S. Headquarters*

Internet: [support@lnxw.com](mailto:support@lnxw.com)

Phone: 408-879-3940

Fax: 408-879-3945

*LynuxWorks Europe*

Internet: [tech\\_europe@lnxw.com](mailto:tech_europe@lnxw.com)

Phone: (+33) 1 30 85 06 00

Fax: (+33) 1 30 85 06 06

*World Wide Web*

<http://www.lynuxworks.com>



## ***CHAPTER 1:***

# **Introduction**

The TotalView debugger is part of a suite of software development tools for debugging, analyzing, and tuning the performance of programs, including multiprocess multithreaded programs. This chapter highlights the features of TotalView and includes the following sections:

- TotalView's advantages
- TotalView's windows
- Examining source and machine code
- Controlling processes and threads
- Using action points
- Examining and manipulating data
- Distributed debugging
- Multiprocess programs
- Multithreaded programs
- Context-sensitive help

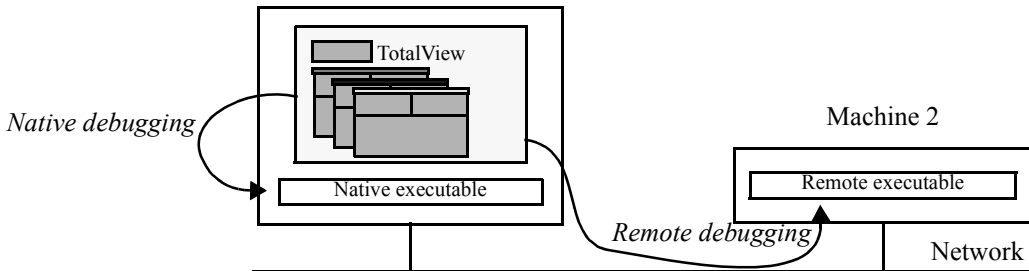
---

# TotalView's Advantages

TotalView provides many advantages over conventional UNIX debuggers (such as **dbx**, **gdb**, and **adb**):

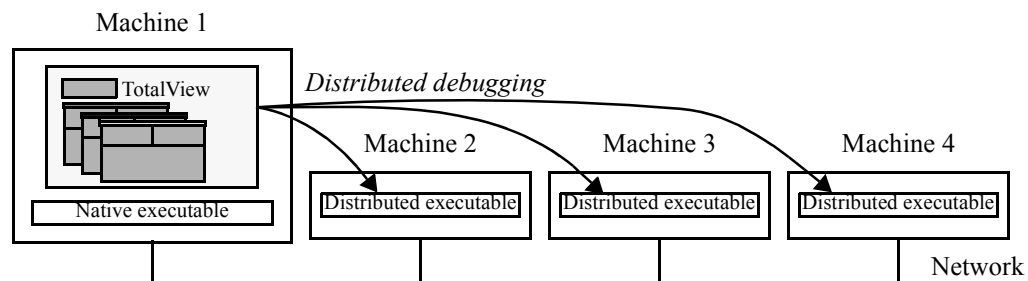
- You can learn TotalView quickly and be more productive because of its graphical interface (based on the X Window System). TotalView's interface provides windows, pop-up menus, and a context-sensitive help system. You can enter most commands with the mouse. Further, with TotalView's interface, you can already see a lot of useful information without entering any commands.
- You can debug *multiprocess multithreaded programs* because TotalView can manage multiple processes, and multiple threads within a process. TotalView displays each process in its own window, showing the source code, stack trace, and stack frame for one or more threads in the process. You can display all process windows simultaneously and perform all debugging tasks across processes.

**Figure 1.** You can debug *remote programs* over the network because of TotalView's distributed architecture, as shown in Figure 1. Remote programs are programs that run on a different machine from TotalView, while native programs are programs that run on the same machine as TotalView.



**Figure 1.** Debugging a Remote Program

- You can debug *distributed programs* over the network because TotalView can manage multiple remote programs and multiprocess multithreaded programs simultaneously, as shown in Figure 2. Distributed programs are programs that run on a group of different machines.



**Figure 2.** Debugging a Distributed Program

- You can also *cross debug* programs using TotalView. Cross debugging is similar to remote debugging and distributed debugging, except that with cross debugging the host machine and target machines are *not* of the same type. For example, a cross debugging session might consist of a Solaris host machine and several Intel x86 target machines.
- You can write source code fragments within TotalView and insert them temporarily into the program you're debugging. On some platforms, you can write machine code fragments as well. This feature can save you time in testing bug fixes.
- You can debug code that was not compiled with the `-g` switch or for which you don't have access to the source file because TotalView provides machine-level debugging features.
- You can attach to running processes.

# TotalView's Windows

TotalView displays extensive information in its windows, as shown in Figure 3. Several commands may be required to display this with other debuggers.

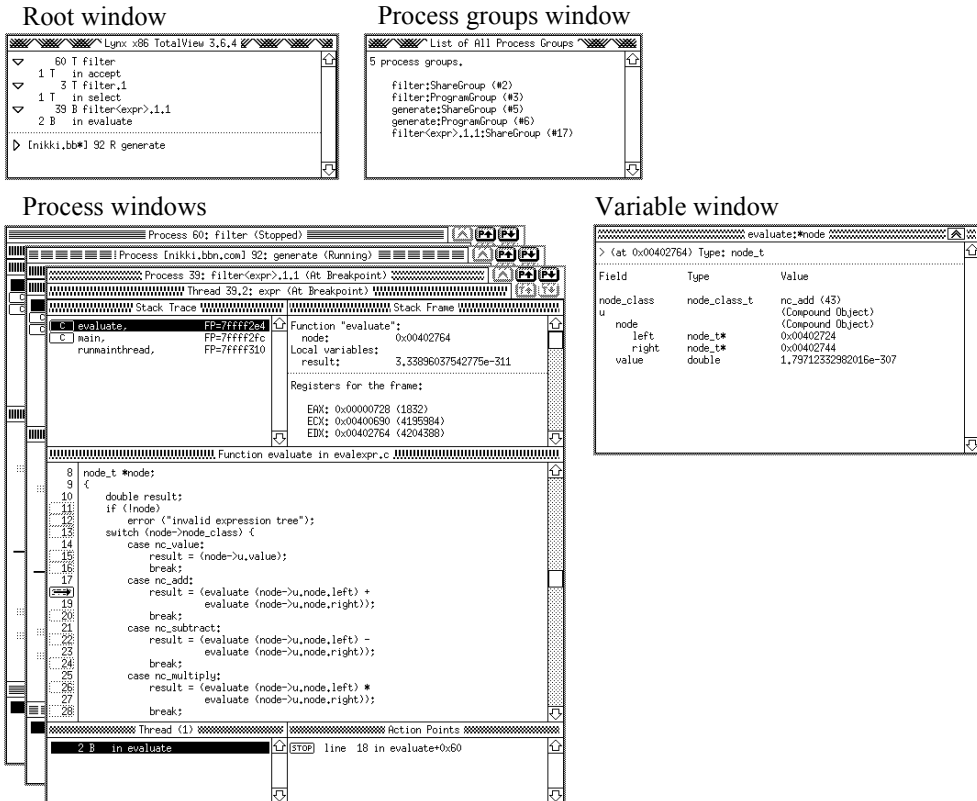


Figure 3. Sample TotalView Session

Figure 3 shows a sample TotalView session containing the following windows:

<b>Root</b>	Lists the name, location (if remote process), process ID, status, and optionally the list of threads for each process you are debugging. Lists the thread ID, status, and current routine executing for each thread.
<b>Process</b>	Displays information about the process and a thread within that process. Displays the stack trace, stack frame, and source code for the selected thread in a series of separate panes. Optionally displays disassembled machine code or interleaved source code and disassembled machine code.
<b>Process groups</b>	Displays the process groups for all of the multiprocess programs you are debugging.
<b>Variable</b>	Displays the address, data type, and value of a local variable, register, or global variable. Also displays the values (and optionally, the machine-level instructions) stored in a block of memory.

The process window provides very detailed information about a process, including:

- The name, location (if remote process), process ID, and status of the process
- The name, location (if remote thread), thread ID, and status of the selected thread within the process
- The stack trace for the thread, with the selected routine highlighted
- The stack frame for the selected routine
- The source code for the selected frame (providing the routine was compiled with source line information) or disassembled machine code

- The current Program Counter (PC) for the selected stack frame, which is represented by an arrow on the line number of source code
- The breakpoints, evaluation points, and event points that are set in the source or machine code, as shown in the source pane
- The list of threads that exist within the process
- The list of breakpoints, evaluation points, and event points that are set in the process

---

## Examining Source and Machine Code

TotalView provides the following features for examining your code:

- “Dive” on functions

When you dive (press the right mouse button) on a function, its source code is displayed in the source code pane of the process window.

- Search for functions

You can search for functions using a dialog in the process window.

---

## Controlling Processes and Threads

For controlling processes and threads, the TotalView debugger offers a full range of functions from the process window.

- Start and stop processes and threads

You can start, stop, resume, delete, and restart.



- Attach to existing processes

TotalView provides a window for examining processes that are not running under the debugger's control. Attaching to one of these processes is as easy as diving on it.

- Examine core files

When you start TotalView, you can load a core file and examine it in the same way as any executable. Or, you can load a core file any time during a TotalView debugging session.

- Change the way TotalView handles signals

TotalView provides a dialog for tailoring how signals are handled. TotalView can stop the process and place it in the stopped state, stop the process and place it in the error state, send the signal on to the process, or discard the signal.

- Single step

You can single step through your program or step over function calls. You can also continue execution to a selected source line or instruction and continue execution until a function completes execution.

- Reload the executable file

After editing and recompiling a program, you can reload the executable file.

- Change the Program Counter (PC)

You can change the value of the PC to resume execution at a different point in the program.

---

## Using Action Points

TotalView provides a broad range of *action points*: points in a program where you stop execution, evaluate an expression, or record an event.

- Action points

You can set, delete, enable, disable, suppress, and unsuppress the following kinds of action points at both the source level and machine level.

- Breakpoints

- Conditional breakpoints, which are breakpoints that occur only if a code fragment (expression) is satisfied
- Evaluation points, which are points where a code fragment is evaluated
- Event points, which are points where an event is recorded in an event log

- Expressions and code fragments

With the expression evaluation window and evaluation points, you can write and evaluate fragments of code, including function calls used by the current process. Depending on the platform, you can write fragments in C, Fortran, or Assembler. On most platforms, TotalView interprets code fragments, but on some platforms, TotalView compiles the fragments.

---

## Examining and Manipulating Data

The TotalView debugger also offers a number of useful functions for examining and manipulating data in your program:

- Diving

You can examine data by “diving” (clicking the right mouse button) into the variable or by issuing a command. You can examine local variables, registers, global variables, machine-level instructions, and areas of memory. In all cases, the debugger displays the information about the variable, register, or memory region in a separate variable window.

- Changing values

You can edit the value of a variable or a memory location to change it for the current running process.

- Changing types

You can edit the type strings of variables to display the data in different formats.

---

## Distributed Debugging

TotalView provides a distributed architecture that suits many different operating environments, including:

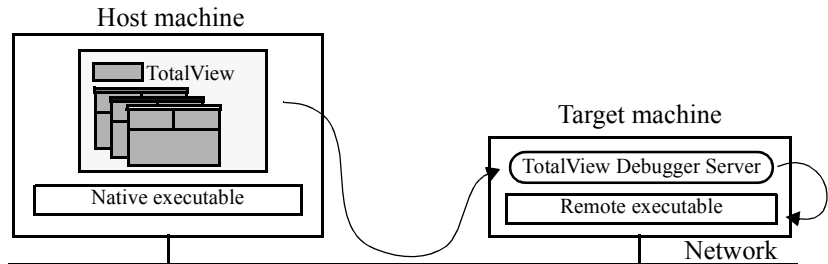
- Remote programs running on a separate machine from TotalView
- Distributed programs running on a set of machines
- Multiprocess programs running on a multiprocessor machine
- Multiprocess programs running on a cluster of separate homogeneous machines
- Client-server programs with the server running on one machine type and the clients running on another machine type

---

**Note:** Distributed debugging currently requires that all machines have the same machine architecture and operating system.

---

The machine on which TotalView is running is known as the *host machine*, while the machine on which the process being debugged is running is known as the *target machine*. When the host and target machines are the same, you can use TotalView as a native debugger. When the host and target machines are different, you can use TotalView as a distributed debugger. When you use TotalView as a distributed debugger, it starts a process on each remote target machine. This process is called the TotalView Debugger Server (**tvdsvr**), and TotalView communicates with it using standard TCP/IP protocols (see Figure 4).



**Figure 4.** The TotalView Debugger Server

There are no differences in debugging distributed programs; TotalView offers the same set of rich features as with native programs and multiprocess programs.

For more information on distributed debugging, refer to:

- “Debug remote processes” on page 45
- “Start the debugger server for remote debugging” on page 45
- “TotalView Debugger Server Command Syntax” on page 181

---

## Multiprocess Programs

The TotalView debugger has some special features for debugging multiprocess programs. Note that all of the user interface and debugging features that were discussed earlier in this chapter are also available for multiprocess programs.

- Separate windows for each process

Each process has its own process window displaying information for that particular process. You can monitor the status, thread list, breakpoint list and source code, for *each* process in a multiprocess

program. You don't have to display all the process windows in a multiprocess program; you can choose which process windows to open and close.

- Sharing of breakpoints among processes

By setting options on the breakpoint in a parent process, you can control whether or not the breakpoint is shared among the child processes. You can also control whether or not all processes in the group stop when any process in the group reaches the breakpoint.

- Process groups

The TotalView debugger treats multiprocess programs as process groups. If you debug several multiprocess programs at once, you can view information about all process groups. You also can view information about a particular multiprocess program by requesting information about its process group. You can start and stop an individual process group.

- Single event log containing information for all processes

The TotalView debugger logs significant events about each process you are debugging. Thus, you can view the history of your entire debugging session by scrolling through the event log window.

- Automatically attach to child processes

If a program calls **fork()** or **execve()**, TotalView automatically attaches to the child processes and includes them in the process group.

- Multiple symbol tables

If you are debugging more than one executable at a time, TotalView automatically handles the symbol table for each executable.

---

## Multithreaded Programs

Most modern operating systems support running programs with multiple threads of execution. The implementation of threads varies among operating systems, but most thread implementations share the following characteristics:

- Shared address space  
The threads share an address space (memory) with other threads. They can read and write the same variables and can execute the same code.
- Private execution context  
Each thread has its own set of general-purpose registers and floating-point registers (if applicable to the processor).
- Private execution stack  
Each thread has a region of address space reserved for its execution stack. This is typically a range of addresses in the address space reserved for the thread's stack. However, one thread's stack can be read and written by other threads sharing the address space.
- Thread private data  
Some operating systems (not all) allow a program to “declare” thread private data. A program variable that is declared thread private provides each thread its own copy of the variable. Changes to the variable by one thread are not seen by the others. This facility usually requires compiler and linker support, in addition to operating system support.

TotalView supports debugging threaded applications on a variety of operating systems. In most versions of UNIX-style operating systems that support threads, a process consists of an address space and a list of one or more threads. Other non-UNIX-style operating systems that TotalView supports implement tasks or threads running in the memory space of a computer, and have no facilities for multiple processes or address spaces on a single machine.

To handle this diversity, TotalView implements a general model of address spaces and execution contexts. For conciseness, we use the term **thread** to mean a thread or task with an execution context, and **process** to mean an address space or computer memory that is capable of running one or more threads.

---

## Context-Sensitive Help

You can request help from every window in the TotalView debugger. The help command displays context-sensitive information about the window you are currently working in or the debugging operation you are currently using. The debugger displays the information in a separate help window, so you can scroll through the information as you debug your program. As you make successive help requests, the debugger displays the new information in the help window.





## **CHAPTER 2:**

# **TotalView Basics**

This chapter introduces you to the TotalView interface. You'll learn how to:

- Compile your program
- Start TotalView
- Use the primary windows
- Use the mouse buttons and menus
- Scroll windows and fields
- Get online help
- Dive into objects
- Edit text
- Search for text strings
- Use the spelling corrector
- Save the contents of windows
- Exit TotalView

---

## Compiling Programs

Before you start TotalView, you need to compile your program with the `-g` compiler switch, which generates debugging information in the symbol table. For example:

```
% gcc -g program -o executable
```

For more information on compiling your program for TotalView, see “Compiling Programs” on page 46.

If necessary, you can debug programs that have not been compiled with the `-g` compiler switch or programs for which you do not have the source code. For more information, refer to “Examining Source and Assembler Code” on page 84.

---

## Starting TotalView

To start TotalView, use the `totalview` command with the name of your program (*filename*):

```
% totalview filename
```

For more information on starting TotalView, see “Starting the TotalView Debugger” on page 47.

---

## Using the Primary Windows

When you start the TotalView debugger, two windows appear:

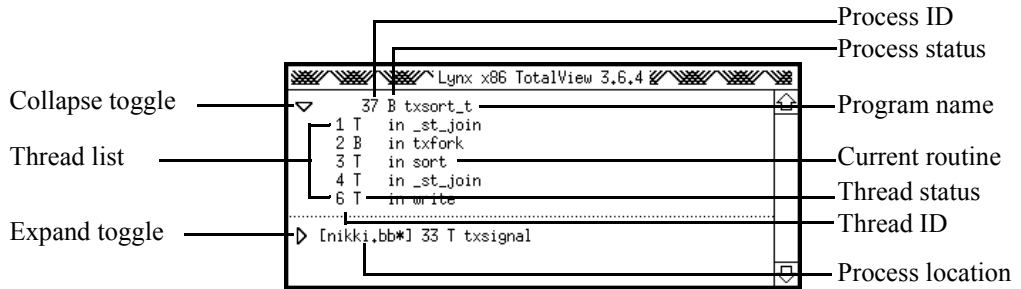
- The root window displays a list of all the processes that you are debugging, and optionally a list of threads for each process (see Figure 5). Until you start a process, the root window lists only the name of the program with which you started TotalView.

- The process window displays the thread list, action point list and the selected thread of a particular process that you are debugging (see Figure 6). The process window also displays the source code, stack frame, and stack trace of the selected thread in that process. Until you start the process, the process window displays only the source code for the program.

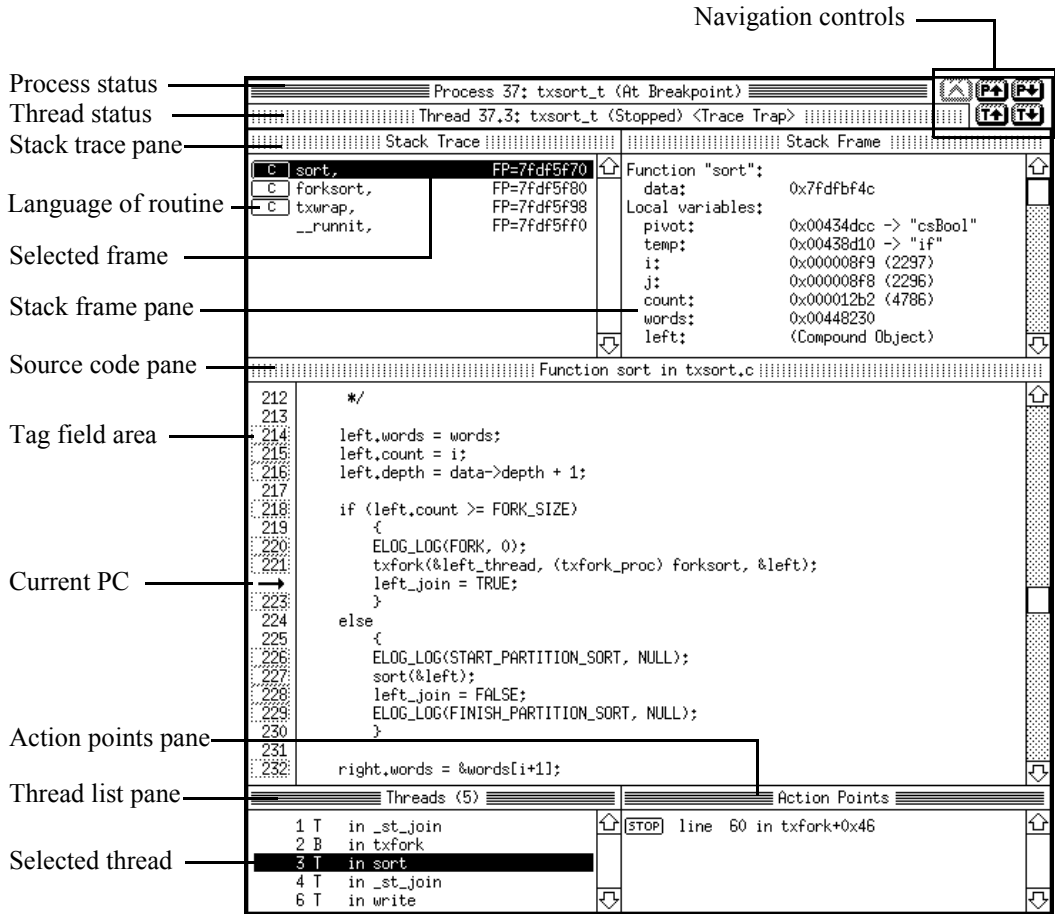
To run your program:

1. Move your cursor to the process window.
2. In your source code, select a boxed line number with the left mouse button to set a breakpoint on that line.
3. Type **g**.

The process starts running and then stops at the breakpoint you set.



**Figure 5.** Root Window



**Figure 6.** Process Window

When you are debugging a remote process, the abbreviated hostname on which the process is running appears in square brackets in the root window, and the full hostname appears in square brackets in the title bar of the process window. For example, in Figure 5, the process running **txsignal** is on the machine **nikki.bbn.com**, which is abbreviated to **[nikki.bb\*]** in the root window. In the process window, the full hostname of the process **[nikki.bbn.com]** is displayed.

As you examine the process window, notice the following:

- The thread list pane shows the list of threads that currently exist in the process. The number in the thread list pane title is the count of the number of threads that currently exist in the process. When you select a different thread in the thread list, TotalView updates the stack trace pane, stack frame pane, and source code pane to show you the information for that thread. When you dive on a different thread in the thread list, TotalView finds or opens a new window displaying information for that thread. Holding down the Shift key when you dive will force TotalView to open a new process window focused on that thread.
- The stack trace pane shows the call stack of routines that are executed by the selected thread. You can move up and down the call stack by selecting the desired routine (stack frame). When you select a different stack frame in the call stack, TotalView updates the stack frame pane and source code pane to show the information about the selected routine.
- The stack frame pane displays all the function parameters, local variables, and registers for the selected stack frame.
- The information displayed in the stack trace and stack frame panes reflects the state of the process when it was last stopped. Therefore, the stack trace and stack frame panes are *not* current while the thread is running.
- In the left margin of the source code pane, the tag field area contains line numbers opposite all lines of source code. You can place a breakpoint at any line of source code that generated object code, which is indicated by a boxed line number. The arrow (→) in the tag field indicates the current location of the Program Counter (PC).
- In a multiprocess or multithreaded program, each thread has its own point of execution, so the right arrow (→) points to a unique PC in each process window for a particular thread. Therefore, when you stop a multiprocess or multithreaded program, the routine selected in the stack trace pane for a particular thread depends on the PC for the thread. At the time you stop the program, some threads might be executing in one routine, while others might be executing in other routines.

- The action point list pane shows the list of breakpoints, eval points, and event points for the process.
- The navigation control buttons in the upper right-hand corner of the process window allow you to easily navigate through the processes and threads you are debugging.

You can resize the panes in the process window. If you do not want to see a particular pane, you can resize the pane to a zero size. To do so:

1. Move the mouse cursor over the edge of the window pane until the cursor with crossed arrows appears, as shown in Figure 7:



**Figure 7.** The Sizing Cursor

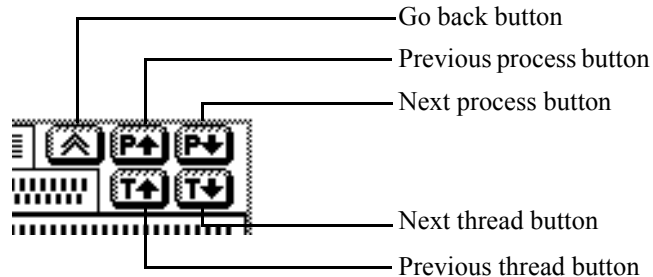
2. Hold down the left mouse button and drag the edge until the pane is the desired size.

## **Navigating in the Process Window**

The navigation control buttons in the upper right-hand corner of the process window allow you to easily navigate through the processes and threads you are debugging. Using these buttons you can:

- Move up and down the list of processes you are debugging.
- Move up and down the list of threads in a particular process.
- Go back to the previous contents of the process window.

Figure 8 shows the navigation controls available in the process window.



**Figure 8.** Process Window Navigation Controls

## Navigating in the Root Window

You can also navigate through the processes and threads you are debugging from the root window. In general, selecting a process or thread with the left mouse button will not open a new window. Selecting tries to minimize the number of open process windows. However, diving (pressing the right mouse button) on a process or thread will open a new process window if an exactly matching process/thread combination could not be found. Finally, holding down the Shift key when you dive will always open a new window.

- When you select a process in the root window, TotalView will find or open a process window for that process. If a matching window can't be found, it will replace the contents of an existing process window and show you the selected process.
- When you dive on a process in the root window, TotalView will find or open a process window for that process. Holding down the Shift key when you dive will force TotalView to open a new process window focused on that process.
- When you select a thread in the root window, TotalView will find or open a process window for that process and show you the selected thread. If a matching window can't be found, it will replace the contents of an existing process window and show you the selected thread.

- When you dive on a thread in the root window, TotalView will find or open a process window for that process and thread combination. Holding down the Shift key when you dive will force TotalView to open a new process window focused on that thread.

## The Process Window Stack

Whenever the process and/or thread is replaced in the process window, the previous contents of the window are *pushed* onto a stack. The go back button *pops* the stack and shows you the previous contents of the process window. The process window stack is pushed in the following cases:

- Select or dive in the thread list pane in the process window.
- Select or dive on any of the four process/thread previous/next buttons in the process window.
- A select operation in the root window on a process or thread that causes the contents of a process window to be replaced with the selected process or thread.



---

# Using the Mouse Buttons and Menus



The TotalView debugger uses a three-button mouse, as outlined in Table 1.

**Table 1.** Mouse Button Functions

<b>Button</b>	<b>Default Position</b>	<b>Purpose</b>	<b>How to Use It</b>
Select	Left	Select or edit object; scroll windows and panes	Move the pointer over the object and click the button.
Menu	Middle	Display pop-up menu	Move the pointer into the window and hold down the button.
		Select command from menu	Move pointer down the menu until the desired command is highlighted, and release the button.
		Leave menu without selecting command	Move the pointer off the menu and release the button.
Dive	Right	Dive into object to display information about it	Move the pointer over the object and click the button.

---

The select button has a special function in the tag field area of the source code pane. Selecting a line number for an executable line of code sets a breakpoint at that line. TotalView displays a STOP sign in the tag field. If a breakpoint has already been set, selecting the STOP sign clears (deletes) the breakpoint. If an evaluation or event point has already been set (indicated by an EVAL or GIST sign), selecting the sign disables it. For more information on breakpoints, evaluation points, and event points, refer to “Setting Action Points” on page 133.

Each TotalView window displays information and provides a pop-up menu of commands for examining and manipulating the information. Many commands have keyboard equivalents, which are displayed to the right of the command on the pop-up menu. This manual displays the keyboard equivalent for a command in parentheses: **Go Group (G)**. Some menus have submenus, which we indicate in the manual with the  symbol: **Go/Halt/Step/Next  Go Group(G)**.

The following commands are only available from the keyboard:

Control-L	Repaints the current window.
Control-Q	Exits from the debugger after you confirm.
Control-R	Raises the root window.
Shift-Return	Ends a field editor session. See “Editing Text” on page 38.
Control-C	Cancel the single-step operation and other time-consuming operations, such as searching for a string.

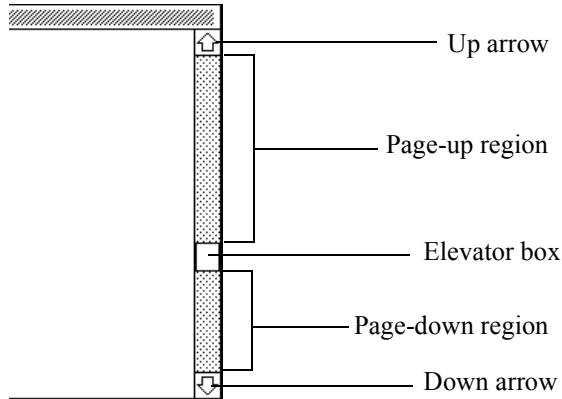
---

## Scrolling Windows and Fields

### Scrolling Windows

You can use the scroll bars to scroll through the information in TotalView windows and panes, as shown in Figure 9.

- To scroll one line at a time, click the Select mouse button (the left mouse button) on the up or down arrows (at the top and bottom of the scroll bar).
- To scroll one page at a time, click the Select mouse button above or below the elevator box inside the scroll bar.
- To scroll an arbitrary amount, hold down the Select mouse button and drag the elevator box inside the scroll bar.



**Figure 9.** Scroll Bar

To scroll continuously by line or by page, you can hold down the Select mouse button instead of clicking it. If TotalView scrolls too fast or too slow, you can adjust the scrolling speed using X resources. Refer to “totalview\*scrollLineSpeed: n” on page 170 for further information.

You can also scroll windows using the keys on your keyboard’s numeric keypad:

↑	Scrolls up one line.
Meta-↑	Scrolls up one page.
↓	Scrolls down one line.
Meta-↓	Scrolls down one page.
Page up	Scrolls up one page.
Page down	Scrolls down one page.

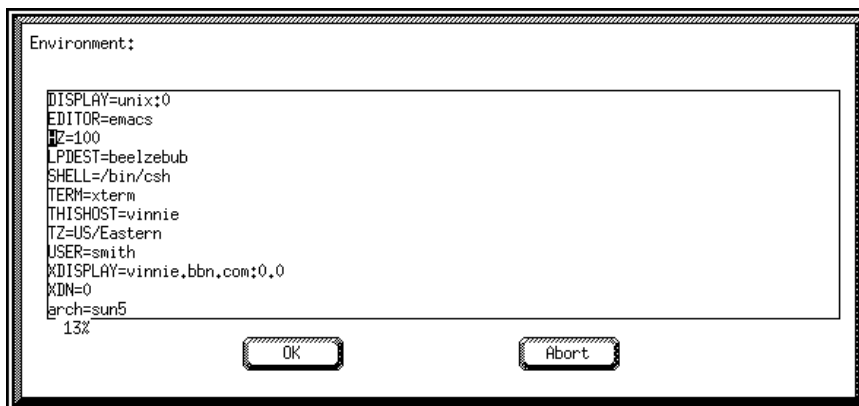
On some platforms, you may need to adjust your X Window System keyboard mapping to use certain keys on the numeric keypad. Refer to your platform-specific supplement, such as the *TotalView Supplement for LynxOS Users*, for details.

## Scrolling Multiline Fields

You can scroll multiline fields in dialog boxes, allowing you to create more lines than are visible. The bottom left corner of the multiline field indicates your location in the field with the following symbols:

- **All** – All of the lines in the field are visible.
- **Top** – The top line of the field is visible, but there are more lines below the bottom of the field that are not visible.
- **Bot** – The bottom line of the field is visible, but there are more lines above the top of the field that are not visible.
- *nn%* – The percentage of the lines above the top of the field that are not visible.

Figure 10 shows an example of a scrollable multiline field.



**Figure 10.** Scrollable Multiline Field

You can use the  $\uparrow$  key or **Control-P** to move up a line in a multiline field. You can also use the  $\downarrow$  key or **Control-N** to move down a line in a multiline field. When you move off the top or bottom of the field and there are more lines above or below, the field scrolls automatically by one line.

You can scroll a multi-line field by more than one line at a time by combining **Control-U** with any of the other commands for moving up or down a line. When you precede an editing command with **Control-U**, it repeats the command four Times New Roman. For example, if you enter **Control-U Control-P**, the cursor moves up four lines.

---

## Getting Help

You can request help from any TotalView window or dialog box by selecting the **Help** command from the pop-up menu or by pressing **Control-?**. When you request help, a separate help window appears. To close the help window, select the **Close Window (q)** command from the menu.

---

## Diving into Objects

To display more detail about an object (such as a variable), you dive into it by clicking the Dive mouse button. You can dive into any object that has a block of data associated with it, such as a pointer, structure, or subroutine. TotalView displays the information about the object in the current window or in a separate window, as outlined in Table 2.

**Table 2.** Uses for Diving

<b>Object</b>	<b>Information Displayed by Diving</b>
Process or thread	A process window appears focused on a thread. See “Using the Primary Windows” on page 26.
Routine in the stack trace pane	The stack frame and source code for the routine appear in the process window.

**Table 2.** Uses for Diving (Continued)

<b>Object</b>	<b>Information Displayed by Diving</b>
Pointer	The referenced memory area appears in a separate variable window.
Variable	The contents of the variable appears in a separate variable window.
Array element, structure element, or referenced memory area	The contents of the element or memory area replace the contents that were in the variable window. This is known as a <i>nested</i> dive.
Subroutine <sup>1</sup>	The source code for the subroutine appears in the process window.

1. A subroutine must be compiled with source line information (usually, with the **-g** switch) for you to dive into it and see source code. If the subroutine was not compiled with source line information, the debugger displays the Assembler code for the routine.

For more information about diving, refer to “Diving in Variable Windows” on page 114.

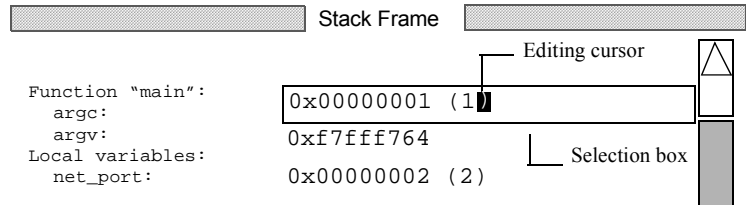
---

## Editing Text

To edit the text in TotalView windows and dialogs, you use the field editor, which is a basic text editor. To do so:

1. Select the text.

If you can edit the selected text, it is enclosed in a rectangle, and the *editing cursor* (a black rectangle) appears, as shown in Figure 11.



**Figure 11.** Editing Cursor

2. Edit the text and press Return (for single-line fields) or Shift-Return (for multiline fields).

You can copy and paste text within TotalView windows, between TotalView windows, or between TotalView windows and other X Window System windows.

To copy text from a TotalView window into an X Window System copy buffer:

1. Select the text or text field from which the text is to be copied by clicking the Select mouse button.
2. Move the editing cursor to the beginning of the text to be copied.
3. Move the mouse pointer (*not* the editing cursor) to the end of the text to be copied.
4. Hold down Control and click the Dive mouse button. Although you do not see anything happen on the screen, the text between the editing cursor and the mouse pointer is copied to an X Window System copy buffer.

To paste text from an X Window System copy buffer into a TotalView window:

1. Select the text or text field in which the text is to be pasted.
2. Move the editing cursor to the location where you want to paste the text.
3. Move the mouse pointer (*not* the editing cursor) anywhere in the text field.

4. Hold down Control and click the Menu mouse button. The text is pasted into the field at the location of the editing cursor.

---

**Note:** The preceding steps apply to copy and paste operations for TotalView windows only, not to other X Window System clients.

---

The field editor supports some of the same commands as GNU Emacs, as outlined in Table 3.

**Table 3.** Field Editor Commands

<b>Keystrokes</b>	<b>Action</b>
Control-A	Move the cursor to the beginning of the line.
Control-B	Move the cursor backward one character.
Control-C	Abort the field editor, and discard all changes.
Control-D	Delete the character under the cursor.
Control-E	Move the cursor to the end of the line.
Control-F	Move the cursor forward one character.
Control-H, Backspace, or Delete	Delete the previous character.
Control-K	Delete all text to the end of the line, or delete a newline character.
Control-N	Move the cursor to the next line (in fields with multiple lines only).
Control-O	Insert a newline (in fields with multiple lines only).
Control-P	Move the cursor to the previous line (in fields with multiple lines only).
Control-U [ <i>n</i> ]	Multiply the number of times the command is executed by <i>n</i> . <i>n</i> is optional; the default is 4. Issue this command in combination with another command. For example, to move the cursor forward 50 characters, you enter: Control-U 50 Control-F.



**Table 3.** Field Editor Commands (Continued)

<b>Keystrokes</b>	<b>Action</b>
Tab	Space over to the next tab stop. (Tab stops are located every four characters.)
Return	For single-line fields, stop the field editor and deselect the field. In dialog boxes, confirm the dialog box as if the <b>OK</b> , <b>Continue</b> or <b>Yes</b> button was selected. For multi-line fields, insert a newline.
Shift-Return	For both single-line and multi-line fields, stop the field editor and deselect the field. In dialog boxes, confirm the dialog box as if the <b>OK</b> , <b>Continue</b> or <b>Yes</b> button was selected.
↑, ↓, ←, →	Move up, down, backward, and forward one character.

---

## Searching for Text

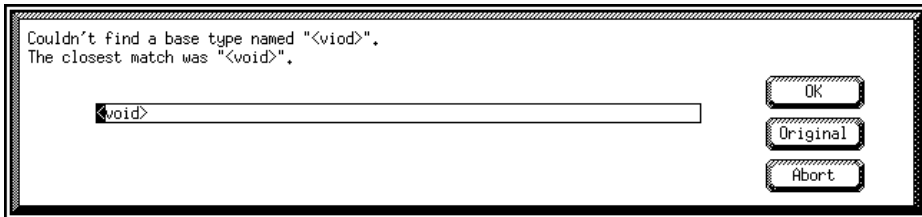
You can search for text strings in most TotalView windows. You can use the following commands:

- |                                       |  |
|---------------------------------------|--|
| <b>Search for String... (f)</b>       | Searches forward in the window for a text string. The debugger prompts you for the string. The search starts from the first line of text that is <i>visible</i> in the window.   |
| <b>Search Backward for String (b)</b> | Searches backward in the window for a text string. The search starts from the last line of text that is <i>visible</i> in the window.  |
| <b>Reexecute Last Search (.)</b>      | Repeats the last forward or backward search without prompting for a string. The search starts from the point where the last search left off and continues in the same direction. |

---

## Using the Spelling Corrector

TotalView checks the spelling of text entries for certain commands. If TotalView does not find the name you entered, it displays a dialog box with the closest match, as shown in Figure 12.



**Figure 12.** Dialog Box for Spelling Corrector

You can edit the closest match, and then select **OK** to use it, **Original** to get back the original text, or **Abort** to cancel.

To customize the behavior of the spelling corrector with X Window System resources, refer to “totalview\*spellCorrection: {verbose|brief|none}” on page 173.

---

## Saving the Contents of Windows

You can save the contents of most window panes as ASCII text. You can save the contents in the following ways:

- Write it to a file. When you specify *filename*, TotalView creates the file (if it does not exist) and overwrites its contents with the text.
- Append it to a file. When you specify *+filename*, TotalView creates the file (if it does not exist) and appends the text to the end of it.

- Pipe it to UNIX shell commands. When you specify *|command...*, TotalView pipes the commands to **/bin/sh** for execution. You can use a series of complex shell commands if desired. For example, to ignore the top five lines of output, compare the current ASCII text to an existing file, and write the differences to another file, you specify:

```
|tail +5 | diff - filename > filename.diff
```

To save the contents of the current window pane:

1. Move the mouse pointer into the desired pane.
2. Select the **Save Window to File...** command.
3. Enter *filename*, *+filename*, or *|command...* in the dialog box.
4. Press Return.

To save a series of panes in a window, you can use the **Reexecute Last Save Window** command. This command repeats the last **Save Window to File...** command (including the information entered in the dialog box) but for the current window pane.

---

## Exiting from the TotalView Debugger

You can exit from the debugger in two different ways:

- Press Control-Q in any window.
- Select the **Quit Debugger (q)** command in the root window.

In the dialog box, select **yes** or type **y** to confirm. To cancel the exit, select **no** or type **n**.



## **CHAPTER 3:**

# **Setting Up a Debugging Session**

This chapter explains how to set up TotalView sessions to suit your needs. For example, you may need to examine a core file instead of a process or pass arguments to your program. You'll learn how to:

- Compile programs
- Start TotalView
- Load executables
- Attach to processes
- Debug remote processes
- Detach from processes
- Examine core files
- Start the debugger server for remote debugging
- Debug over a serial line
- Determine the status of processes and threads
- Handle signals
- Set search paths
- Set command arguments
- Specify environment variables
- Set input and output files
- Monitor your TotalView session

---

# Compiling Programs

Before you start to debug a program with the TotalView debugger, you must compile the program with the appropriate switches and libraries for your situation. Table 4 discusses some general considerations, but you must check your TotalView platform-specific supplement (such as the *TotalView Supplement for LynxOS Users*) to determine the exact syntax and any other considerations for your platform.

**Table 4.** Compiler Considerations

---

<b>Compiler Switch or Library</b>	<b>What It Does</b>	<b>When to Use It</b>
Debugging symbols switch (usually <b>-g</b> )	Generates debugging information in the symbol table	Before debugging <i>any</i> program with TotalView
Optimization switch (usually <b>-O</b> )	Moves code around to optimize execution of program <sup>1</sup>	After you finish debugging your program with TotalView
Multiprocess programming library (usually <b>dbfork</b> )	Uses special versions of the <b>fork()</b> and <b>execve()</b> system calls	Before debugging a multiprocess program that explicitly calls <b>fork()</b> or <b>execve()</b> <sup>2</sup>

---

1. Some compilers don't permit you to use the **-O** switch simultaneously with the **-g** switch. Even if your compiler does permit this, we recommend against it. Although you can do some debugging with the **-O** option on, your debugging session may produce strange results.

2. Refer to "Processes That Call fork()" on page 138 and "Processes That Call execve()" on page 138.

---

## Starting the TotalView Debugger

The complete command syntax for starting the TotalView debugger is as follows:

```
% totalview [filename [corefile]] [options]
```

where *filename* specifies the name of the object file to be debugged and *corefile* specifies the name of the core file to be debugged.

Here are some of the most common ways of starting the debugger:

<b>totalview</b>	Starts the debugger without loading a program or core file. Once in TotalView, you can load a program by issuing the <b>New Program Window (n)</b> command from the root window.
<b>totalview filename</b>	Starts the debugger and loads the program specified by <i>filename</i> .
<b>totalview filename corefile</b>	Starts the debugger and loads the program specified by <i>filename</i> and the core file specified by <i>corefile</i> .
<b>totalview filename -a args</b>	Starts the debugger and passes all subsequent arguments (specified by <i>args</i> ) to the program specified by <i>filename</i> . The <b>-a</b> option must be the last option on the command line.
<b>totalview filename -grab</b>	Starts the debugger and grabs the keyboard whenever it displays a dialog box. You should use this option whenever you start TotalView with a window manager that uses a “click-to-type” model.

**totalview** *filename* **-remote** *hostname*[:*portnumber*]

Starts the debugger on this host and the TotalView debugger server (**tvdsrv**) on the remote host. Loads the program specified by *filename* on the remote host *hostname* for remote debugging. You can specify a hostname or TCP/IP address for *hostname*, and optionally, a TCP/IP port number for *portnumber*.

For more information on the **totalview** command, refer to “TotalView Debugger Server Command Syntax” on page 181. For more information on remote debugging, refer to “Debugging Remote Processes” on page 53, “Starting the Debugger Server for Remote Debugging” on page 57, and “TotalView Debugger Server Command Syntax” on page 181.

---

## Loading Executables

### Loading a New Executable

If you did not load an executable when starting TotalView, you can load one at any time using the **New Program Window** command. To do so, do the following:



1. From the root window, select the **New Program Window (n)** command. A dialog box appears, as shown in Figure 13.

The dialog box is titled "New Program Window Dialog Box" and contains the following elements:

- A label "Executable file name:" followed by a text input field.
- A label "Filter" followed by a text input field.
- Two radio buttons:  "Find or create a process" and  "Create a new process".
- A horizontal dotted line separator.
- A label "Attach to existing process or core file" followed by a text input field.
- Two radio buttons:  "Attach to an existing process (Enter PID)" and  "Core file (Enter core file name)".
- A horizontal dotted line separator.
- A label "Program location (or blank if local):" followed by a text input field.
- Two radio buttons:  "Remote host (Enter remote host name or IP address)" and  "Serial line (Enter device name)".
- Two buttons at the bottom: "OK" and "Abort".

**Figure 13.** New Program Window Dialog Box

2. Enter the name of the executable in the top section of the dialog box. The name can be a full or relative pathname.

If you supply a simple filename, TotalView searches for it in the list of directories specified with the **Set Search Directory** command and specified by your PATH environment variable.

3. (Optional) If you prefer to create a brand new process instead of reusing an existing one (the default), select the **Create a new process** radio button. When you select this option, TotalView creates a new entry in the root window for the process.
4. Press Return.

---

**Note:** If you use the **New Program Window** command to load the same executable again, TotalView does not reread the executable, and it reuses the existing symbol table. To have TotalView reread the executable, you need to use the **Reload Executable File** command, as described in the next section.

---

## Reloading a Recompiled Executable

If you have edited and recompiled your program during a debugging session, you can reload your updated program without exiting from the debugger. To do so:

1. Confirm that the current process has exited. If it has not, display the **Arguments/Create/Signal** submenu and select the **Delete Process (^Z)** command from the process window.
2. Confirm that duplicate copies of the process do not exist by issuing the **ps** command in a shell. If duplicate processes exist, delete them with the **kill** command.
3. Recompile your program.
4. In the process window, display the **Arguments/Create/Signal** submenu and select the **Reload Executable File** command. The debugger updates the process window with the new source file and loads a new executable file. The next time you start the process, the debugger uses the new executable file.

---

## Attaching to Processes

If a program you are testing is hung or looping (or misbehaving in some other way), you can attach to it with TotalView. You can attach to single processes, multiprocess programs, and remote processes.

To attach to a process, you can either use the **Show All Unattached Processes** or **New Program Window** commands.

---

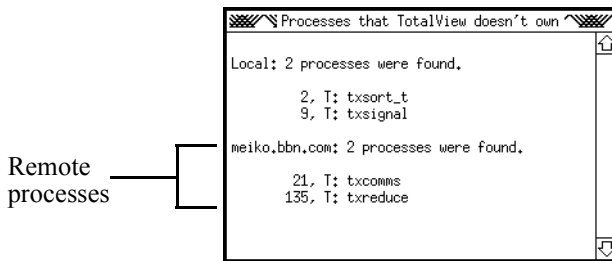
**Note:** If the process or any of its children has called the **execve()** routine, you may need to attach to it by creating a new program window. The reason for this is that on some platforms TotalView uses the **ps** command to obtain the name of the executable file for the process. Since **ps** can give incorrect names, TotalView might not be able to find the executable for the process.

---

To attach to a process using the **Show All Unattached Processes** command, go to the root window and complete the following steps:

1. Select the **Show All Unattached Processes (N)** command.

The unattached processes window appears, as shown in Figure 14. This window lists the process ID, status, and name of each process that is *not* attached to the debugger and is associated with your username.



**Figure 14.** Unattached Processes Window

If you are debugging a remote process in this session, the unattached processes window also shows processes running under your username on each remote hostname. You can attach to any remote process listed. For more information on remote debugging, refer to “Starting the Debugger Server for Remote Debugging” on page 57 and “TotalView Debugger Server Command Syntax” on page 181.

2. Dive into the process you wish to debug.

A process window appears. The right arrow points to the current PC (where the program is executing or hung).

To attach to a process with the **New Program Window (n)** command, follow these steps:

1. Get the process ID (PID) of the process by using the **ps** command in a shell.

2. Issue the **New Program Window (n)** command from the root window. A dialog box appears, as shown in Figure 15.

Executable file name:

Filter

Find or create a process  
 Create a new process

Attach to existing process or core file

13478

Attach to an existing process (Enter PID)  
 Core file (Enter core file name)

Program location (or blank if local):

Remote host (Enter remote host name or IP address)  
 Serial line (Enter device name)

OK Abort

**Figure 15.** New Program Window Dialog Box

3. Enter the name of the executable in the top section of the dialog box. The name can be a full or relative pathname. If you supply a simple filename, TotalView searches for it in the list of directories specified with the **Set Search Directory** command and specified by your PATH environment variable.
4. Enter the process ID (PID) of the *unattached* process in the middle section of the dialog box.
5. Press Return.

If the executable is a multiprocess program, the debugger asks you if you want to attach to all relatives of the process. If you want to examine all processes, select **Yes**.

If the process has children that called **execve()**, the debugger tries to determine the correct executable for each of them. If the debugger cannot determine the executables for the children, you need to delete (kill) the parent process and start it again using TotalView.

Finally, a process window appears. The right arrow points to the current PC (where the program is executing or is hung).

---

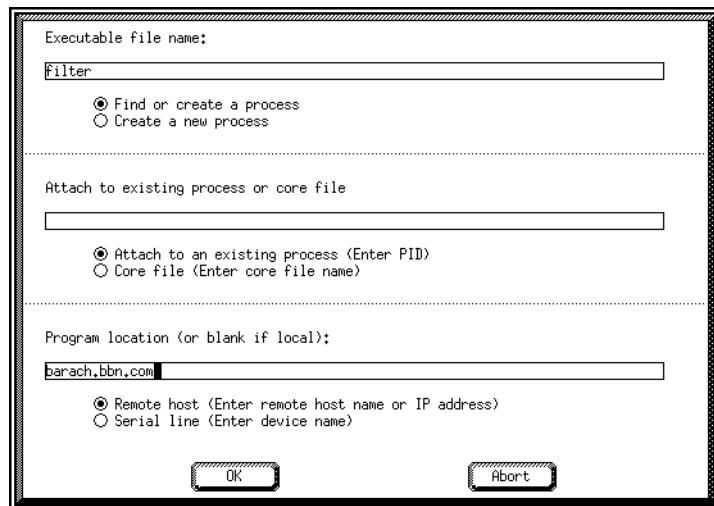
## Debugging Remote Processes

You can begin debugging remote processes by either loading a remote executable, or attaching to a remote process. Note that you cannot examine remote core files.

### Loading a Remote Executable

To load a remote program into TotalView, do the following:

1. Complete steps 1 to 3 in “Loading a New Executable” on page 48.
2. Enter the hostname or TCP/IP address of the machine on which the executable should be running in the bottom section of the dialog box, as shown in Figure 16.



**Figure 16.** New Program Window Dialog Box

---

**Note:** On some multiprocessor platforms, there will be additional radio buttons in the lower section of the dialog box. You can use these buttons for debugging programs that are running on groups or clusters of processors. For information about these radio buttons, refer to your platform-specific supplement, such as the *TotalView Supplement for LynxOS Users*.

---

3. Press Return.

---

**Note:** If this method does not work, you might need to disable the auto-launch feature for this connection and start the debugger server manually. In step 2, as an alternative, you can specify *hostname:portnumber*, where *portnumber* is the TCP/IP port number on which the debugger server (**tvdsvr**) is communicating with TotalView. For more information on this alternative, refer to “Starting the Debugger Server for Remote Debugging” on page 57.

---

## Attaching to a Remote Process

To attach to a remote process, complete the following steps:

1. Complete steps 1 to 4 on page 51 and page 52.
2. Enter the hostname or TCP/IP address of the machine on which the executable should be running in the bottom section of the dialog box.

---

**Note:** On some multiprocessor platforms, there will be additional radio buttons in the lower section of the dialog box. You can use these buttons for debugging programs that are running on groups or clusters of processors. For information about these radio buttons, refer to your platform-specific supplement, such as the *TotalView Supplement for LynxOS Users*.

---

3. Press Return.

---

**Note:** If this method does not work, you might need to disable the auto-launch feature for this connection and start the debugger server manually. In step 2, as an alternative, you can specify *hostname:portnumber*, where *portnumber* is the TCP/IP port number on which the debugger server (**tvdsvr**) is communicating with TotalView. For more information on this alternative, refer to “Starting the Debugger Server for Remote Debugging” on page 57.

---

---

## Detaching from Processes

You can detach from any processes to which you have attached (that is, processes that TotalView did not create) when you finish debugging them. When you detach from a process, TotalView removes all breakpoints that you set in that process.

To detach from a process:

1. If you want to send the process a signal, select the **Set Continuation Signal** command. Choose the signal that TotalView should send to the process when it detaches from it. For example, to detach from a process and leave it stopped, set the continuation signal to SIGSTOP.
2. Display the **Arguments/Create/Signal** submenu and select the **Detach from Process** command.

---

## Examining a Core File

If a process encounters a serious error and dumps a core file, you can examine it from the debugger. TotalView provides two different methods for examining a core file:

- You can start the TotalView debugger with the following command:

```
% totalview filename corefile [options]
```

where *corefile* is the name of the core file.

- You can issue the **New Program Window (n)** command from the root window. In the dialog box (shown in Figure 13 on page 49), enter the name of the core file in the middle section of the dialog, select the **Core file** radio button, and press Return.

---

**Note:** You can debug only local core files. TotalView does not support remote debugging of core files.

---

The process window displays the core file, with the stack trace, stack frame, and source code panes showing the state of the process when it dumped core. The title bar of the process window specifies the signal that caused the core dump. The right arrow (?) in the tag field of the source code pane indicates the value of the PC when the process encountered the error.

You can examine all of the variables to see their state at the time the process found the error. For more information on examining variables, refer to “Examining and Changing Data” on page 107.

If you start a process while you are examining a core file, the debugger stops using the core file and starts a fresh process with the executable.



---

# Starting the Debugger Server for Remote Debugging

Debugging a remote process with TotalView is identical to debugging a native process except for the following:

- The performance of your session depends on the performance of the network between the native and remote machines. If the network is overloaded, debugging can be slow. In general, we've designed remote debugging to work with the speeds encountered on a LAN.
- TotalView works with another process running on the remote machine, called the TotalView Debugger Server (**tvdsvr**), to debug the remote process.

The rest of this section discusses the different ways you can start the TotalView debugger server.

---

**Note:** Remote debugging support is an option that you must purchase separately.

---

## The Auto-Launch Feature

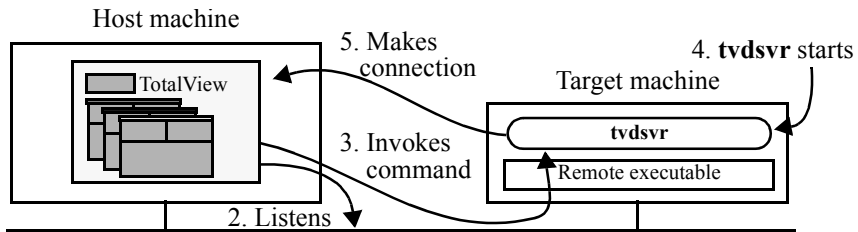
By default, TotalView automatically launches **tvdsvr** for you, which is known as the auto-launch feature. The advantage of auto-launch is that it makes it easy to start debugging remote processes—you don't need to take any action to start the debugger server.

If you want to know more about auto-launch, here is the sequence of actions carried out by you, TotalView, and **tvdsvr** when auto-launch is enabled:

1. With the **New Program Window** command, you specify the hostname of the machine on which you want to debug a remote process, as described in “Debugging Remote Processes” on page 53.
2. TotalView begins listening for incoming connections.

3. TotalView launches the **tvdsvr** process with the server launch command. “The Server Launch Command” on page 58 describes the command in detail.
4. The **tvdsvr** process starts on the remote machine.
5. The **tvdsvr** process establishes a connection with TotalView.

Figure 17 summarizes the actions carried out by the auto-launch feature.



**Figure 17.** Auto-Launch Feature

## Auto-Launch Options

If the auto-launch feature doesn't work on your system, you can tailor it as needed. Specifically, you can tailor the following items:

- The command used by TotalView to launch **tvdsvr**
- The arguments passed to the launch command or to **tvdsvr**
- The length of time TotalView waits (that is, the timeout) to receive a connection from **tvdsvr**
- Whether or not the auto-launch feature is enabled

The only constraint in tailoring auto-launch is that **tvdsvr** must be started on the remote machine with the **-callback** and **-set\_pw** arguments.

## The Server Launch Command

By default, TotalView uses the following command string when it automatically launches the debugger server:

```
rsh %R -n "cd %D && tvdsvr -callback %L -set_pw %P"
```

With this command string, the **rsh** command invokes a shell on the hostname specified by **%R** and invokes the commands enclosed in quotation marks, where:

- |           |  |
|-----------|--|
| <b>%R</b> | Expands to the hostname of the remote machine that you specified in the <b>New Program Window</b> command. |
| <b>-n</b> | Causes the remote shell to read standard input from <b>/dev/null</b> .                                     |

When the remote shell is started by **rsh**, it first changes to the **%D** directory with the **cd** command:

- |           |  |
|-----------|--|
| <b>%D</b> | Expands to the full pathname of the directory to which TotalView is connected. |
|-----------|--|

Note that the “**cd %D**” portion of the command assumes that the host machine and the target machine mount identical filesystems. That is, the pathname of the directory to which TotalView is connected must be identical on both the host and target machines.

Next, the remote shell starts the TotalView Debugger Server with the **tvdsvr** command and the following arguments:

- |                  |  |
|------------------|--|
| <b>-callback</b> | Establishes a connection from <b>tvdsvr</b> to TotalView using the specified hostname and port number.                                       |
| <b>%L</b>        | Expands to the hostname and TCP/IP port number ( <i>hostname:port</i> ) on which TotalView is listening for connections from <b>tvdsvr</b> . |
| <b>-set_pw</b>   | Sets a 64-bit password for security. TotalView must supply this password when <b>tvdsvr</b> establishes the connection with it.              |
| <b>%P</b>        | Expands to the password that TotalView automatically generated.  |

To change the server launch command each time you start TotalView, you can set an X resource. See “totalview\*serverLaunchString: command\_string” on page 171 for more information.

For the complete syntax of the **tvdsvr** command, refer to “TotalView Debugger Server Command Syntax” on page 181.

## Changing the rsh Command

If desired, you can substitute a different command for **rsh**, but the command must invoke the **tvdsvr** process with the arguments shown (**-callback** and **-set\_pw**).

---

**Note:** If you’re not sure whether **rsh** works at your site, try the **rsh hostname** command from an **xterm**, where *hostname* is the name of the host on which you want to invoke the remote process. If this command prompts you for a password, you must add the hostname of the host machine to your **.rhosts** file on the target machine for TotalView to invoke **tvdsvr** properly.

---

For example, although the **rsh** command provides reasonable security, your site may prefer to invoke remote processes with a more secure command. As another example, you could even use a combination of the **echo** and **telnet** commands:

```
echo %D %L %P; telnet %R
```

Once **telnet** establishes the connection to the remote host, you could use the **cd** and **tvdsvr** commands directly, using the values of **%D**, **%L**, and **%P** that were displayed by the **echo** command:

```
% cd directory
```

```
% tvdsvr -callback hostname:portnumber -set_pw  
password
```

If you have no command for invoking a remote process, you cannot use the auto-launch feature and should disable it.

For information on the **rsh** command, refer to the manual page supplied with your operating system.

## Changing the Arguments

You can also change the command-line arguments passed to **rsh** (or whatever command you select to invoke the remote process).

For example, if the host machine does not mount the same filesystems as your target machine, it may need to use a different path to access the executable to be debugged. If this is the case, you could change **%D** to an appropriate directory on the target machine.

If your remote executable reads from standard input, you cannot use the **-n** option with **rsh** because this causes the remote executable to receive an EOF immediately on standard input. If you omit **-n**, the remote executable reads standard input from the **xterm** in which you started TotalView. Therefore, if your remote program reads from standard input, you should invoke **tvdsvr** from an **xterm** window. Use the following command string to launch the debugger server:

```
rsh %R -n "cd %D && xterm -display hostname:0 -e tvdsvr -callback %L -set_pw %P"
```

Now, each time TotalView launches **tvdsvr**, a new **xterm** appears on your screen to handle standard input and output for the remote program.

## The Connection Timeout

When TotalView automatically launches **tvdsvr**, it waits for 30 seconds to receive a successful connection from **tvdsvr**. If TotalView receives nothing, it times out. If desired, you can specify a timeout of anywhere between 1 and 3600 seconds (1 hour).

---

**Note:** If you notice that TotalView fails to launch **tvdsvr** (as shown in the **xterm** window from which you started TotalView) before the timeout expires, you can press Control-C in any TotalView window to have TotalView terminate the launch. Otherwise, TotalView terminates the launch when the timeout occurs.

---

To change the timeout for every TotalView session, you can set an X resource. See “totalview\*serverLaunchTimeout: n” on page 171 for more information.

## Disabling Auto-Launch

If changing the auto-launch options will not make the auto-launch feature useful for you, you can disable the auto-launch feature and start **tvdsvr** manually. You can disable the auto-launch feature in several different ways:

- When you change the auto-launch options, as described in “Changing the Options” on page 62, deselect the **TotalView Debugger Server Auto Launch Enabled** checkbox at the top of the dialog box. This disables auto-launch for your current TotalView session.
- Set an X resource that disables auto-launch, as described in “totalview\*serverLaunchEnabled: {true | false}” on page 170. This disables auto-launch for every TotalView session.

---

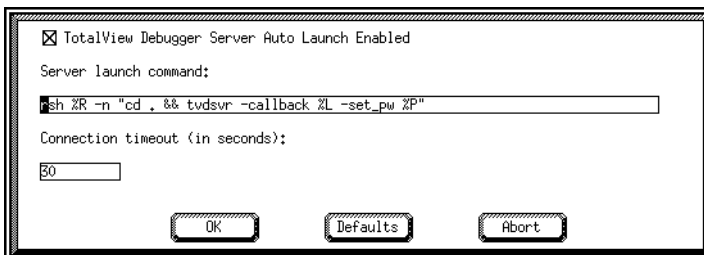
**Note:** If you disable the auto-launch feature, you must start **tvdsvr** *before* you load a remote executable or attach to a remote process.

---

## Changing the Options

To actually change the server launch command or the connection timeout used by TotalView to launch **tvdsvr**, or to actually disable the auto-launch feature entirely, you use the Server Launch Window command. To do so:

1. From the root window, select the **Server Launch Window** menu command. A dialog box appears, as shown in Figure 18.



**Figure 18.** Dialog Box for Launching Debugger Server

2. Change the desired options.

3. Press Return.

---

**Note:** If you make a mistake or decide you want to revert to the default option settings in the dialog, select the **Defaults** button. You can revert to the default settings even if you used an X resource to change the settings. Then, to apply the original option settings, you need to select the **OK** button.

---

## Starting the Debugger Server Manually

If you cannot tailor the auto-launch feature to work on your system, you can start the debugger server manually if needed. The disadvantage of this method is that it is less secure: other users could connect to your instance of **tvdsvr** and begin using your UNIX UID.

To start **tvdsvr** manually:

1. From the root window, select the **Server Launch Window** command. A dialog box appears, as shown in Figure 18.
2. Deselect the **TotalView Debugger Server Auto Launch Enabled** checkbox to disable the auto-launch feature.
3. Press Return.
4. Log in to the remote machine and start **tvdsvr**:

```
% tvdsvr -server
```

The **tvdsvr** command prints out the port number used and the password assigned and then begins listening for connections. Be sure to make note of the password; you'll need to enter it later in step 9.

If the default port number (4142) is not suitable, you need to use the **-port** or **-search\_port** options with the **tvdsvr** command. For details, refer to "TotalView Debugger Server Command Syntax" on page 181.

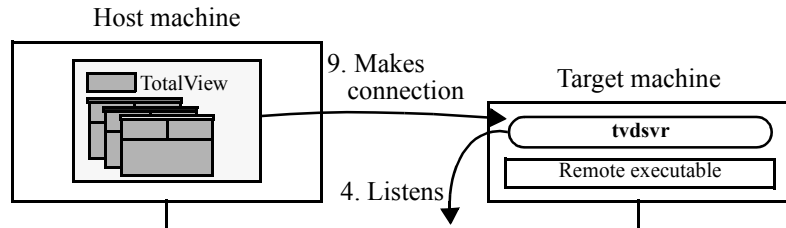
5. From the process window in TotalView, select the **New Program Window** command. A dialog box appears, as shown in Figure 13 on page 49.

6. Enter the name of the executable in the top of the dialog.
7. Enter the *hostname:portnumber* in the bottom of the dialog.
8. Press Return.

TotalView now attempts to establish a connection to **tvdsvr**.

9. When TotalView prompts you for the password, enter the password that **tvdsvr** displayed in step 4.

Figure 19 summarizes the steps used when you start **tvdsvr** manually.



**Figure 19.** Manual Launching of Debugger Server



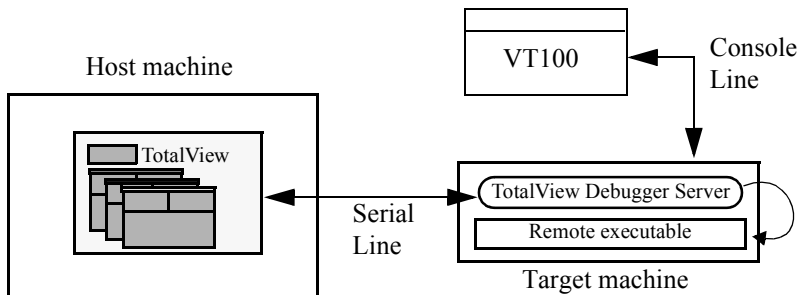
---

## Debugging Over a Serial Line

In addition to debugging over a TCP/IP socket connection, TotalView allows you to debug over a serial line. However, in cases where a network connection exists, you will probably want to use TCP/IP sockets remote debugging for better performance.

You will need to have two connections to the target machine. One connection will be for the console and the other dedicated for use by TotalView. Do not try to use one serial line; TotalView cannot share a serial line with the console.

Figure 20 shows an example TotalView debugging session over a serial line. In this example, TotalView is running on a host machine and communicating over a dedicated serial line with the TotalView Debugger Server running on the target host. A VT100 terminal is connected to the target host's console line, allowing you to type commands on the target host.



**Figure 20.** TotalView Debugging Session over a Serial Line

## Start the TotalView Debugger Server

To start a TotalView debugging session over a serial line from the command line, you must first start the TotalView debugger Server.

Through the console connected to the target machine, issue the command to start the TotalView Debugger Server (**tvdsvr**) and specify the name of the serial port device on the target machine. The syntax of the TotalView Debugger Server command is:

```
% tvdsvr -serial device[:options]
```

where *device* is the name of the serial line device and *options* are options to control the serial line on the target machine. The TotalView Debugger Server will wait for TotalView to establish a connection.

For example:

```
% tvdsvr -serial /dev/com1:baud=38400
```

```
TotalView Debugger Server 4.0 (ICCDP protocol level 17,  
rev 15)
```

```
Copyright 1998-2002 by LynuxWorks, Inc. ALL RIGHTS  
RESERVED.
```

```
Copyright 1996-1998 by Dolphin Interconnect Solutions, Inc.  
ALL RIGHTS RESERVED.
```

```
Copyright 1989-1996 by BBN Inc.
```

Currently the only option you are allowed to specify is the baud rate, which defaults to **38400**.

## Starting TotalView on a Serial Line

Start TotalView on the host machine and include the name of the serial line device. The syntax of the TotalView command is:

```
% totalview -serial device[:options] filename
```

where *device* is the name of the serial line device on the host machine, *options* are options to control the serial line on the host machine and *filename* is the name of the executable file. TotalView will connect to the TotalView Debugger Server.

For example:

```
% totalview -serial /dev/term/a test_pthreads
```

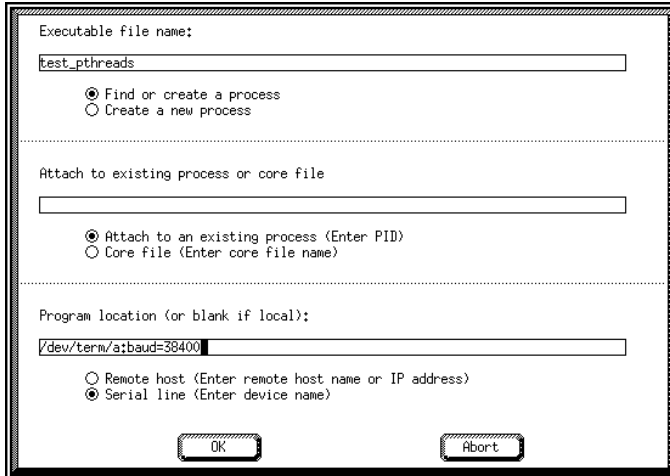
Currently the only option you are allowed to specify is the baud rate, which defaults to **38400**.

## New Program Window

To start a TotalView debugging session over a serial line when you are already in TotalView, do the following:

1. Start the TotalView Debugger Server. See “Start the TotalView Debugger Server” on page 66.
2. Issue the **New Program Window (n)** command from the root window to display the New Program Window dialog box, shown in Figure 21.
3. Enter the name of the executable file in the **Executable file name** field.
4. Enter the name of the serial line device in the **Program location** field, and select the **Serial line** radio button.

5. Press Return or select OK.



**Figure 21.** New Program Window Dialog Box

---

## Determining the Status of Processes and Threads

Process and thread states are displayed in:

- The root window, for processes and threads
- The unattached processes window, for processes
- The process and thread status bars of the process window, for processes and threads
- The thread list pane of the process window, for threads

## Process Status

The root window displays a single character to identify the state of a process. The process status in the root window takes the following form:

*[L] N S process\_name*

where *[L]* is the process location (present only for remote processes), *N* is the process ID, *S* is the single-character representation of the process state, and *process\_name* is TotalView's name for the process.

The unattached processes window lists all running processes that are not attached to the debugger but are associated with your username. The format of the information in the unattached process window is similar to the format of processes in the root window. Process states are specified with a single character.

The process status bar of the process window displays information in the following format:

**Process** *[L] N: process\_name (state)*

where *[L]* is the process location (present only for remote processes), *N* is the process ID, *process\_name* is TotalView's name for the process, and *state* is the state name of the process based on the state of its threads.

## Thread Status

The root window displays a single character to identify the state of a thread. The thread status in the root window takes the following form:

*T S in routine\_name*

where *T* is the TotalView assigned thread ID, *S* is the single-character representation of the thread state, and *routine\_name* is the name of the routine in which the thread was executing when last stopped by TotalView.

The thread list pane in the process window uses the same thread status format as the root window.

The thread status bar of the process window displays information in the following format:

**Thread *N:T*: *process\_name* (*state*) <*reason*>**

where *N* is the process ID, *T* is the TotalView assigned thread ID, *process\_name* is TotalView's name for the process, *state* is the state name of the thread, and <*reason*> is the reason the thread stopped.

## Unattached Process States

The state information for a process displayed in the unattached processes window is derived from the system. The state characters TotalView uses to summarize the state of an unattached process do not necessarily match those used by the system.

Table 5 summarizes the possible states in the unattached processes window.

**Table 5.** Summary of Unattached Process States

State	State Character	Meaning for a process
Running	<b>R</b>	Process is running or can run.
Stopped	<b>T</b>	Process is stopped.
Idle	<b>I</b>	Process has been idle or sleeping for <i>more</i> than 20 seconds.
Sleeping	<b>S</b>	Process has been idle or sleeping for <i>less</i> than 20 seconds.
Zombie	<b>Z</b>	Process is a “zombie,” a child process that has terminated and is waiting for its parent process to gather its status.

## Attached Process States

The state of processes and threads that TotalView is attached to is displayed in various windows.

Table 6 summarizes the possible states for an attached process or thread, and how the states are displayed.

**Table 6.** Summary of Attached Process and Thread States

State Name	State Character	Meaning for a thread and process
<b>Exited or never created</b>	Blank	<i>Process only:</i> does not exist.
<b>Running</b>	<b>R</b>	<i>Thread:</i> is running or can run. <i>Process:</i> all threads in the process are running or can run.
<b>Mixed</b>	<b>M</b>	<i>Process only:</i> some threads in the process are running and some are not running, or the process is expecting some of its threads to stop.
<b>Error &lt;reason&gt;</b>	<b>E</b>	<i>Thread:</i> is stopped because of error <i>reason</i> . <i>Process:</i> one or more threads are in the <b>Error</b> state.
<b>At Breakpoint</b>	<b>B</b>	<i>Thread:</i> stopped at a breakpoint. <i>Process:</i> one or more threads are stopped at a breakpoint, but none are in the <b>Error</b> state.
<b>Stopped &lt;reason&gt;</b>	<b>T</b>	<i>Thread:</i> stopped because of <i>reason</i> , but not at a breakpoint and not because of an error. <i>Process:</i> one or more threads are stopped, but none are in the <b>At Breakpoint</b> state and none are in the <b>Error</b> state.

The **Error** state usually indicates that your program received a fatal signal from the operating system. Some signals, such as SIGSEGV, SIGBUS, and SIGFPE may indicate an error in your program. You can control how TotalView handles signals your program receives.

For more information on signals, refer to “Handling Signals” on page 72.

---

# Handling Signals

If your program contains a signal handler routine, you might need to adjust the way the debugger handles signals. You can change the way in which TotalView handles signals by using a dialog box (described in this section), an X resource (see “totalview\*signalHandlingMode: action\_list” on page 171), or a command-line option to the **totalview** command (refer to “TotalView Command Syntax” on page 175).

By default, TotalView handles UNIX signals as outlined in Table 7.

**Table 7.** Default Signal Handling Behavior

---

<b>Signals that are Passed Back to Your Program</b>	<b>Signals that Stop Your Program or Cause an Error</b>
SIGHUP	SIGILL
SIGINT	SIGTRAP
SIGQUIT	SIGIOT
SIGKILL	SIGEMT
SIGALRM	SIGFPE
SIGURG	SIGBUS
SIGCONT	SIGSEGV
SIGCHLD	SIGSYS
SIGIO	SIGPIPE
SIGVTALRM	SIGTERM
SIGPROF	SIGTSTP
SIGWINCH	SIGTTIN
SIGLOST	SIGTTOU
SIGUSR1	SIGXCPU
SIGUSR2	SIGXFSZ

---

**Note:** The SIGTRAP and SIGSTOP signals are used internally by the TotalView debugger. If the process encounters any of these signals, TotalView neither stops the process with an error nor passes the signal back to your program. Further, you cannot alter the way the debugger uses these signals.

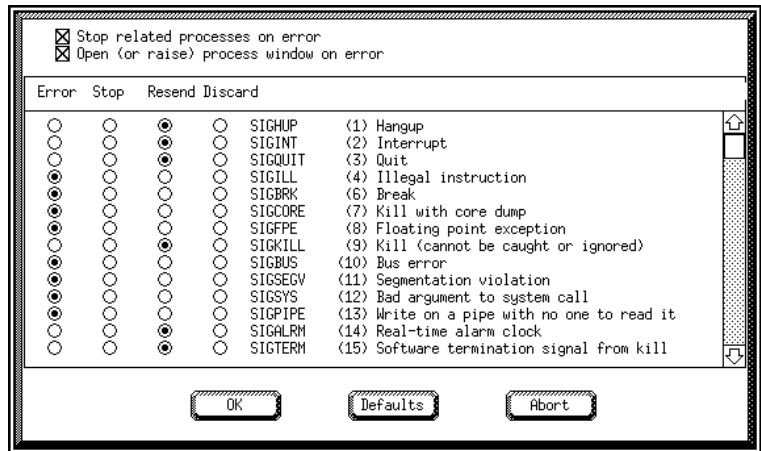
---



Some hardware registers can affect how signals are handled on your platform, such as the SIGFPE signal and others. For more information, refer to “Interpreting Status and Control Registers” on page 89 and your platform-specific supplement, such as the *TotalView Supplement for LynxOS Users*.

If the TotalView debugger’s defaults are not satisfactory, you can change the signal handling mode. To do so, go to the process window and complete the following steps:

1. Display the **Arguments/Create/Signal** submenu and select the **Set Signal Handling Mode...** command. A dialog box appears, as shown in Figure 22.



**Figure 22.** Dialog Box for Set Signal Handling Mode Command

---

**Note:** The set of signal names and numbers shown in the dialog box are platform-specific. The dialog box displayed on your platform may have additional signals and different signal numbers.

---

2. By default, when your program encounters an error signal, TotalView stops all related processes and opens or raises the process window. If you do not want this behavior, deselect the appropriate checkboxes at the top of the dialog box.

---

**Note:** If the processes in a multiprocess program encounter an error, the debugger automatically opens a process window for only the *first* process that encounters an error. Thus, if your program has many processes, this feature prevents the screen from filling up with process windows.

---

3. Scroll the signal list to the desired signal.
4. For each signal listed in the dialog box, choose one of the following signal handling modes by selecting its radio button:

<b>Error</b>	Stops the process, places it in the error state, and displays an error in the title bar of the process window. If the <b>Stop related processes on error</b> checkbox is selected, the debugger also stops all related processes. You should select this signal handling mode for severe error conditions, such as SIGSEGV and SIGBUS signals.
<b>Stop</b>	Stops the process and places it in the stopped state. Select this signal handling mode if you want the signal to be handled like the SIGSTOP signal.
<b>Resend</b>	Sends the signal to the process. If your program contains a signal handling routine, you should use this mode for all the signals that it handles. By default, the common signals for terminating a process (SIGKILL and SIGHUP) use this mode.

**Discard**

Discards the signal and restarts the process without a signal.

---

**Note:** Don't use Discard mode for fatal signals, such as SIGSEGV and SIGBUS. If you do, the debugger can get caught in a signal/resignal loop with your program, with the signal immediately recurring because of repeated reexecution of the failing instruction.

---

5. Select **OK** to confirm your changes, **Abort** to cancel the changes, or **Defaults** to return to the default mode settings.

---

## Setting Search Paths

If your source code, executable or object files reside in a number of different directories, you can set search paths in the debugger for these directories with the **Set Search Directory** command. By default, the debugger searches the following directories (in order) for source code:

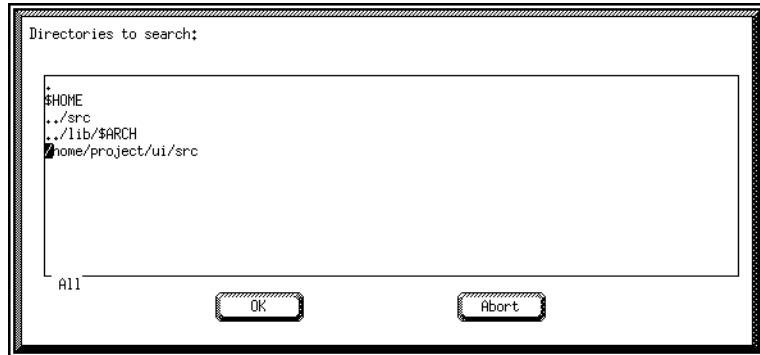
1. The current working directory (.).
2. The directories you specify with the **Set Search Directory** command, in the exact order you enter them in the dialog box.
3. If you specified a full pathname for the executable when you started TotalView, it searches the directory specified.
4. The directories specified in your PATH environment variable.

These search paths apply to *all* processes that you are debugging, and to *all* directory search situations in TotalView.

To use the **Set Search Directory** command, go to the process window and complete these steps:

1. Display the **Display/Directory/Edit** submenu and select the **Set Search Directory... (d)** command.

A dialog box appears, as shown in Figure 23.



**Figure 23.** Dialog Box for Set Search Directory Command

2. Enter the directories in the order you want them searched, separating each directory with a space. You can use multiple lines if needed.

The current working directory (.) is the first directory listed in the window. You can move the current working directory further down the list, but if you remove it, TotalView inserts it at the top of the list again.

You can specify relative pathnames, which are interpreted with respect to the current working directory.

3. Select **OK** (or press Shift-Return).

Once you change the list of directories with the **Set Search Directory** command, the debugger automatically searches again for the source file that is currently displayed in the process window.

---

**Note:** You can specify search directories that apply across TotalView sessions with an X Window System resource. Refer to “totalview\*searchPath: dir1[,dir2,...]” on page 170.

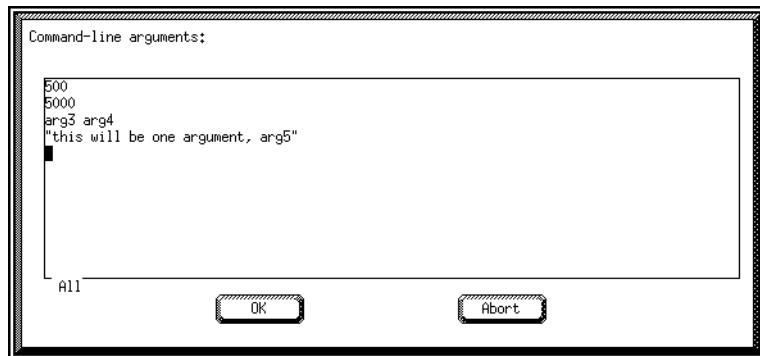
---

---

## Setting Command Arguments

When the debugger creates a process, it passes one argument to the program by default: the name of the file containing the executable code for the process. If your program requires any arguments from the command line, you must set these arguments *before* you start the process. To do so, go to the process window and complete the following steps:

1. Display the **Arguments/Create/Signal** submenu and select the **Set Command Arguments... (a)** command. A dialog box appears, as shown in Figure 24.



**Figure 24.** Dialog Box for Set Command Arguments Command

2. Enter the arguments to be passed to the program. Separate each argument with a space, or place each argument on a separate line. If an argument has spaces in it, enclose the whole argument in double quotes.
3. Select **OK** (or press Shift-Return).

You can also set command-line arguments with the **-a** option of the **totalview** command, as discussed in “Starting the TotalView Debugger” on page 47.

---

## Specifying Environment Variables

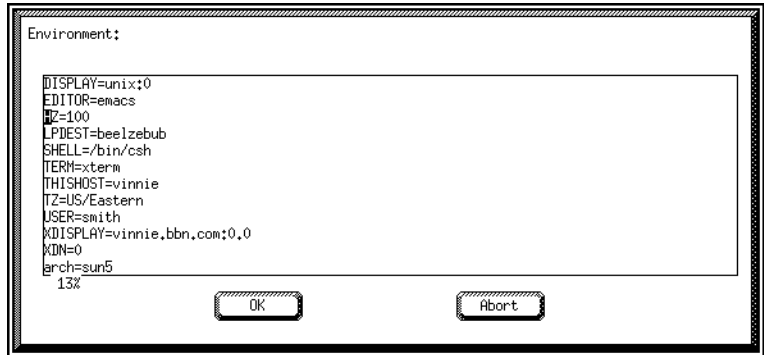
You can set and edit the environment variables that TotalView passes to a process when it creates the process. When TotalView creates a new process, it passes a list of environment variables to the process. By default, a new process inherits TotalView's environment variables, and a remote process inherits **tvdsvr**'s environment variables.

If the environment variable dialog is empty, the process inherits its environment variables from TotalView or **tvdsvr**. If you add environment variables to the dialog, the process no longer inherits its environment variables from TotalView or **tvdsvr**, it only receives the variables specified in the dialog box. Therefore, if you want to add to the variables inherited from TotalView or **tvdsvr**, you must enter all of the variables inherited into the dialog and then make your additions in the dialog.

An environment variable is specified by: *name=value*. For example, **DISPLAY=unix:0.0** specifies an environment variable named **DISPLAY** with the value **unix:0.0**.

To add, delete, or modify the environment variables, go to the process window and complete the following steps:

1. Display the **Arguments/Create/Signal** submenu and select the **Set Environment Variables** command. Figure 25 shows the dialog box.



**Figure 25.** Environment Variables Dialog Box

In the dialog box, you must place each environment variable on a separate line. TotalView ignores blank lines.

2. To change the name or value of an environment variable, edit the line.
3. To add a new environment variable, insert a new line and specify the name and value.
4. To delete an environment variable, delete the line. Deleting all the lines causes the process to inherit TotalView's or **tvdsvr**'s environment.
5. Select **OK** (or press Shift-Return).

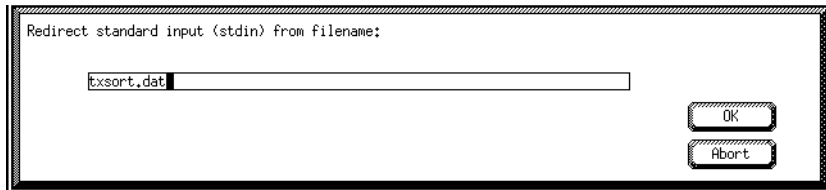
---

## Setting Input and Output Files

Before beginning execution of the program you're debugging, TotalView determines how to handle standard input (**stdin**) and standard output (**stdout**). By default, TotalView creates the program so that it reads **stdin** from and writes **stdout** to the shell window from which you started TotalView.

If desired, you can redirect **stdin** or **stdout** to a file. To do so, complete these steps from the process window before you start executing your program:

1. Display the **Arguments/Create/Signal** submenu and select either **Input from File... (<)** or **Output to File... (>)**. A dialog box appears. Figure 26 shows the dialog for **Input from File**.



**Figure 26.** Dialog Box for Input from File Command

2. Enter the name of the file, relative to your current working directory.
3. Select **OK** (or press Shift-Return).

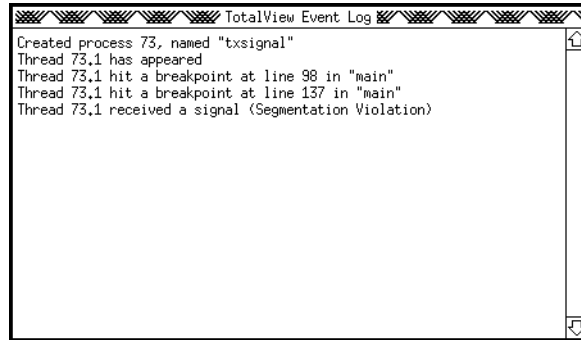
---

## Monitoring TotalView Sessions

The TotalView debugger logs all significant events occurring for all processes you are debugging. To view the event log, go to the root window and select the **Show Event Log Window** command. The event log window displays a sequential list of events that you can scroll.



Figure 27 shows the event log window.



**Figure 27.** Event Log Window



## **CHAPTER 4:**

# **Debugging Programs**

This chapter explains how to debug programs with TotalView. You'll learn how to:

- Examine source and assembler code
- Find the source code for functions
- Edit source text
- Change the editor launch string
- Interpret status and control registers
- Control program execution
- Start processes and threads
- Examine process groups
- Set a breakpoint
- Single step
- Step over functions
- Execute to a selected line
- Execute to the completion of a function
- Continue with a specific signal
- Set the program counter
- Stop processes and threads
- Delete processes
- Restart processes

---

# Examining Source and Assembler Code

In the source code pane of the process window, you can display your program in several different ways, as shown at the top of Table 8. If you display Assembler in the source code pane, you can also display addresses in two different ways, as shown at the bottom of Table 8.

**Table 8.** Ways to Display Source and Assembler Code

<b>To Display This in the Source Code Pane...</b>	<b>Select This from the Display/Directory/Edit Submenu...</b>
Source code only (Default)	<b>Source Display Mode (Meta-s)</b>
Assembler code only	<b>Assembler Display Mode (Meta-a)</b>
Source code interleaved with Assembler code <sup>1</sup>	<b>Interleave Display Mode (Meta-i)</b>
Absolute addresses for all locations and references (Default) <sup>2</sup>	<b>Display Assembler by Address</b>
Symbolic addresses (function names and offsets) for all locations and references <sup>2</sup>	<b>Display Assembler Symbolically</b>

1. Source statements are treated like comments. You can set breakpoints, evaluation points, or event points only at the machine level, not at the source level. Setting an action point at the first instruction after a source statement, however, is equivalent to setting a point at that source statement.

2. If an address matches the address of a function, TotalView displays the function name.

Figure 28 on page 85 illustrates the effect of displaying Assembler code in different ways in the source code pane. You can also display Assembler instructions in a variable window. For more information, see “Displaying Machine Instructions” on page 113.

Assembler Only  
(absolute addresses)

Gridset (dotted grid)  
indicates action point  
can be set on instruction

```

Function _main in filter.c
: 0x000022b4: 0x40000047 call _listen_and_accept
: 0x000022b8: 0x01000000 nop
: 0x000022bc: 0xd027bff8 st %0,[%fp-0x8]
STOP 0x000022c0: 0x9027a010 sub %fp,0x10,%0
: 0x000022c4: 0x40001227 call _pipe
: 0x000022c8: 0x01000000 nop
: 0x000022cc: 0xd027bfe4 st %0,[%fp-0x1c]
: 0x000022d0: 0x9027a018 sub %fp,0x18,%0
: 0x000022d4: 0x40001223 call _pipe
: 0x000022d8: 0x01000000 nop
: 0x000022dc: 0xd027bfe4 st %0,[%fp-0x1c]
: 0x000022e0: 0x40001214 call _fork
: 0x000022e4: 0x01000000 nop
: 0x000022e8: 0xd027bfe0 st %0,[%fp-0x20]
: 0x000022ec: 0xd207bfe0 ld [%fp-0x20],%01
: 0x000022f0: 0x80900009 orcc %g0,%01,%g0
: 0x000022f4: 0x12800025 bne 0x2388
: 0x000022f8: 0x01000000 nop
  
```

Location by absolute address

References by absolute address

Assembler Only  
(symbolic addresses)

Location by function and offset

References by function and offset

```

Function _main in filter.c
: _main+0x24: 0x40000047 call _listen_and_accept
: _main+0x28: 0x01000000 nop
: _main+0x2c: 0xd027bff8 st %0,[%fp-0x8]
STOP _main+0x30: 0x9027a010 sub %fp,0x10,%0
: _main+0x34: 0x40001227 call _pipe
: _main+0x38: 0x01000000 nop
: _main+0x3c: 0xd027bfe4 st %0,[%fp-0x1c]
: _main+0x40: 0x9027a018 sub %fp,0x18,%0
: _main+0x44: 0x40001223 call _pipe
: _main+0x48: 0x01000000 nop
: _main+0x4c: 0xd027bfe4 st %0,[%fp-0x1c]
: _main+0x50: 0x40001214 call _fork
: _main+0x54: 0x01000000 nop
: _main+0x58: 0xd027bfe0 st %0,[%fp-0x20]
: _main+0x5c: 0xd207bfe0 ld [%fp-0x20],%01
: _main+0x60: 0x80900009 orcc %g0,%01,%g0
: _main+0x64: 0x12800025 bne _main+0xf8
: _main+0x68: 0x01000000 nop
  
```

Interleaved  
Source/Assembler  
(absolute addresses)

Source line

Location by absolute address

References by absolute address

```

Function _main in filter.c
30 |
31 | rv = pipe (in_pipe_fds);
STOP 0x000022c0: 0x9027a010 sub %fp,0x10,%0
: 0x000022c4: 0x40001227 call _pipe
: 0x000022c8: 0x01000000 nop
: 0x000022cc: 0xd027bfe4 st %0,[%fp-0x1c]
32 | rv = pipe (out_pipe_fds);
: 0x000022d0: 0x9027a018 sub %fp,0x18,%0
: 0x000022d4: 0x40001223 call _pipe
: 0x000022d8: 0x01000000 nop
: 0x000022dc: 0xd027bfe4 st %0,[%fp-0x1c]
33 | pid = fork();
: 0x000022e0: 0x40001214 call _fork
: 0x000022e4: 0x01000000 nop
: 0x000022e8: 0xd027bfe0 st %0,[%fp-0x20]
34 | if (!pid) {
: 0x000022ec: 0xd207bfe0 ld [%fp-0x20],%01
: 0x000022f0: 0x80900009 orcc %g0,%01,%g0
  
```

Figure 28. Different Ways to Display Assembler Code

---

## Finding the Source Code for Functions

You can search for the source code for a function called by your program in two ways, providing the function was linked to your program at compile time:

- Dive into the function in the source code pane.
- Display the **Function/File/Variable** submenu and select the **Function or File... (f)** command. When prompted, enter the function name in the dialog box.

If the TotalView debugger finds the source code, it displays it in the source code pane. If the function was not compiled with `-g`, the debugger displays the disassembled machine code for the function instead of the source code. TotalView maps filename extensions to source languages as shown in Table 9.

**Table 9.** Source Language Mapping

File Extension	Source Language
.cxx, .cc, .C	C++
.F, .f	Fortran
All others	C

When you want to return to the original contents of the source code pane, dive into the undive icon in the upper right corner of the pane.

---

**Note:** By default, TotalView discards source line information from header files. You can use the **Edit Source Text** command (see “Editing Source Text” on page 87) or your favorite X Window System client (such as **xmore**, **vi**, **emacs**) to display these files while debugging. As an alternative, you can start TotalView with the `-nii` (**-no\_ignore\_includes**) command-line option to have it retain source line information from include files.

---

Except for the file types shown in Table 10, TotalView discards source line information from preprocessor files that generate **#line** directives during preprocessing. To retain source line information from an include file type or preprocessor file type not shown in Table 10, start TotalView with the **-ext *extension*** command-line option. For example, if your executable contains a preprocessed **yacc** file with a nonstandard filename extension named **input.xyz**, you can specify that source line information be retained for the **.xyz** extension as follows:

```
% totalview -ext .xyz
```

**Table 10.** Built-In Preprocessor Extensions for which Source Line Information Is Retained

File Extension	Meaning
.C, .cpp, .cc, .cxx	C++
.F	Fortran preprocessed with <b>/lib/cpp</b>
.l, .lex	<b>lex</b>
.y	<b>yacc</b>

---

## Editing Source Text

You can use the **Edit Source Text (M-e)** command on the **Display/Directory/Edit** submenu to edit source files while you are debugging. TotalView starts your editor on the source file being displayed in the source pane of the process window.

TotalView uses the editor launch string to determine how to start your editor. To change the value of the editor launch string, see “Changing the Editor Launch String” on page 88.

---

## Changing the Editor Launch String

You can change the editor launch string to control the way the debugger starts your editor when you use the **Edit Source Text** command.

The editor launch string is processed by TotalView and expanded into a command string that is then executed by the shell **sh**. This allows you to configure exactly how the editor is started.

TotalView recognizes certain items in the launch string that are expanded before the debugger starts your editor. The expanded items are as follows:

<b>%E</b>	Expands to the value of the EDITOR environment variable, or to <b>vi</b> if EDITOR is not set.
<b>%N</b>	Expands to the line number in the middle of the source pane. Use this option if your editor allows you to specify an initial line number at which to position the cursor.
<b>%S</b>	Expands to the source file name displayed in the source pane.
<b>%F</b>	Expands to the font name with which you started TotalView.

The default editor launch string is:

```
xterm -e %E +%N %S
```

which creates an xterm window in which to run the editor. If you use an editor that creates its own X window, such as **emacs** or **xedit**, you do not need to create the xterm window, and you should change the editor launch string.

You can change the editor launch string by using one of the following methods:



- Using an X resource

Refer to “totalview\*editorLaunchString: command\_string” on page 165 for more information.

- Using the **Editor Launch String...** command on the **Display/Directory/Edit** submenu of the process window.

---

## Interpreting Status and Control Registers

The stack frame pane in the process window lists the contents of CPU registers for the selected frame (you may need to scroll down to see them). To learn about the meaning of these registers, you need to consult the user’s guide for your CPU and your platform-specific supplement (such as the *TotalView Supplement for LynxOS Users*).

For your convenience, TotalView interprets the bit settings of certain CPU registers, such as the registers that control the rounding and exception enable modes. You can edit the values of these registers and continue execution of your program. For example, you might do this to examine the behavior of your program with a different rounding mode.

Since the registers that are interpreted vary from platform to platform, see your platform-specific supplement for information on the registers supported for your CPU. For general information on editing the value of variables (including registers), refer to “Displaying Areas of Memory” on page 112.

---

# Controlling Program Execution

TotalView tries to give you as much control over your program's execution as possible. However, depending on the mix of capabilities available in the target operating system and the characteristics of the target operating system, some of TotalView's commands are disabled or behave differently.

## Capabilities and Characteristics

Capabilities and characteristics for controlling threads and processes vary among operating systems. The precise set of capabilities and characteristics for a given system determines how TotalView behaves when debugging a program on that system.

The following paragraphs summarize some of the capabilities and characteristics that TotalView handles across all platforms. See the platform-specific supplement, such as the *TotalView Supplement for LynxOS Users*, for a description of the capabilities and characteristics of your platform.

- **Synchronous vs. Asynchronous Stop**

Some operating systems implement a synchronous stop model and others implement an asynchronous stop model. With the synchronous stop model, when one thread in the process stops for any reason, they all stop. With the asynchronous stop model, only the thread that encounters the stop condition stops, and does not affect the running state of the other threads.

- **Allows Asynchronous Stop**

Operating systems that allow asynchronous stop enable the debugger to stop one thread without stopping all the threads in the process. Without the asynchronous stop capability, the debugger must stop all the threads in the process even if it only wanted to stop one thread.

- **Synchronous vs. Asynchronous Run**

Some operating systems implement a synchronous run model and others implement an asynchronous run model. With the synchronous run model, when one thread in the process wants to

run for any reason, they must all run. With the asynchronous run model, only the thread that wants to run must be run.

- **Allows Atomic Run**

Some operating systems allow the debugger to atomically continue a set of threads in a single operation. Without this capability, the threads must be continued individually.

- **Allows Read While Running**

Some operating systems allow the debugger to read from (and possibly write to) the process while one or more threads are running. With this capability, TotalView can read (and possibly write) the memory of the process regardless of the state of the threads. Without this capability, TotalView must first stop all the threads before attempting a read or write operation to the process.

- **Allows Multithreaded Signal Delivery**

Some operating systems allow the debugger to continue all the threads in a process with their own signal values. With this capability, you can set a continuation signal for each individual thread, then continue the process. Without this capability, you can specify only one thread in the process to receive a continuation signal when the process is continued.

---

## Starting Processes and Threads

To start a process, go to the process window and select one of the following commands from the **Go/Halt/Step/Next** submenu.

**Go Process (g)**

Creates and starts this process. If the process already exists and is stopped or at a breakpoint, this command resumes execution. Starting a process causes all threads in the process to resume execution.

**Go Group (G)** Creates and starts this process and all other processes in the multiprocess program (program group). If the process already exists and is stopped or at a breakpoint, this command resumes its execution and the execution of all processes in the program group.

Note that issuing **Go Group** on a process that's already running starts the other members of the program group.

**Go Thread (^G)** Starts this thread. Disabled if asynchronous run is not available (see “Synchronous vs. Asynchronous Run” on page 90).

For a single-process program, **Go Process** and **Go Group** are equivalent. For a single-threaded process, **Go Thread** and **Go Process** are equivalent.

If you want to change global variables after a process is created, but before it runs, you can use one of the following commands from the process window:

**Arguments/Create/Signal \* Create Process (without starting it) (C)**  
Creates the process image, but does not start the process.

**Go/Halt/Step/Next \* Step (source line) (s)**  
Creates the process and runs it to the first line of the **main()** routine.

---

## Examining Process Groups

When you debug multiprocess programs, TotalView places processes in process groups for convenience. TotalView's process groups are not related to UNIX process groups or PVM groups in any way.

## Types of Process Groups

When you start a multiprocess program, the debugger adds each process to a process group as the process starts. The debugger groups the processes depending on the type of system call (**fork()** or **execve()**) that created or changed the processes. There are two different types of process groups:

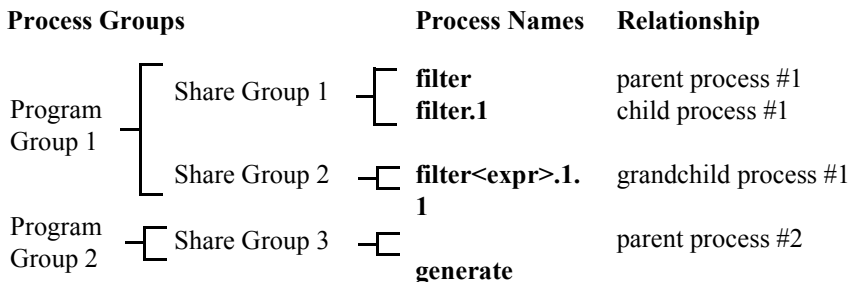
- |                      |   |
|----------------------|---|
| <b>Program Group</b> | Includes the parent process and <i>all</i> related processes. A program group includes children that were forked (processes that share the same source code as the parent) and children that were forked but with a subsequent call to <b>execve()</b> (processes that do <i>not</i> share the same source code as the parent). |
| <b>Share Group</b>   | Includes only the related processes that share the same source code.  |

In general, if you are debugging a multiprocess program, the program group and share group differ only if the program has some children that are forked with a subsequent call to **execve()**.

Commands that contain the term *group* (such as **Go Group**) refer to all members of the program group. The term *relatives* generally refers to the program group as well.

The debugger names the processes in program groups and share groups according to the name of the source program. The parent process is named after the source program. Child processes that were forked have the same name as the parent, but with a numerical suffix (*.n*). Child processes that call **execve()** after they were forked have the parent's name, the name of the new executable (in angle brackets), and a numerical suffix.

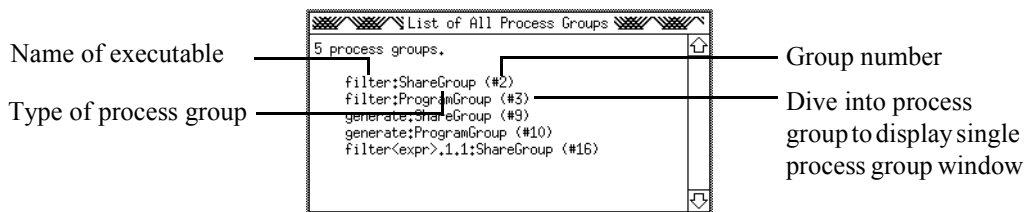
For example, if the **generate** process forks no children and the **filter** process forks a child process, which makes a subsequent call to **execve()** to execute the **expr** program, the debugger names and groups the processes as shown in Figure 29.



**Figure 29.** Example of Program Groups and Share Groups

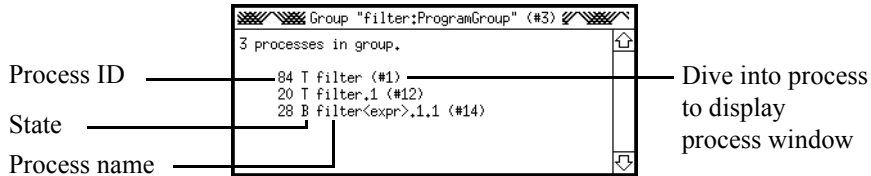
## Displaying Process Groups

The root window displays the names of individual processes in multiprocess programs, but not the process groups. To display a list of process groups, select the **Show All Process Groups** command from the root window. The process groups window appears, as shown in Figure 30.



**Figure 30.** Process Groups Window

If you dive into any process group listed in the window, a single process group window appears, as shown in Figure 31. By diving into any process listed in the window, you display the process window for the process. (You can also dive into the process listed in the root window to display its process window.)



**Figure 31.** Single Process Group Window

## Changing Program Groups

In most situations, TotalView places a process in the correct program group, so you do not normally need to change the program group of a process.

If necessary, however, you can move processes into different program groups. When you move a process into a different program group, TotalView automatically places it in the correct share group. The advantage of moving a process into a different program group is that members of the same program group can start and stop on a breakpoint at the same time. Furthermore, members of the same share group share the same set of action points.

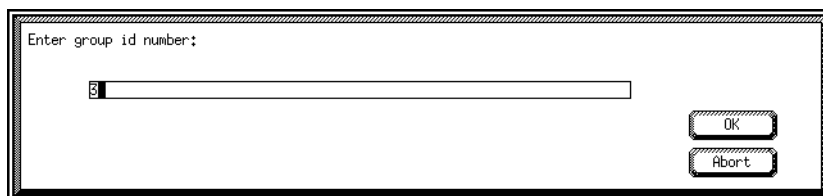
---

**Note:** TotalView uses the name of the executable to determine the share group to which the program belongs. TotalView does not examine the program in any way to see if it is identical to another program with the same name: TotalView assumes the programs are identical based on their names being identical. Also, TotalView does not expand a program's full pathname, so if one instance of a program is named with the full pathname (**./foo**), and another is named with the leaf name (**foo**), the programs will be placed into different share groups.

---

To move a process into a different program group:

1. From the root window, select **Show All Process Groups**. The process groups window appears.
2. Make note of the group ID number for the *program group* into which you're moving the process. This number is displayed in parentheses.
3. From the process window for the process to be moved, display the **Arguments/Create/Signal** submenu, and select **Set Process Program Group**. A dialog box appears, as shown in Figure 32.



**Figure 32.** Dialog Box for Changing Process Groups

4. Enter the group ID number into the dialog box and press Return.

## Finding Active Processes

Although a well-balanced multiprocess program distributes work evenly among processes, this situation does not always occur in practice. In some multiprocess programs, most of the active processes may be waiting for work. In this situation, it's tedious to look through the entire group to find the processes that are doing work. Instead, you can use the **Find Interesting Relative** command to find them quickly.

When you display the **Current/Update/Relatives** submenu and select the **Find Interesting Relative** command from the process window:

- A process group window appears, listing the processes in decreasing order of interest.
- A process window appears for the most interesting process in the group (if it does not already have a process window open).

To see additional process windows for processes in decreasing order of interest, select the **Find Interesting Relative** command again, or dive into the processes listed in the process group window.



TotalView uses the following criteria to determine the order of interest:

- Running processes are more interesting than stopped processes.
- Processes with threads at breakpoints are more interesting than those that are stopped at arbitrary locations.
- Processes with threads with deep (larger) stacks are more interesting than processes with shallow (smaller) stacks.
- Processes with threads with unusual PCs are more interesting than processes with threads with identical PCs. (The debugger examines all the threads and produces a histogram of their PCs to determine this.)

---

## Setting a Breakpoint

You can set breakpoints in your program by selecting the boxed line numbers in the source code pane of the process window. A boxed line number indicates that the line generates executable code. A STOP icon over the line number indicates that a breakpoint is set on that line. To clear a breakpoint, select the STOP icon.

When your program reaches a breakpoint during execution, it stops. You can then resume execution in a number of different ways:

- Single step
- Step over a function
- Run to a selected line
- Run to the completion of the current function
- Continue with a specific signal

These techniques are described in the sections that follow.

TotalView provides additional features for working with breakpoints and also provides evaluation points (conditional breakpoints) and event points. For more information, refer to “Setting Action Points” on page 133.

---

## Single Stepping

TotalView implements a variety of single stepping commands. The debugger allows you to:

- Single step at the source line level or the machine instruction level
- Single step over or into function calls
- Run to a selected line, which is a temporary breakpoint facility
- Run until a function call returns

These commands are TotalView's *high-level* single stepping commands.

---

**Note:** Because the debugger implements the high-level single stepping commands by using temporary breakpoints. It is possible that a running thread other than the one being single stepped will hit the temporary breakpoint causing it to stop.

---

Whenever possible, TotalView tries to single step a single thread, but depending on the capabilities of the target operating system, the single stepping command will behave differently (see “Synchronous vs. Asynchronous Run” on page 90).

- On asynchronous run systems  
On target operating systems that support starting and stopping a single thread in a multithreaded program, the TotalView single stepping commands will single step a single thread.
- On synchronous run systems  
On target operating systems that do not support starting and stopping a single thread in a multithreaded program, TotalView must implement single stepping commands by running the entire process, which may lead to surprising results.

Since thread scheduling is performed by the operating system, threads other than the one you are trying to single step will run, and possibly encounter a breakpoint, signal or exception before

the thread you are trying to single step executes any instructions. When this happens, TotalView will switch focus to the thread that encountered the condition.

## Stepping Into Functions Calls

To execute a single statement or instruction, and possibly step into a function call, go to the process window and select one of the following commands from the **Go/Halt/Step/Next** submenu:

<b>Step (source line) (s)</b>	Executes a single source statement. If the thread is executing in a function that has no source line information, TotalView performs a <b>Step (instruction)</b> instead.
<b>Step (instruction) (i)</b>	Executes a single machine instruction.

---

**Note:** To cancel a single step in progress, position the mouse pointer in the process window and press CTRL-C.

---

If your program reaches a breakpoint while executing a single step, TotalView cancels the single step and your program stops at the breakpoint.

---

**Note:** You cannot debug inside template functions. When you step into a template function, TotalView usually displays Assembler code, although in some cases, it may display source code.

---

## Stepping Over Function Calls

If you single step through a statement that contains a function call, you step into the function, providing there is debugging information available for it. If desired, you can single step over a function call. When you step over a function, TotalView stops execution when the thread returns from the function and reaches the source line or instruction after the function call. To do this, go to the process window and select one of the following commands from the **Go/Halt/Step/Next** submenu.

<b>Next (source line) (n)</b>	Executes a single source statement and steps over any function calls. If the thread is executing in a function that has no source line information, TotalView performs a <b>Next (instruction)</b> instead.
<b>Next (instruction) (x)</b>	Executes a single machine instruction and steps over function calls.

---

**Note:** To cancel a single step in progress, position the mouse pointer in the process window and press CTRL-C.

---

If your program reaches a breakpoint while stepping over a function, TotalView cancels the operation and your program stops at the breakpoint.

## Executing to a Selected Line

You don't have to set a breakpoint to stop execution on a specific line. TotalView provides a convenient way for you to execute a thread to a selected line. To do so, complete these steps from the process window:

1. In the source code pane, select the source line or instruction on which you want the program to stop execution.
2. Display the **Go/Halt/Step/Next** submenu and select the **Run to Selection (r)** command.

TotalView executes the thread and stops when it reaches the selected line within the current function.

You can also run to a selected line in a nested stack frame. To do so:

1. Select a nested frame in the stack trace pane.
2. Select a source line or instruction within the function.
3. Issue the **Run to Selection (r)** command.

TotalView executes the thread until it reaches the selected line in the selected stack frame.

If your program calls recursive functions, you can select a nested stack frame in the stack trace pane to tailor execution even more. In this situation, TotalView uses the frame pointer (FP) of the selected stack frame and the selected source line or instruction to determine when to stop execution. When your program reaches the selected line during execution, TotalView compares the value of the selected FP to the value of the current FP in the following way:

- If the value of the current FP is deeper (more deeply nested) than the value of the selected FP, TotalView automatically continues your program.
- If the value of the current FP is equal or shallower (less deeply nested) than the value of the selected FP, TotalView stops your program.

---

**Note:** To cancel a run to selection that is in progress, position the mouse pointer in the process window and press CTRL-C.

---

If your program reaches a breakpoint while executing to a selected line, TotalView cancels the operation and your program stops at the breakpoint.

## Executing to the Completion of a Function

To finish executing the current function in a thread, go to the process window, display the **Go/Halt/Step/Next** submenu, and select the **Return (out of function) (o)** command. TotalView executes the thread, completes execution of the current function, and returns to the instruction after the one that called the function.

You can also return out of several functions at once. To do so, select a nested stack frame in the stack trace pane, and select the **Return (out of function) (o)** command. TotalView executes the thread until it returns *to* that function.

If your program calls recursive functions or mutually recursive functions, you can select a nested stack frame in the stack trace pane to tailor completion of the function even more. In this situation, TotalView uses the frame pointer (FP) of the selected stack frame and

the selected source line or instruction to determine when to stop execution. When your program reaches the selected line, TotalView compares the value of the selected FP with the value of the current FP in the following way:

- If the value of the current FP is deeper (more deeply nested) than the value of the selected FP, TotalView automatically continues your program.
- If the value of the current FP is equal or shallower (less deeply nested) than the value of the selected FP, TotalView stops your program.

---

**Note:** To cancel a return that is in progress, position the mouse pointer in the process window and press CTRL-C.

---

If your program reaches a breakpoint while executing to the completion of a function, TotalView cancels the operation and your program stops at the breakpoint.

---

## Continuing with a Specific Signal

Continuing execution of your program with a specific signal can be useful if your program contains a signal handler. To do so, complete these steps from the process window:

1. Display the **Go/Halt/Step/Next** submenu and select the **Set Continuation Signal** command.
2. In the dialog box, enter the name (such as SIGINT) or number (such as 2) of the signal to be sent to the thread.

3. Select OK.

---

**Note:** The continuation signal is set for the thread you are focused on in the process window. If the target operating system supports the multithreaded signal delivery capability (see “Allows Multithreaded Signal Delivery” on page 91), you may set a separate continuation signal for each thread. If this capability is not supported, then this command will clear any continuation signal you specified for other threads in the process.

---

4. Continue execution of your program with the **Go**, **Step**, **Next**, or **Detach from Process** command.

TotalView continues the thread(s) with the specified signal(s).

---

## Setting the Program Counter

You might find it useful to resume the execution of a thread at some statement other than the one where it stopped. To do this, you reset the value of the Program Counter (PC). For example, you might want to skip over some code, execute some code again after changing certain variables, or restart a thread that’s in an error state.

Setting the program counter can be crucial when you want to restart a thread that’s in an error state. Although the PC icon (?) in the tag field points to the source statement that caused the error, the PC actually points to the *failed machine instruction* within the source statement. You need to explicitly reset the PC to the beginning of the source statement. (You can verify the actual location of the PC before and after resetting it by displaying it in the stack frame pane or displaying interleaved source and Assembler code in the source code pane.)

In TotalView, you can set the PC of a stopped thread to a selected source line, a selected instruction, or an absolute value (in hexadecimal). 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 sufficient.

To set the PC to a selected line:

1. If you need to set the PC to a location somewhere *within* a line of source code, display the Assembler code. To do so, display the **Display/Directory/Edit** submenu and select the **Interleave Display Mode (M-i)** command.
2. Select the source line or instruction in the source code pane. TotalView highlights the line in reverse video.
3. Display the **Go/Halt/Step/Next** submenu and select the **Set PC to Selection... (p)** command. TotalView asks for confirmation, resets the PC, and moves the PC icon (?) to the selected line.

When you select a line and ask the debugger to set the PC to that line, TotalView attempts to force the thread to continue execution at that statement *in the currently selected stack frame*. If the currently selected stack frame is not the top stack frame, the debugger asks your permission to unwind the stack:

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

If you select **Yes**, the debugger *discards* all deeper stack frames (that is, all stack frames that are more deeply nested than the selected stack frame) and resets the machine registers to the proper value for the selected frame. If you select **No**, the debugger sets the PC to the selected line, but it leaves the stack and registers in their current state. Since you cannot assume that the stack and registers have correct values, selecting **No** can cause problems. We recommend that you select **Yes**.

In general, we only recommend setting the PC to an absolute address for very advanced users. If you need to do this, make sure you have the correct address; *no* verification is done by TotalView.

To set the PC to an absolute address:

1. Display the **Go/Halt/Step/Next** submenu and select the **Set PC to Absolute Value...** command. A dialog box prompts you for a hexadecimal address.
2. Enter the hexadecimal address into the dialog box.
3. Select **OK**. The debugger resets the PC and moves the PC icon (?) to the line containing the absolute address.



---

# Stopping Processes and Threads

To stop a process or a thread, go to the process window and select one of the following commands from the **Go/Halt/Step/Next** submenu:

**Halt Process (h)**

Stops the process.

**Halt Group (H)**

Stops the process and all related processes.

Note that issuing **Halt Group** on a process that's already stopped stops the other members of the program group.

**Halt Thread (^H)**

Stops the thread.

Note that this command is disabled if asynchronous stop is not available (see “Synchronous vs. Asynchronous Stop” on page 90).

When the TotalView debugger stops a process, it updates the process window and all related windows. When you start the process again, execution continues from the point where you stopped the process.

---

**Note:** You can force the process window to update the process information using the **Update Process Info (u)** command from the **Current/Update/Relatives** submenu *without* stopping the process. TotalView will flush its internal process data cache and temporarily stop the process and reread the thread registers and memory. This allows you to quickly refresh your view of a process.

---

---

## Deleting Processes

To delete a process, display the **Arguments/Create/Signal** submenu and select the **Delete Process (^Z)** command. If the process is part of a multiprocess program, the debugger deletes all related processes as well. The next time you start the process, the debugger creates and starts a fresh process.

---

## Restarting Processes

If you need to restart a process from scratch and the process has stopped but has not exited:

1. Display the **Arguments/Create/Signal** submenu and select the **Delete Process (^Z)** command. See “Deleting Processes” on page 106.
2. Start the process. See “Starting Processes and Threads” on page 91.

## ***CHAPTER 5:***

# **Examining and Changing Data**

This chapter explains how to examine and change data as you debug your program. You'll learn how to:

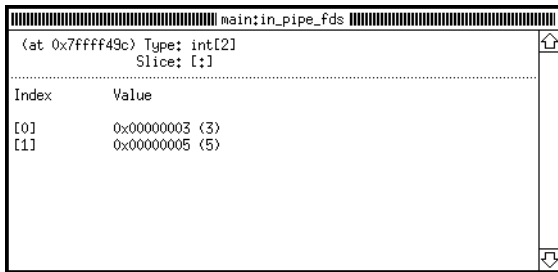
- Display variable windows
- Dive into variables
- Change the values of variables
- Change the data type of variables
- Display array slices
- Change the address of variables
- Display machine instructions
- Display the cache status of variables

# Displaying Variable Windows

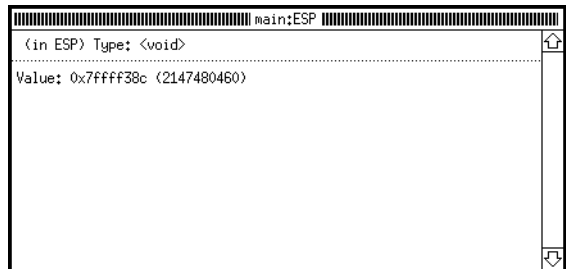
You can display variable windows for local variables, registers, global variables, areas of memory, and machine instructions.

## Displaying Local Variables and Registers

In the stack frame pane of the process window, you can dive into any parameter, local variable, or register to display a variable window. You can also dive into parameters and local variables in the source code pane. The variable window lists the name, address, data type, and value for the object, as shown in Figure 33.



Local variable



Register

**Figure 33.** Diving into Local Variables and Registers

If you keep the variable windows open while you continue running the process or thread, the debugger updates the information in the windows when the process or thread stops for any reason. Keep in mind, however, that the variable window tracks *addresses*, not variable names. For example, suppose you display a variable window for a local variable and keep the window open until the current routine finishes

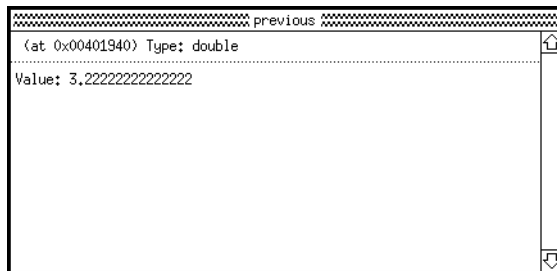
executing. If your program executes the routine again, the local variable may be located at a different address. For local variables and parameters, TotalView recomputes the address based on the value of the new frame pointer (FP).

## Displaying a Global Variable

You can display a global variable in two different ways:

- Diving into the variable in the source code pane.
- Displaying the **Function/File/Variable** submenu and selecting the **Variable... (v)** command. When prompted, enter the name of the variable in the dialog box.

A variable window appears for the global variable, as shown in Figure 34.



**Figure 34.** Variable Window for a Global Variable

---

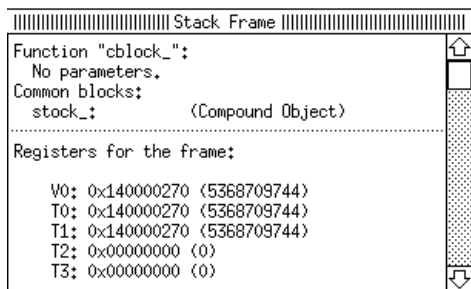
**Note:** If your program has more than one static variable in scope with the same name, TotalView sees each name as a separate entity. If you request to display one of these variables, TotalView picks one at random. To discriminate among these variables, you must display the global variables window and dive into the desired variable. For more information, refer to “Displaying All Global Variables” on page 111.

---

## Displaying Fortran Common Blocks

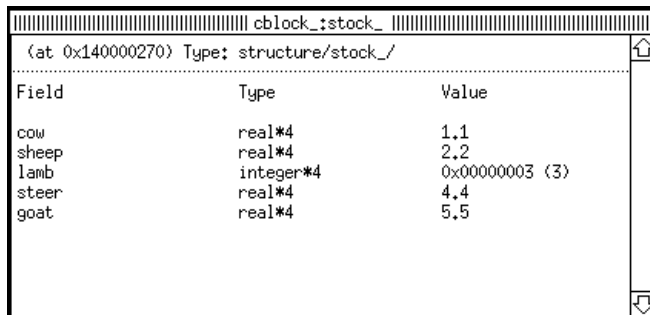
TotalView handles Fortran common blocks in a manner consistent with the semantics of Fortran. The names of common block members have function scope, not global scope.

For each common block that is defined within the scope of a subroutine or function, TotalView creates an entry in that function's common block list. The stack frame pane in the process window displays the name of each common block for a function, as shown in Figure 35.



**Figure 35.** Common Block List in Stack Frame Pane

TotalView gives the common block a structure data type in which each of the common block members are fields in the structure. If you dive on a common block name in the stack frame pane, TotalView displays the entire common block in a variable window, as shown in Figure 36.

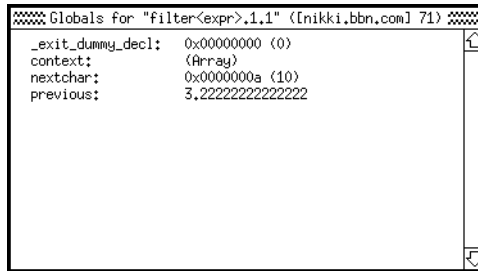


**Figure 36.** Variable Window for Elements of a Common Block

If you dive on a common block member name, TotalView searches all the common blocks for a matching member name and displays the member in a variable window.

## Displaying All Global Variables

For convenience, you can display *all* global variables used by the current process. To do so, display the **Function/File/Variable** submenu and select the **Global Variables Window (V)** command. A global variables window appears listing the name and value of every global variable used by the process, as shown in Figure 37.



**Figure 37.** Global Variables Window

If desired, you can display a variable window for any global variable listed in the global variables window. To do so, either:

- Dive into the variable in the global variables window.
- Select the **Variable... (v)** command from the global variables window, and enter the name of the variable in the dialog box.

## Displaying Areas of Memory

You can display areas of memory in hexadecimal and decimal. To do so, display the **Function/File/Variable** submenu and select the **Variable... (v)** command. When prompted, enter one of the following in the dialog box:

- A hexadecimal address

When you enter a single address, the debugger displays the word of data stored at that address.

- A range of hexadecimal addresses

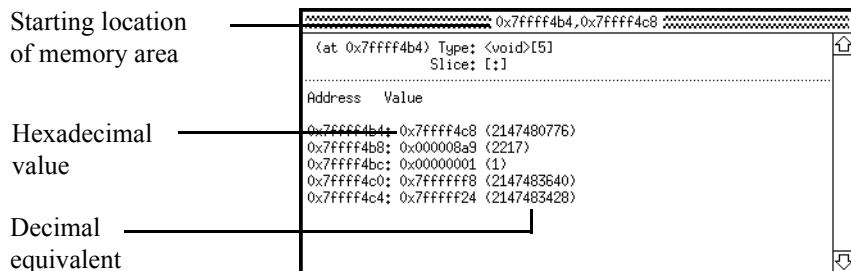
When you enter a range of addresses, the debugger displays the data (in word increments) between the first and last address. To enter a range of addresses, enter the first address, a comma (,), and the last address.

---

**Note:** All hexadecimal addresses must have the “0x” prefix.

---

The variable window for an area of memory, shown in Figure 38, displays the address and contents of each word increment.



**Figure 38.** Variable Window for Area of Memory

## Displaying Large Arrays

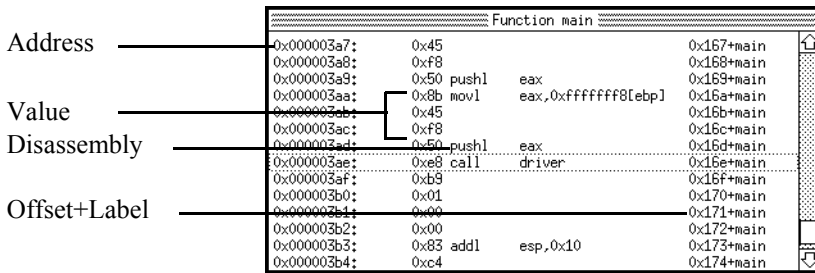
TotalView can quickly display very large arrays in variable windows. If an array overlaps nonexistent memory, the initial portion of the array is correctly formatted. The array elements that fall within nonexistent memory, have “Bad Address” displayed in the subscript.



## Displaying Machine Instructions

You can display the machine instructions for entire routines in the following ways:

- Dive into the address of an Assembler instruction in the source code pane (such as **main+0x10** or **0x60**). A variable window displays the instructions for the entire function and highlights the instruction that you dived into.
- Dive into the PC in the stack frame pane. A variable window lists the instructions for the entire function containing the PC, and highlights the instruction to which the PC points, as shown in Figure 39.
- Cast a variable to type **<code>**, as described in “Changing Type Strings to Display Machine Instructions” on page 130.



**Figure 39.** Variable Window with Machine Instructions

## Closing Variable Windows

When you are finished analyzing the information in a variable window, you can issue the **Close Window (q)** command (to close the window) or the **Close All Similar Windows (Q)** command (to close *all* variable windows).

# Diving in Variable Windows

If the variable you display in a variable window is a pointer, structure, or array, you can dive into the contents listed in the variable window. This additional dive is called a *nested dive*. When you perform a nested dive, the variable window replaces the original information with information about the current variable. With nested dives, the original variable window is known as the *base window*.

Figure 40 shows the results of diving into a variable in the stack frame pane of `main()` in the process window. In this example, we dove into a variable named `node` with a type of `node_t*`, which is a pointer. The first variable window (base window) in the figure displays the value of `node`.

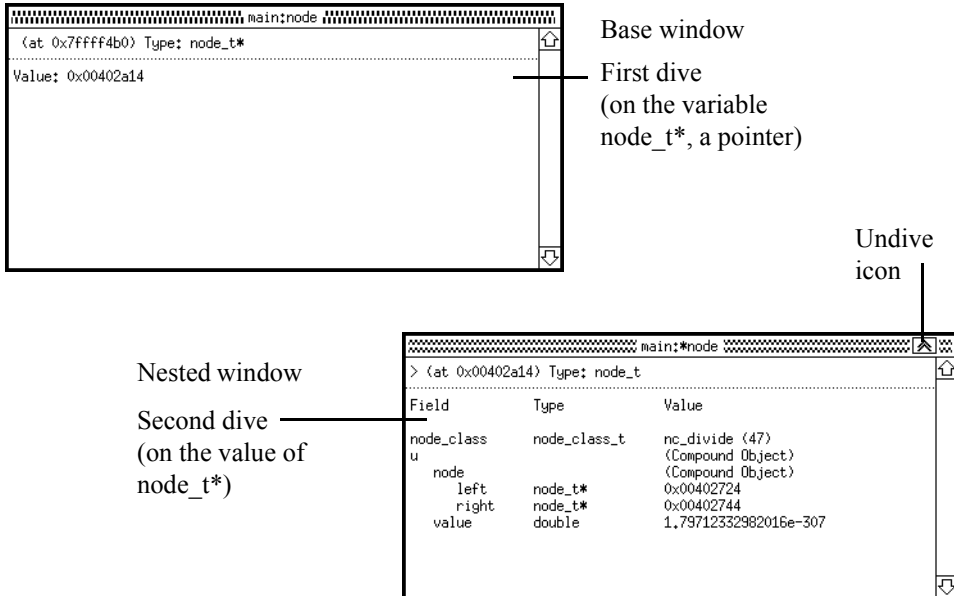


Figure 40. Nested Dives

Then, we dove on the value shown in the base window, and a nested dive window replaced it. The nested dive window is shown at the bottom of the figure; it shows the structure referenced by the **node\_t\*** pointer.

Also, notice that the number of right angle brackets (>) in the upper left hand corner indicates the number of nested dives that were performed in the window. TotalView maintains each dive as part of a dive stack.

You can manipulate variable windows and nested dive windows in the following ways:

- To “undive” from a nested dive, you click the Dive mouse button on the undive icon, and the previous contents of the variable window appears.
- If you have performed several nested dives and want to create a new base window, select the **New Base Window** command from the variable window.
- If you dive into a variable that already has a variable window open, the variable window pops to the surface. If you want a duplicate variable window open, hold down the Shift key when you dive on the variable.
- If you select the **Duplicate Window** command from the variable window, a new variable window appears that is a duplicate of the current variable window except that it has an empty dive stack.

---

## Changing the Values of Variables

You can change the value of any variable or the contents of any memory location by completing these steps in the variable window:

1. Select the value and use the field editor to change the value as desired.

You can type an expression as the value, including logical operators, if desired. For example, you can enter  $1024*1024$ .

2. Press Return to confirm your changes.

You can also edit the value of variables directly from the stack frame pane by selecting them.

---

**Note:** You cannot change the value of bit fields directly; however, you can use the expression window to assign a value to a bit field. See “Evaluating Expressions” on page 150.

---

---

## Changing the Data Type of Variables

The data type that you declared for the variable determines its *format* and *size* (amount of memory) in the variable window. For example, in C, if you declare an **int** variable, the debugger displays the variable as an integer.

You can change the way data is displayed in the variable window by editing the data type. TotalView assigns type strings to all data types, and in most cases, they are identical to their programming language counterparts.

- When displaying a C variable, TotalView type strings are identical to C type representations, except for pointers to arrays. By default, TotalView uses a simpler syntax for pointers to arrays.
- When displaying a Fortran variable, TotalView type strings are identical to Fortran type representations for most data types, including INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER.

To change a type string in a variable window:

Using the field editor, edit the type string in the type field for the window. If the window contains a structure with a list of fields, you can edit the type strings of the fields listed in the window.

---

**Note:** When you edit a type string, the TotalView debugger changes how it displays the variable in the current variable window, but other windows listing the variable remain the same.

---

## How TotalView Displays C Data Types

TotalView's syntax is identical to C cast syntax for all data types except pointers to arrays. Thus, you use C cast syntax for **int**, **unsigned**, **short**, **float**, **double**, **union**, and all named **struct** types.

You read TotalView type strings from right to left. For example, `<string>*[20]*` is a pointer to an array of 20 pointers to `<string>`.

Table 11 shows some common type strings.

**Table 11.** Common Type Strings

Type String	Meaning
<code>int</code>	Integer
<code>int*</code>	Pointer to integer
<code>int[10]</code>	Array of 10 integers
<code>&lt;string&gt;</code>	Null-terminated character string
<code>&lt;string&gt;**</code>	Pointer to a pointer to a null-terminated character string.
<code>&lt;string&gt;*[20]*</code>	Pointer to an array of 20 pointers to null-terminated strings.

---

The following sections comment on some of the more complex type strings.

## Pointers to Arrays

Suppose you declared a variable `vbl` as a pointer to an array of 23 pointers to an array of 12 objects of type `mytype_t`. To *declare* the variable in your C program, you use the syntax:

```
mytype_t ((*vbl)[23])[12];
```

To cast **vbl** to the same type in your C program:

```
(mytype_t ((*)[23])[12])vbl
```

TotalView's type string syntax for **vbl** would be:

```
mytype_t[12]*[23]*
```

## Arrays

Array type names can include a lower and upper bound separated by a colon.

By default, the lower bound for a C or C++ array is 0, and the lower bound for a Fortran array is 1. In the following example, an array of integers is declared in C and then in Fortran:

```
int a[10]
integer a(10)
```

In the C example, the elements of the array range from **a[0]** to **a[9]**, while in the Fortran example, the elements of the array range from **a(1)** to **a(10)**.

When the lower bound for an array dimension is the default for the language, TotalView displays only the extent (that is, the number of elements) of the dimension. Consider the following array declaration in Fortran:

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

Since both dimensions of the array use the default lower bound for Fortran (1), TotalView displays the data type of the array using only the extent of each dimension, as follows:

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

In the case where an array declaration does not use the default lower bound, TotalView displays both the lower bound and upper bound for each dimension of the array. For example, in Fortran, an array of integers with the first dimension ranging from -1 to 5 and the second dimension ranging from 2 to 10 is declared as follows:

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

TotalView displays the following data type for this Fortran array:

```
integer(-1:5,2:10)
```

When you edit a dimension of an array in TotalView, you can enter just the extent (if using the default lower bound) or both the lower and upper bounds separated by a colon.

If desired, you can display a subsection of an array. Refer to “Displaying Array Slices” on page 125 for further information.

## If You Prefer C Cast Syntax

If desired, you can always enter C cast syntax *verbatim* in the type field for any type, and the debugger will understand it. In addition, the debugger can display C cast syntax permanently if you set an X Window Resource. See “totalview\*cTypeStrings: {true | false}” on page 165 for further information.

## Typedefs

The debugger recognizes the names defined with **typedef**, but displays the *definition* of such a type (that is, the base data type), rather than the name. For example, if you declared the following:

```
typedef double *dptr_t;  
dptr_t p_vbl;
```

The debugger displays the type string for **p\_vbl** as **double\***, not as **dptr\_t**.

## Structures

For structures, the debugger treats the string **struct** as a keyword. You can type **struct** in as part of the type string, but it is optional. If you have a structure and another data type with the same name, you must include **struct** with the name of the structure so the debugger can distinguish between the two data types.

If you name a structure using **typedef**, the debugger uses the **typedef** name as the type string. Otherwise, the debugger uses the structure tag for the **struct**.

For example, consider the structure definition:

```
typedef struct mystruc_struct {
    int field_1;
    int field_2;
} mystruc_type;
```

The debugger displays **mystruc\_type** as the type string for **struct mystruc\_struct**.

The debugger does not understand actual structure definitions in the type string. For example, the debugger does not understand the type string **struct {int a; int b;}**.

## Unions

The debugger displays a union as it does a structure. Even though the fields of a union are overlaid in storage, the debugger displays them on separate lines in the variable window.

---

**Note:** When the TotalView debugger displays some complex arrays and structures, it displays the (Compound Object) or (Array) type strings in the variable window. Editing the (Compound Object) or (Array) type strings might yield undesirable results. We do not recommend editing these type strings.

---



## Built-In Type Strings

TotalView provides a number of predefined types. These types are enclosed in angle brackets to avoid conflict with types already defined in the language. You can use these built-in types anywhere a user-defined type can be used, such as in an expression. These types are also useful when debugging executables with no debugging symbol table information. Table 12 lists the built-in types.

**Table 12.** Built-In Type Strings

Type String	Language	Size	Meaning
<string>	C	<b>char</b>	Array of characters
<void>	C	<b>long</b>	Area of memory
<code>	C	<b>parcel</b> <sup>1</sup>	Machine instructions
<address>	C	<b>void*</b>	Void pointer (address)
<char>	C	<b>char</b>	Character
<short>	C	<b>short</b>	Short integer
<int>	C	<b>int</b>	Integer
<long>	C	<b>long</b>	Long integer
<float>	C	<b>float</b>	Single-precision floating-point number
<double>	C	<b>double</b>	Double-precision floating-point number
<extended>	C	<b>long double</b>	Extended-precision floating-point number <sup>2</sup>
<character>	F77	<b>character</b>	Character

**Table 12.** Built-In Type Strings (Continued)

Type String	Language	Size	Meaning
<integer>	F77	<b>integer</b>	Integer
<integer*1>	F77	<b>integer*1</b>	One-byte integer
<integer*2>	F77	<b>integer*2</b>	Two-byte integer
<integer*4>	F77	<b>integer*4</b>	Four-byte integer
<logical>	F77	<b>logical</b>	Logical
<logical*1>	F77	<b>logical*1</b>	One-byte logical
<logical*2>	F77	<b>logical*2</b>	Two-byte logical
<logical*4>	F77	<b>logical*4</b>	Four-byte logical
<real>	F77	<b>real</b>	Single-precision floating-point number
<real*4>	F77	<b>real*4</b>	Four-byte floating-point number
<real*8>	F77	<b>real*8</b>	Eight-byte floating-point number
<double precision>	F77	<b>double precision</b>	Double-precision floating-point number
<complex>	F77	<b>complex</b>	Single-precision floating-point complex number <sup>3</sup>
<complex*8>	F77	<b>complex*8</b>	real*4-precision floating-point complex number <sup>4</sup>
<complex*16>	F77	<b>complex*16</b>	real*8-precision floating-point complex number <sup>5</sup>

1. A parcel is defined to be the number of bytes required to hold the shortest instruction for the target architecture.

2. Extended-precision numbers need to be supported by target architecture.

3. **complex** types contain a Real\_Part and an Imaginary\_Part, which are both of type **real**.

4. **complex\*8** types contain a Real\_Part and an Imaginary\_Part, which are both of type **real\*4**.

5. **complex\*16** types contain a Real\_Part and an Imaginary\_Part, which are both of type **real\*8**.

The following sections give more detail about several of the built-in types.

### **Character arrays (`<string>` data type)**

If you declare a character array as `char vbl[n]`, the debugger automatically changes the type to `<string>[n]`, a null-terminated, quoted string with a maximum length of  $n$ . Thus, by default, the array is displayed as a quoted string of  $n$  characters, terminated by a null character. Similarly, the debugger changes `char*` declarations to `<string>*` (a pointer to a null-terminated string).

Since many character arrays in C are indeed strings, the debugger's `<string>` type string can be very convenient. If, however, you intended the `char` data type to be a pointer to a single character or an *array* of characters, you can edit the `<string>` back to a `char` (or `char[n]`) to display the variable as you declared it.

### **Areas of memory (`<void>` data type)**

The debugger uses the `<void>` type string for data of an unknown type, such as the data contained in registers or in an arbitrary block of memory. The `<void>` type string is similar to the `int` in the C language.

If you dive into registers or display an area of memory, the debugger lists the contents as a `<void>` data type. Further, if you display an array of `<void>` variables, the index for each object in the array is the *address*, not an integer. This address can be useful when you display large areas of memory.

If desired, you can change a `<void>` type string to any other legal type. Likewise, you can change any legal type into a `<void>` to see the variable in hexadecimal.

### **Instructions (`<code>` data type)**

The debugger uses the `<code>` data type to display the contents of a location as machine instructions. Thus, to look at disassembled code that is stored at any location, dive on the location and change the type string to `<code>`. To specify a block of locations, use `<code>[n]`, where  $n$  is the number of locations to be displayed.

## Opaque Type Definitions

An opaque type is a data type that is not fully specified. For example the following C declaration defines **p** with a type of pointer to opaque **struct foo**:

```
struct foo;  
struct foo *p;
```

When TotalView encounters type information that indicates a type is opaque, it enters the type into the type table with **<opaque>** appended to the type name. With the previous example, TotalView enters the following type name in the type table:

```
struct foo <opaque>
```

If the type is opaque and another module defines the type fully, then you can delete **<opaque>** from the data type to have TotalView find the real definition for the type. On the platforms where TotalView uses lazy reading of the symbol table, you must force TotalView to read the symbols from the module containing the full type definition of the opaque type. Use the Function or File command to force TotalView to read the symbols, as described in “Finding the Source Code for Functions” on page 86.

## Example: Displaying the argv Array

Typically, you declare **argv**, the second argument passed to your **main()** routine, as either a **char \*\*argv** or **char \*argv[ ]**. Since these declarations are equivalent (a pointer to one or more pointers to characters), the debugger converts both to the type **<string>\*\*** (a pointer to one or more pointers to null-terminated strings).

Suppose **argv** points to an array of 20 pointers to character strings. To edit the type string (**<string>\*\***) so that the debugger displays the array of 20 pointers:

1. Select the type string for **argv**.
2. Edit the type string using the field editor commands. Change it to:

```
<string>*[20]*
```

3. To display the array, dive into the value field for **argv**.

## Example: Displaying Declared Arrays

You can display declared arrays in the same way you display local and global variables. In the stack frame or source code pane, dive into the declared array. A variable window displays the elements of the array.

## Example: Displaying Allocated Arrays

C code uses pointers for dynamically allocated arrays. For example, consider the following:

```
int *p = malloc(sizeof(int) * 20);
```

In this example, TotalView doesn't know that **p** actually points to an array of integers. To display the array:

1. Dive on the variable of type **int\***.
2. Change its type to **int[20]\***.
3. Dive on the value of the pointer to display the array of 20 integers.

---

## Displaying Array Slices

TotalView can display subsections of arrays, which are called *slices*. Every TotalView variable window that displays an array contains an additional **Slice** field. You can edit this field to view subsections of your array. Initially, the field contains either **[:]** for C arrays or **(:)** for Fortran arrays, which displays the entire array.

## Slice Descriptions

A slice description consists of the following:

```
lower_bound:upper_bound:stride
```

This description specifies that TotalView should display every *stride* element of the array, starting at the *lower\_bound* and continuing through the *upper\_bound*, inclusive.

For example, if you specified a slice of **[0:9:9]** for a 10-element C array, TotalView displays the first element and last element (the 9th element beyond the lower bound).

In the case where the stride of a slice is **1**, you can specify the slice with just two numbers separated by colons: the lower and upper bounds. For example, to display a slice of **[0:9:1]**, you can specify the following:

```
[ 0 : 9 ]
```

The slice **[0:9]** displays array elements 0 through 9, whereas the slice **[4:6]** displays array elements 4 through 6.

If the stride is **1** and the lower and upper bound are the same number, you can specify the slice with just a single number, which indicates both the lower and upper bound. For example, to display a slice of **[9:9:1]**, you can specify the following:

```
[ 9 ]
```

The slice **[9]** displays element 9.

---

**Note:** The *lower\_bound*, *upper\_bound*, and *stride* portions of a slice description must be constant values. Expressions are not supported yet.

---

For multidimensional arrays, you can specify a slice for each dimension using the following syntax:

C and C++	<code>[slice][slice]...</code>
Fortran	<code>(slice,slice,...)</code>

## Strides

You can use the stride of a slice to either skip elements of an array or to invert the order in which elements of an array are displayed.

For example, if you specify a slice of **:::2** for a C or C++ array (with a default lower bound of 0), TotalView displays only the even elements of the array: 0, 2, 4, and so on. However, if you specify this same slice for a Fortran array (with a default lower bound of 1), TotalView

displays only the odd elements of the array: 1, 3, 5, and so on. As an example of skipping elements in a multidimensional array, you can specify a slice of **(::9,::9)** to display the four corners of a 10-element by 10-element Fortran array, as shown in Figure 41.

Index	Value
(1,1)	11
(10,1)	101
(1,10)	20
(10,10)	110

**Figure 41.** Slice Displaying the Four Corners of an Array

To invert the order in which elements are displayed, you can specify a negative number as the stride of a slice. If you specify a slice of **(::-1)**, TotalView begins with the upper bound of the array and displays the array in inverted order. For example, if you specified this slice of **(::-1)** with a Fortran array of **integer(10)**, TotalView displays the following elements:

```
( 10 )
(  9 )
(  8 )
. . .
```

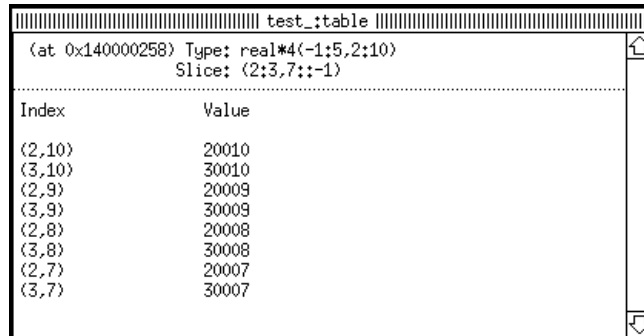
You can use a stride to combine inverse order with skipping elements. For example, if you specify a slice of **(::-2)**, TotalView begins with the upper bound of the array and displays every other element until it reaches the lower bound of the array. For example, if you specify this slice of **(::-2)** with a Fortran array of **integer(10)**, TotalView displays the following elements:

```
( 10 )
(  8 )
```

( 6 )

. . .

You can also combine inverse order and a limited extent to display a small section of a large array. For example, if you specified a slice of **(2:3,7::-1)** with a Fortran array of **real\*4(-1:5,2:10)**, Figure 42 shows the elements that are displayed by TotalView:



Index	Value
(2,10)	20010
(3,10)	30010
(2,9)	20009
(3,9)	30009
(2,8)	20008
(3,8)	30008
(2,7)	20007
(3,7)	30007

**Figure 42.** Fortran Array with Inverse Order and Limited Extent

As you can see in the figure, TotalView only shows in rows 2 and 3 of the array, beginning with column 10 and concluding with column 7.

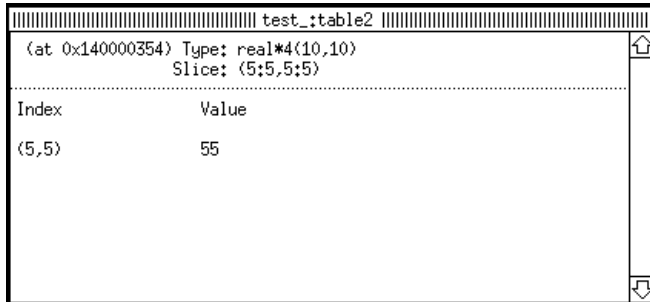
## Using Slices in the Variable Command

When you use the **Variable** command to display a variable window, you can include a slice expression as part of the variable name. Specifically, if you include an array name followed by a set of slice descriptions in the variable dialog box, TotalView initializes the slice field in the variable window to the slice descriptions that you specified.

If you include an array name followed by a list of subscripts in the variable dialog box, TotalView interprets the subscripts as a slice description rather than as a request to display an individual value of the array. As a result, you can display different values of the array by changing the slice expression.



For example, suppose that you have a 10-element by 10-element Fortran array named **table2**, and you want to display element **(5,5)**. Using the **Variable** command, you specify **table2(5,5)** in the dialog box, which sets the initial slice to **(5:5,5:5)**, as shown in Figure 43.



**Figure 43.** Variable Window for table2(5,5)

If desired, you can force TotalView to display a single value in a variable window by enclosing the array name and list of subscripts (that is, the information normally included in a slice expression) inside parentheses, such as **(table2(5,5))**. In this case, the variable window just displays the type and value of the element and does not show its array index.

---

## Changing the Address of Variables

You can edit the *address* of a variable in a variable window. When you edit the address, the variable window shows the contents of the new location.

You can also enter an address expression, such as **0x10b8-0x80**.

---

## Changing Type Strings to Display Machine Instructions

You can display machine instructions in any variable window. To do so:

1. Select the type string at the top of the variable window.
2. Change the type string to be an array of **<code>** data types, where the number of elements, *n*, indicates the number of instructions to be displayed:

**<code>[*n*]**

The debugger displays the contents of the current variable, register, or area of memory, as machine-level instructions.

The variable window (shown in Figure 39 on page 113) lists the following information about each machine instruction:

<b>Address</b>	The machine address of the instruction.
<b>Value</b>	The hexadecimal value stored in the location.
<b>Disassembly</b>	The instruction and operands stored in the location.
<b>Offset+Label</b>	The symbolic address of the location as a hexadecimal offset from a routine name.

You can also edit the value listed in the **value** field for each machine instruction.

---

## Caching of Variables

If your processor provides caching and TotalView supports the display of cache status for your platform, you may see a character in brackets next to the address of the variable, as shown in Figure 44. This character indicates the cache status, such as whether the variable is shared or uncached. For complete information, refer to your platform-specific supplement.



**Figure 44.** Cache Status of a Variable

## **CHAPTER 6:**

# **Setting Action Points**

This chapter explains how to use action points. You can set three different kinds of action points: breakpoints, evaluation points, and event points. When the debugger reaches an action point, it either stops execution (if it is a breakpoint) or executes a code fragment (if it is an evaluation point).

You'll learn how to:

- Set breakpoints
- Set evaluation points
- Set conditional breakpoints
- Patch programs
- Control action points
- Save action points to a file
- Evaluate expressions
- Write code fragments

---

## Setting Breakpoints

The TotalView debugger offers several options for setting breakpoints. You can set source-level breakpoints, machine-level breakpoints, and breakpoints that are shared among all processes in multiprocess programs. You can also control whether or not TotalView stops all processes in a process group when a single member reaches a breakpoint.

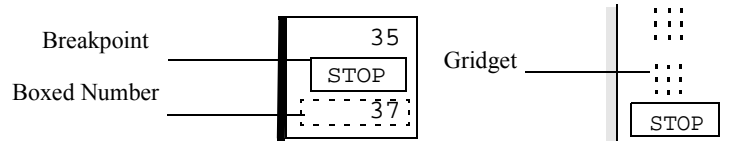
---

**Note:** Breakpoints apply to the entire process, not just to a single thread. Any thread executing in the process could hit the breakpoint, thus causing it to stop. If the operating system uses the synchronous stop model (see “Synchronous vs. Asynchronous Stop” on page 90), all threads in the process will stop.

---

### Source-Level and Machine-Level Breakpoints

Typically, you set and clear breakpoints before you start a process; however you can set a breakpoint while a process is running. If you set the breakpoint while the process is running, TotalView stops the process temporarily to insert the breakpoint and then continues running it. To set a source-level breakpoint, select a boxed line number in the tag field of the process window. A boxed line number indicates that the line generates executable code. A STOP sign appears, as shown in Figure 45. The STOP sign indicates that the breakpoint occurs *before* the source statement is executed.



**Figure 45.** Breakpoint Symbol

To set a machine-level breakpoint, you must first display Assembler code or source interleaved with Assembler. (Refer to “Examining Source and Assembler Code” on page 84 for information.)

Then you select the tag field that is opposite the appropriate instruction. The tag field must contain a gridget, indicating that the line is the beginning of a machine instruction. Since the instruction sets on some platforms support variable-length instructions, you may see multiple lines associated with a single gridget. The stop sign appears, indicating that the breakpoint occurs *before* the instruction is executed.

---

**Note:** When the source pane displays source interleaved with Assembler, source statements are treated as comments. You can set breakpoints on instructions, not source statements. If you set a breakpoint on the first instruction after a source statement, however, you create the equivalent of a source-level breakpoint.

If you set machine-level breakpoints on one or more instructions that are part of a single source line and then display source code in the source pane, TotalView displays an ASM sign on the line number. To see the specific breakpoints, you must display Assembler or Assembler interleaved with source code.

---

After you set all desired breakpoints, you can start the process. When a process reaches a breakpoint, TotalView does the following:

- Suspends the process
- Displays the PC symbol (→) over the stop sign to indicate the PC currently points to the breakpoint
- Displays “at breakpoint” in the title bar of the process window and other windows
- Updates the stack trace panes, stack frame panes, and variable windows.

## Thread Specific Breakpoints

TotalView implements thread specific breakpoints through the TotalView expression system. The expression system has several intrinsic variables that allow a thread to retrieve its thread ID. For example:

```
/* Stop when thread 3 evaluates this
expression. */
if ($tid == 3) $stop;
```

## Breakpoints for Multiple Processes

In multiprocess programs, you can set breakpoints in the parent process and child processes before you start the program and at any time during its execution. To do this, you use the action point options dialog box, as shown in Figure 46. This dialog box provides two checkboxes for process groups:

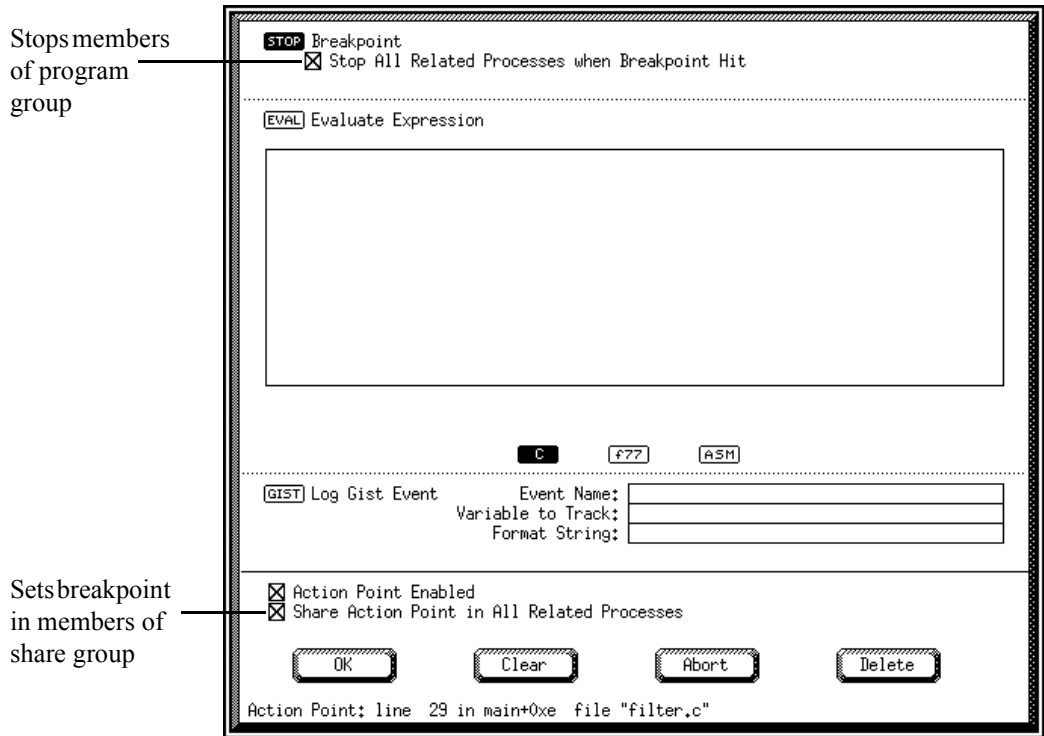
- Stop All Related Processes when Breakpoint Hit

If selected, stops all members of the program group when the breakpoint is reached. Otherwise, only the process that hits the breakpoint stops.



- Share Action Point in All Related Processes

If selected, enables and disables the breakpoint in all members of the share group at the same time. Otherwise, you enable and disable the breakpoint in each share group member individually.



**Figure 46.** Action Point Options Dialog Box

You can control the default setting of these two checkboxes using X Resources. Refer to “totalview\*stopAll: {true | false}” on page 173 and “totalview\*shareActionPointInAllRelatedProcesses: {true | false}” on page 171.

In addition to the controls in the action point options dialog, you can write an expression in the expression box to control the behavior of program group members and share group members. Refer to “Writing Code Fragments” on page 152 for more information.

---

**Note:** You must link with the **dbfork** library to debug programs that call **fork()** and **execve()**. See “Compiling Programs” on page 26.

---

## Processes That Call `fork()`

By default, breakpoints are shared by all processes in the share group, and when any process reaches the breakpoint, TotalView stops all processes in the program group.

To override these defaults:

1. Dive into the tag field to display the action point options dialog box.
2. Deselect these checkboxes: **Stop All Related Processes when Breakpoint Hit** and **Share Action Point in All Related Processes**.
3. Select the **OK** button.

## Processes That Call `execve()`

Breakpoints that are shared by a parent and children with the same executable do not apply to children with different executables. To set the breakpoints for children that call `execve()`:

1. Set the breakpoints and breakpoint options desired in the parent and the children that do *not* call `execve()`.
2. Start the multiprocess program by displaying the **Go/Halt/Step/Next** submenu and selecting the **Go Group (G)** command. When the *first* child calls `execve()`, a dialog box appears with the following message:

Process *name* has called exec (*name*),  
Do you wish to stop it before it enters MAIN?

3. Answer **yes**. TotalView opens a process window for the process. (If you answer **no**, the program executes without allowing you to set breakpoints.)
4. Set the breakpoints desired for the process. Once you set the breakpoints for the first child that uses this executable, the debugger does not prompt you when other children call **execve()** to use this executable. Therefore, if you *don't* want to share the breakpoints among other children using the same executable, dive into the breakpoints, and set the breakpoint options appropriately.
5. Select the **Go Group (G)** command to resume execution.

## Multiprocess Breakpoint Example

The following example program illustrates the different points at which you can set breakpoints in multiprocess programs:

```

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

```

Table 13 shows the results of setting a breakpoint on different lines of the example.

**Table 13.** Setting Breakpoints in Multiprocess Programs

Line Number	Result
1	Stops the parent process before it forks.
2	Stops both the parent and child processes (if the child process was successfully created).
3	Stops the parent process if <b>fork()</b> failed.
5	Stops the child process.
7	Stops the parent process.

---

## Defining Evaluation Points and Conditional Breakpoints

You can define *evaluation points*, points in your program where TotalView evaluates a code fragment. The fragment can include special commands to stop a process and its relatives. Thus, you can use evaluation points to set *conditional breakpoints* of varying complexity. You can also use evaluation points to test potential fixes for your program.

---

**Note:** We recommend that you stop a process before setting an evaluation point. This ensures that the evaluation point is set in a stable context in the program.

---

You can define an evaluation point at any source line that generates executable code (marked with boxed line number in the tag field). If you display Assembler or source interleaved with Assembler in the process window, you can also define evaluation points on machine-level instructions.

As part of defining an evaluation point, you provide the code fragment to be evaluated. You can write the code fragment in C, Fortran, or Assembler.

---

**Note:** Not all platforms support the use of Assembler constructs; see your platform-specific supplement for details.

---

At each evaluation point, the code fragment in the evaluation point is executed *before* the code on that line. Typically, the program then executes the program instruction at which the evaluation point is set. But your code fragment can modify this behavior:

- It can include a branching instruction (such as GOTO in C or Fortran). The instruction can transfer control to a different point in the target program, enabling you to code and test program patches.

- It can contain a **\$stop**, **\$stopall**, **\$count**, or **\$countall** statement. These special TotalView statements define breakpoints and countdown breakpoints within the code fragment. By including them within other statements that you code, you can define conditional breakpoints. For more information on these statements, refer to Table 16, “Built-In Statements That Can Be Used in Expressions,” on page 154.

TotalView evaluates code fragments in the context of the target program. This means that you can refer to program variables and pass control to points in the target program.

---

**Note:** For complete information on what you can include in code fragments, refer to “Writing Code Fragments” on page 152.

Evaluation points modify only the processes that you are debugging. They do not permanently modify the source program or create a permanent patch in the executable. If you save the evaluation points for a program, however, TotalView reapplies them whenever you start a debugging session for that program. To save your evaluation points, refer to “Saving Action Points in a File” on page 150.

---

## Setting an Evaluation Point

To set an evaluation point:

1. Dive into the tag field for an instruction in the process window. TotalView displays the action point options dialog box.
2. Select the **EVAL (Evaluate Expression)** button.
3. Select the button (if it’s not already selected) for the language in which you will code the fragment.
4. Select the evaluation text box and enter the code fragment to be evaluated. Use the field editor commands as required. For information on supported C, Fortran, and Assembler language constructs, refer to “Writing Code Fragments” on page 152.

5. For multiprocess programs, decide whether to share the evaluation point among all processes in the program's share group. By default, the **Share Action Point in All Related Processes** is selected for multiprocess programs, but you can override this by deselecting the checkbox.
6. Select the **OK** button to confirm your changes. If the code fragment has an error, TotalView displays an error message. Otherwise, TotalView processes the code, closes the dialog box, and places an EVAL icon in the tag field.

## Setting Conditional Breakpoints

To set a conditional breakpoint, complete steps 1 to 4 of "Setting an Evaluation Point" on page 141. Here are some examples of conditional breakpoints and the code fragments that you would need to supply in step 4:

- To define a breakpoint that is reached whenever a variable **i** is greater than 20 but less than 25:
 

```
if (i > 20 && i < 25)
    $stop;
```
- To define a breakpoint that is reached every 10th time the **\$count** statement is executed:
 

```
$count 10
```
- To define a breakpoint with a more complex expression, consider this one:
 

```
$count i * 2
```

When the variable **i** equals 4, the process stops the 8th time it executes the **\$count** statement. After the process stops, the expression is reevaluated. If **i** now equals 5, the next stop occurs after the process executes the **\$count** statement 10 more Times New Roman.

Then, complete steps 5 and 6 of "Setting an Evaluation Point" on page 141.

For complete descriptions of the **\$stop** and **\$count** statements, refer to "Built-In Statements" on page 154.

## Patching Programs

You can use expressions in evaluation points to patch your code. Specifically, you can use the **goto** (C) and **GOTO** (Fortran) statements to jump to another point in your program's execution.

You can patch programs in two different ways:

- You can patch out pieces of code so they are not executed by the program.
- You can patch in new pieces of code to be executed by the program.

In many cases, you correct an error in a program, so you need to use both types of patching. You patch out the incorrect lines of code and patch in the corrections.

## Conditionally Patching Out Code

For example, suppose a section of your C program dereferences a null pointer:

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

In this example, the caller of the **check\_for\_error** function assumes that passing 0 as the value of **error\_ptr** is allowed. The code should allow null values of **error\_ptr**, but line 3 dereferences a null pointer.

To correct this error, you can patch in code that checks for a null pointer. To do so, you set an evaluation point on line 3 and specify the following code fragment in the evaluation point:

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

If the value of **error\_ptr** is null, line 3 is not executed.

## Patching In a Function Call

As an alternative, you can patch in a **printf** statement that displays the value of **global\_error**. To do so, you create an evaluation point on line 4 and specify the following code fragment:

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

In this case, the code fragment is executed before the code on line 4, that is, before **global\_error** is set to 0.

## Correcting Code

In this final example, there is a coding error—the maximum value is returned instead of the minimum value:

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

To correct this error, you can set an evaluation point on line 4 and specify the following code fragment to correct the program's **if** statement.

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

## Interpreted and Compiled Expressions

On most platforms, TotalView interprets expressions, while on others, TotalView compiles expressions. See your platform-specific supplement to find out how TotalView handles expressions on your platform.

With interpreted expressions:

- TotalView sets a breakpoint in your code and executes the evaluation point. Since TotalView is executing the expression, interpreted expressions run slower than compiled expressions. With multiprocess programs, interpreted expressions can run



more slowly because processes may be waiting serially for the debugger to execute the expression. With remote debugging, interpreted expressions can run more slowly because the debugger, not the debugger server (**tvdsvr**), is executing the expression.

- If the expression contains **\$stop** or **\$count**, TotalView terminates the evaluation of the expression and stops the process. Thus, if you use **\$stop** or **\$count**, they should be at the end of your expression because TotalView stops evaluating the expression at that point.
- If you define an evaluation point at the same location as the PC, continuing execution does *not* execute the evaluation point.

With compiled expressions:

- TotalView compiles, links and patches the expression into the target process. To do this, TotalView replaces an instruction with a branch instruction, relocates the original instruction, and appends the expression. Then the code is executed by the target process, so conditional breakpoints can execute very fast.
- If the expression contains **\$stop** or **\$count**, TotalView stops the execution of the process in the compiled expression, so you can single step through it and continue executing the expression as you would the rest of your code.
- If you define an evaluation point at the same location as the PC, continuing execution executes the evaluation point.

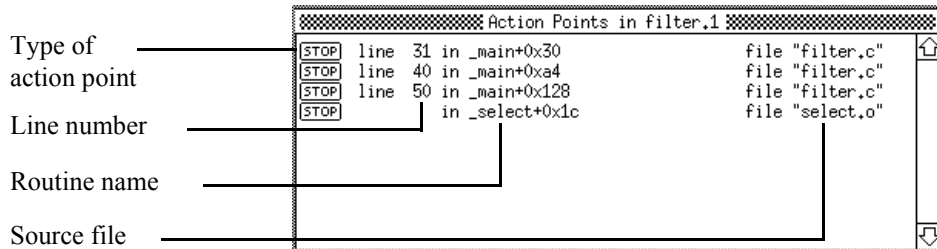
---

## Controlling Action Points

TotalView provides three methods of controlling action points: the action points window, the action points pane in the process window and the action point options dialog box.

## Displaying the Action Points Window

The action points window displays a summary of the action points that are set in your program. To display this window, display the **STOP/EVAL/GIST** submenu and select the **Open Action Points Window (b)** command. The action points window appears, as shown in Figure 47.



**Figure 47.** Action Points Window

---

**Note:** The list of action points displayed in the action points window is the same as shown in the action points pane in the process window.

---

If you dive into an action point in the action point list, TotalView displays the line of source code containing the action point in the source code pane of the process window.

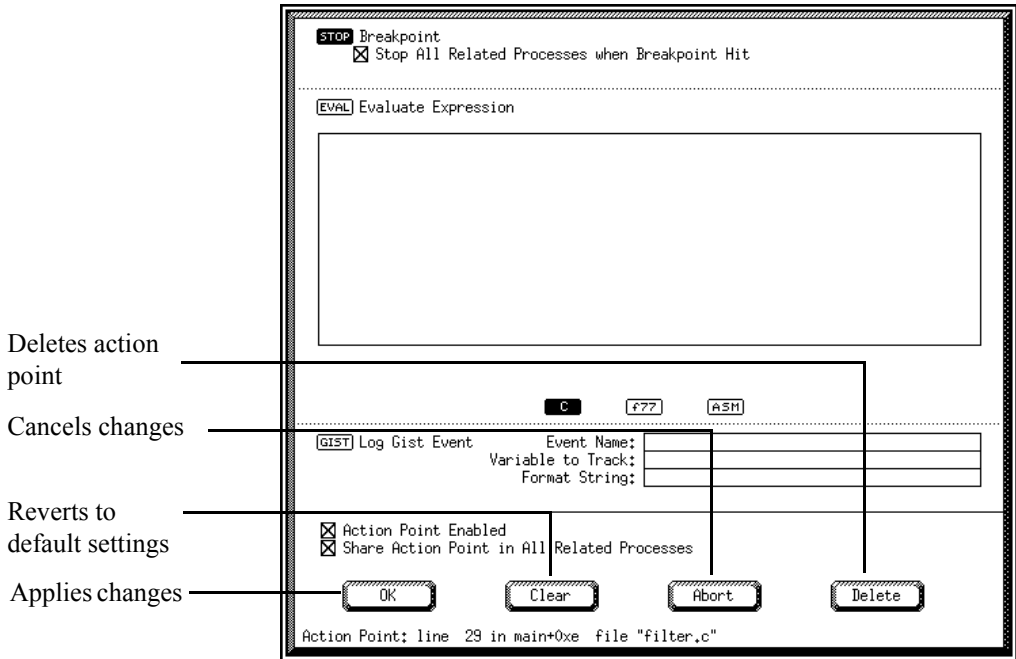
---

**Tip:** Action points make it easier to navigate your source files. You can define disabled breakpoints in your code and dive into the breakpoint to quickly display the corresponding source code in the process window. Thus, breakpoints can act like bookmarks in your program.

---

## Displaying the Action Point Options Dialog

The action point options dialog box lets you set and control an action point in your program. To display this dialog box, dive into the tag field beside a source line or an instruction. TotalView displays the dialog box, illustrated in Figure 48.



**Figure 48.** Action Point Options Dialog Box

## Controlling Action Points

You can take the following actions to control the use of action points in your program:

- |         |  |
|---------|--|
| Delete  | Permanently removes the action point.                                      |
| Disable | Keeps the definition for the action point but ignores it during execution. |
| Enable  | Makes the action point active during execution.                            |

Suppress	Keeps the definition for the action point, ignores it during execution, and prevents creation of additional action points.
Unsuppress	Makes the action point active during execution and allows creation of additional action points.

Table 14 shows how to control action points with the process window, action point options dialog, and the action points window.

**Table 14.** Clearing, Disabling, Enabling, Suppressing, and Unsuppressing Action Points

<b>How to Accomplish it for Each Type of Action Point</b>			
<b>Action</b>	<b>Breakpoint</b>	<b>Evaluation Point</b>	<b>Event Point</b>
Deleting	Select the <b>STOP</b> sign in tag field. Or Select the <b>Delete</b> button in the action point options dialog.	Select the <b>Delete</b> button in the action point options dialog.	Select the <b>Delete</b> button in the action point options dialog.
	To clear all breakpoints and evaluation points, go to the process window or action points window, display the <b>STOP/EVAL/GIST</b> submenu, and select the <b>Clear All STOP and EVAL</b> command.		To clear all event points, display the <b>STOP/EVAL/GIST</b> submenu and select the <b>Clear All GIST</b> command.
Disabling <sup>1</sup>	Deselect <b>Action Point Enabled</b> in the action point options dialog. Or Select the <b>STOP</b> sign in the action points window.	Select the <b>EVAL</b> sign in the tag field. Or Deselect <b>Action Point Enabled</b> in the action point options dialog. Or Select the <b>EVAL</b> sign in the action points window.	Select the <b>GIST</b> sign in the tag field.

**Table 14.** Clearing, Disabling, Enabling, Suppressing, and Unsuppressing Action

Action	How to Accomplish it for Each Type of Action Point		
	Breakpoint	Evaluation Point	Event Point
Enabling	Select the dimmed STOP, EVAL, or GIST sign in the process or action points window. Or Select <b>Action Point Enabled</b> in the action point options dialog.		
Suppressing <sup>2</sup>	To suppress all action points, display the <b>STOP/EVAL/GIST</b> submenu and select the <b>Suppress All Action Points (^D)</b> command.		To suppress all event points, display the <b>STOP/EVAL/GIST</b> submenu and select the <b>Suppress All GIST</b> command.
Unsuppressing	To unsuppress all action points, display the <b>STOP/EVAL/GIST</b> submenu and select the <b>Unsuppress All Action Points (^E)</b> command.		To unsuppress all event points, display the <b>STOP/EVAL/GIST</b> submenu and select the <b>Unsuppress All GIST</b> command.

1. Disabling an action point does not clear it. TotalView remembers that an action point exists for the line, but ignores it as long as it is disabled. For evaluation points, TotalView keeps the definition in case you want to use it again later.

2. When you suppress action points, you disable them. In addition, you cannot update any existing action points or create new ones.

---

## Saving Action Points in a File

You can save the action points for each program you debug in a file. TotalView names the file *program.TVD.breakpoints*, where *program* is the name of your program. To save your action points, display the **STOP/EVAL/GIST** submenu and select the **Save All Action Points** command from the process window. The debugger places the action points file in the same directory as your program.

If you always want to save your action points before you exit from TotalView, you can set an X Window System resource to do this. Refer to “totalview\*autoSaveBreakpoints: {true | false}” on page 164. Alternatively, you can use the **-sb** option each time you start the debugger, as described in “TotalView Command Syntax” on page 175.

Once you create an action points file, TotalView automatically loads the file each time you invoke the debugger. TotalView uses the same search paths as it does to locate source files. If you prefer to suppress this behavior, you can set an X resource (see “totalview\*autoLoadBreakpoints: {true | false}” on page 164) or use the **-nlb** option each time you start the debugger (see “TotalView Command Syntax” on page 175).

---

## Evaluating Expressions

In the TotalView debugger, you can open a window for evaluating expressions in the context of a particular process and evaluate expressions in C, Fortran, or Assembler.

---

**Note:** Not all platforms support the use of Assembler constructs; see your platform-specific supplement for details.

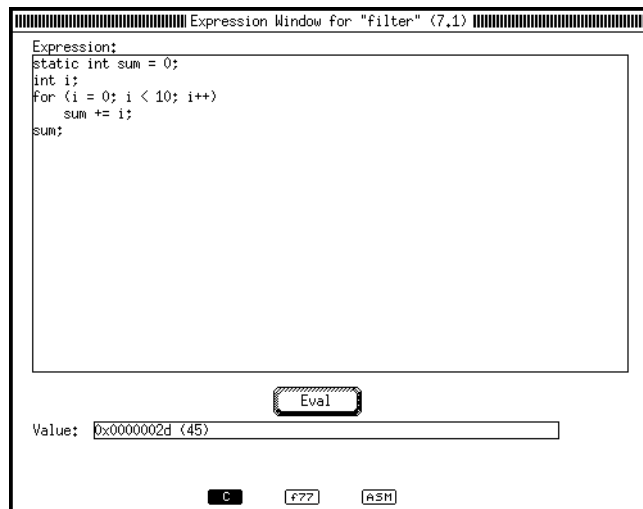
---

To evaluate an expression:

1. Make sure that a process is created, running, or stopped in the process window.

2. Select the **Open Expression Window (e)** command from the process window. An expression evaluation window appears.
3. Select the button (if it is not already selected) for the language in which you will write the code.
4. Select the Expression box and enter the code fragments to be evaluated using the field editor commands. For a description of the supported language constructs, see “Writing Code Fragments” on page 152.

The last statement in the code fragment can be a free-standing expression; you don’t have to assign the expression’s return value to a variable. Figure 49 shows a sample expression.



**Figure 49.** Sample Expression Window

5. Select the **Eval** button. If TotalView finds an error, it positions the cursor on the incorrect line and displays an error message. Otherwise, it interprets (or on some platforms, compiles and executes) the code, and displays the value of the last expression in the Expression box in the Value field.

While the code is being executed, you can’t modify anything in the window because it is suspended. If execution takes a long

time, notice that TotalView displays diagonal lines across the window, indicating that the window is temporarily inaccessible.

Since code fragments are evaluated in the context of the target process, the stack variables are evaluated according to the currently selected stack frame. If the fragment reaches a breakpoint (or stops for any other reason), the expression window remains suspended. Assignment statements can affect the target process because they can change the value of a variable in the target process.

You can use the expression window in many different ways, but here are two examples:

- Expressions can contain loops, so you could use a **for** loop to search an array of structures for the entry containing a particular field set to a certain value. In this case, you use the loop index at which the value is found as the last expression in the expression evaluation window.
- You can call subroutines from the expression window, so you could test and debug a single routine in your program without building a test program to call the routine.

Once you have selected and edited an expression in the window, you cannot use a keyboard equivalent (**q**) to exit from the window because the field editor is still active. To exit, display the menu and select the **Close Window** command or press Shift-Return.

---

## Writing Code Fragments

You can use code fragments in evaluation points and in the expression evaluation window. This section describes the intrinsic variables, built-in statements and language constructs supported by TotalView.



## Intrinsic Variables

The TotalView expression system supports built-in variables that allow you to access special thread and process values. All of the variables are of type 32-bit integer, which is type `<int>` or `<long>` on most platforms. The variables are not lvalues, so you cannot assign to them or take their addresses. Table 15 lists the intrinsic variable names and their meanings.

**Table 15.** Intrinsic Variables

Name	Meaning
<b>\$tid</b>	Returns the TotalView-assigned thread ID. When referenced from a process, generates an error.
<b>\$systid</b>	Returns the system-assigned thread ID. When referenced from a process, generates an error. Currently not implemented.
<b>\$pid</b>	Returns the process ID.
<b>\$nid</b>	Returns the node ID.
<b>\$clid</b>	Returns the cluster ID.
<b>\$duid</b>	Returns the TotalView-assigned Debugger Unique ID (DUID).
<b>\$processduid</b>	Returns the DUID of the process.

Intrinsic variables allow you to create thread specific breakpoints from the expression system. For example, using the **\$tid** intrinsic variable and the **\$stop** built-in operation, you can create a thread specific breakpoint as follows:

```
if ($tid == 3)
    $stop;
```

This would cause TotalView to stop the process only if thread 3 evaluated the expression. You can also create complex expressions using intrinsic variables:

```

if ($pid != 34 && $tid > 7)
    printf ("Hello from %d.%d\n", $pid,
    $tid);

```

## Built-In Statements

TotalView provides a set of built-in statements that you can use when writing code fragments. The statements are available in all languages, and are shown in Table 16.

**Table 16.** Built-In Statements That Can Be Used in Expressions

Statement	Use
<b>\$stopthread</b>	Sets a thread-level breakpoint. The thread that executes this statement stops, but all other threads in the process continue to execute. If the target system does not support asynchronous stop, this executes as a <b>\$stopprocess</b> .
<b>\$stopprocess</b>	Sets a process-level breakpoint. The process that executes this statement stops, but other processes in the program group continue to execute.
<b>\$stopall</b>	Sets a program-group-level breakpoint. <i>All</i> processes in the program group stop when any thread or process in the group executes this statement.
<b>\$stop</b>	Same as <b>\$stopprocess</b> .
<b>\$countthread <i>expression</i></b>	Sets a thread-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by <i>expression</i> <sup>1</sup> , it stops. The other threads in the process continue to execute. If the target system does not support asynchronous stop, this executes as a <b>\$countprocess</b> .
<b>\$countprocess <i>expression</i></b>	Sets a process-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by <i>expression</i> , the process stops. The other processes in the program group continue to execute.
<b>\$countall <i>expression</i></b>	Sets a program-group-level countdown breakpoint. <i>All</i> processes in the program group stop when any process in the group executes this statement for the number of times specified by <i>expression</i> .

**Table 16.** Built-In Statements That Can Be Used in Expressions (Continued)

Statement	Use
<b>\$count</b> <i>expression</i>	Same as <b>\$countprocess</b> .

1. A thread evaluates *expression* when it executes the **\$count** statement for the first time, and it must evaluate to a positive integer. A thread reevaluates **\$count** only when it results in a breakpoint. Then, the process' internal counter for the breakpoint is reset to the value of *expression*. The internal counter is stored in the process and shared by all threads in that process.

## C Constructs Supported

When writing code fragments in C, keep these guidelines in mind.

### Syntax

- C-style as well as C++-style comments are permitted. For example:

```
// This code fragment creates a temp patch  
i = i + 2;          /* Add two to i */
```

- Semicolons can be omitted when no ambiguity would result.
- Dollar signs (\$) in identifiers are permitted.

### Data Types and Declarations

- Data types permitted: **char**, **short**, **int**, **float**, **double**, and pointers to any primitive type or any named type in the target program.
- Only simple declarations are permitted. The **struct**, **union**, and array declarations are *not* permitted.
- References to variables of any type in the target program are permitted.
- Unmodified variable declarations are considered local. References to them override references to similarly named global variables and other variables in the target program.
- (Compiled evaluation points only) The **global** declaration makes a variable available to other evaluation points and expression windows in the target process.
- (Compiled evaluation points only) The **extern** declaration references a global variable that was or will be defined elsewhere.

If the global variable has not yet been defined, TotalView displays a warning.

- **Static** variables are local and persist even after an evaluation point has been evaluated.
- For static and global variables, expressions that initialize data as part of the variable declaration are performed only the first time the code fragment is evaluated. Local variables are initialized each time the code fragment is evaluated.

## Statements

- Permitted statements: assignment, **break**, **continue**, **if/else** structures, **for**, **goto**, and **while**.
- With the **goto** statement, you can define and branch to symbolic labels. These labels are considered local to the window. As an extension, you can also refer to a line number in the target program. This line number refers to the *tag field* number of the source code line. Here's a **goto** statement that causes the program to branch to source line number 432 of the target program:

```
goto 432;
```

- Function calls are permitted, but structures cannot be passed to a function.
- Type casting is permitted.
- All operators are permitted, with these limitations:
  - The conditional operator **?:** is not supported.
  - The **sizeof** operator can be used for variables, but not data types.
  - The **(type)** operator cannot cast to fixed-dimension arrays using C cast syntax.

## Fortran Constructs Supported

When writing code fragments in Fortran, keep these guidelines in mind.

## Syntax

- Syntax is free-form. No column rules apply.
- One statement is allowed per line, and one line is allowed per statement.
- The space character is significant and sometimes required. (Some Fortran 77 compilers ignore *all* space characters, wherever they are coded.) For example:

Valid	Invalid
DO 100 I=1,10	DO100I=1,10
CALL RINGBELL	CALL RING BELL
X .EQ. 1	X.EQ.1

GOTO, GO TO, ENDIF, and ENDIF are all allowed. But ELSEIF is not; use ELSE IF.

- Comment lines can be defined in several formats. For example:

```
C I=I+1
/*
I=I+1
J=J+1
ARRAY1(I,J)= I * J
*/
```

## Data Types and Declarations

- Data types permitted: INTEGER (assumed to be long), REAL, DOUBLE PRECISION, and COMPLEX.
- Implied data types are *not* permitted.
- Only simple declarations are permitted. The COMMON, BLOCK DATA, EQUIVALENCE, STRUCTURE, RECORD, UNION, and array declarations are *not* permitted.
- References to variables of any type in the target program are permitted.

## Statements

- Permitted statements: 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).

- As an extension to the GOTO statement, you can refer to a line number in the target program. This line number refers to the *tag field* number of the source code line. For example, this GOTO statement causes the program to branch to source line number 432 of the target program:

```
GOTO $432 ;
```

The dollar sign is *required* before the line number to distinguish the tag field number from a statement label.

- All expression operators are supported except CHARACTER operators and the logical operators .EQV., .NEQV., and .XOR..
- Subroutine function and entry definitions are not permitted.

## **Assembler Constructs Supported**

Refer to your platform-specific supplement, such as the *TotalView Supplement for LynxOS Users*, for information on the Assembler constructs supported. Some platforms do not support Assembler constructs at all.

## CHAPTER 7:

# Troubleshooting

This chapter describes how to solve common problems that you might encounter while using TotalView. Refer to Table 17.

**Table 17.** Symptoms and Solutions

Symptom	Possible Solutions
Windows don't appear or operate correctly	<ul style="list-style-type: none"><li>• Your DISPLAY environment variable is not set correctly.</li><li>• The resource “totalview*useTransientFor: {on   off}” on page 174 is not set correctly. Change it from <b>on</b> to <b>off</b>, or from <b>off</b> to <b>on</b>.</li><li>• Start Totalview with the <b>-grab</b> command-line option.</li><li>• Use the <b>xhost +</b> command to allow all hosts to access your display.</li></ul>
Pressing Control-C in an <b>xterm</b> window causes TotalView to exit	<ul style="list-style-type: none"><li>• Start TotalView with the <b>-ignore_control_c</b> or <b>-icc</b> command-line option.</li></ul>
Source code doesn't appear in source code pane	<ul style="list-style-type: none"><li>• Set the search path for directories with the <b>Set Search Directory (d)</b> command in the process window.</li><li>• Start TotalView with the <b>-nii</b> command-line option.</li></ul>
License manager does not operate correctly	<ul style="list-style-type: none"><li>• Set the <b>LM_LICENSE_FILE</b> environment variable to the pathname of the TotalView license file. See your platform-specific supplement for details.</li></ul>

**Table 17.** Symptoms and Solutions (Continued)

Symptom	Possible Solutions
Fatal error: Checkout ... failed	<ul style="list-style-type: none"><li>• Check the value of the <b>LM_LICENSE_FILE</b> environment variable. Make sure the value ends with the string <b>license.dat</b>.</li><li>• Make sure the TotalView license manager <b>lmgrd</b> is running on the license manager host machine. The name of this machine is listed in the <b>SERVER</b> line of your <b>license.dat</b> file.</li><li>• Make sure that the <b>lmgrd</b> that is running matches the one which came with your TotalView distribution.</li></ul>
Out of memory error	<ul style="list-style-type: none"><li>• Increase the swap space on your machine. See your platform-specific supplement for details.</li><li>• Increase the data and stack size limits in the C shell. Use the C shell's <b>limit</b> command, such as: <pre>% <b>limit datasize unlimited</b> % <b>limit stacksize unlimited</b></pre></li></ul>
Error creating new process	<ul style="list-style-type: none"><li>• Increase the swap space on your machine. See your platform-specific supplement for details.</li><li>• Increase the number of process slots in your system. See your operating system documentation for details.</li><li>• Check the <b>xterm</b> window to see if the <b>execve()</b> call failed, and if it did, set the <b>PATH</b> environment variable.</li><li>• Make sure that the <b>/proc</b> filesystem is mounted on your system. Refer to your platform-specific supplement for details.</li></ul>
Error launching process or Attempt to delete the target of an unbound process	<ul style="list-style-type: none"><li>• Run your program at the UNIX command line prompt to see if it will load and start executing. When it passes this test, you can run TotalView on your program to debug it.</li><li>• If the operating system can't load your program and start it, make sure your program is built for the machine you are debugging on.</li></ul>



**Table 17.** Symptoms and Solutions (Continued)

Symptom	Possible Solutions
Program behaves differently under TotalView's control	<ul style="list-style-type: none"> <li>• Make sure your program does not <code>setuid</code> or <code>exec</code> another program which does, for example, <b>rsh</b>. Normally, the operating system will not allow a debugger to debug a <code>setuid</code> executable nor allow a <code>setuid</code> system call while a program is being debugged. Often these operations fail silently. To debug <code>setuid</code> programs, login as the target UID before starting TotalView.</li> <li>• TotalView uses the <code>SIGSTOP</code> signal to stop processes. On most UNIX systems, system calls can fail with the <code>errno</code> set to <code>EINTR</code> when the process receives a <code>SIGSTOP</code> signal. You need to change your code so that it handles the <code>EINTR</code> failure. For example: <pre style="margin-left: 40px;">do {     n = read(fd, buf, nbytes); } while (n &lt; 0 &amp;&amp; errno == EINTR);</pre> </li> </ul>
X resources are not recognized	<ul style="list-style-type: none"> <li>• Use the <b>xrdb</b> command (part of the X Window System) to display the current X resources: <pre style="margin-left: 40px;">xrdb -query</pre> </li> <li>• Use the <b>xrdb</b> command to load your X resources: <pre style="margin-left: 40px;">xrdb -load \$HOME/.Xdefaults</pre> </li> <li>• Read the <b>xrdb</b> manual page for more information.</li> </ul>
Single stepping is slow or TotalView is slow to respond to breakpoints	<ul style="list-style-type: none"> <li>• Close some of the variable windows that you have open.</li> <li>• The global variables window is open and has a large number of variables. Close the global variables window.</li> </ul>
Other fatal error or Internal error in TotalView	<ul style="list-style-type: none"> <li>• Report this problem. See “Reporting Problems” on page 9.</li> </ul>



## CHAPTER 8:

# X Resources

This chapter provides reference information about the X Window System resources that you can use to customize TotalView. You can use these resources in your X resources files (such as **.Xdefaults** on UNIX systems or **decw\$sm\_general.dat** on VMS systems). For information on X resources files, refer to the X Window System documentation that came with your machine or the *X Window System User's Guide*, by O'Reilly & Associates (ISBN 1-56592-015-5).

On most UNIX systems, you load your X resources file using the **xrdb** command (part of the X Window System executables). For example:

```
% xrdb -load $HOME/.Xdefaults
```

The default value for each resource in this chapter is shown in **bold**. You can override some of the resources with command-line options for the **totalview** command, as described in “TotalView Command Syntax” on page 175.

Values for the location of windows are expressed as:

$$=widthxheight+x+y$$

where *width* is the width of the window in pixels, *height* is the height of the window in pixels, *x* is the distance from the **upper-left corner** of the window to the left screen edge in pixels, and *y* is the distance from the upper-left corner of the window to the top screen edge in pixels. A value of **-1** for *x* or *y* indicates that the window should be **centered** in the screen with respect to the x-axis or y-axis. If desired, you can express *x* or *y* as **negative** numbers to indicate the distance from the **lower-right corner** of the window to the bottom screen edge

or right screen edge instead of the distance from the upper-left corner. A value of zero (**0**) indicates that TotalView should use the **default** value. Also, you can supply just the size (*width* and *height*), and TotalView will use the default location (*x* and *y*) with it.

As an example, the expression **=0x0-1+20** uses the default width and height, centers the window horizontally, and places the window 20 pixels down from the top of the screen. The expression **=330x120+20-20** makes the window 330 pixels wide by 120 pixels high and places the window 20 pixels from the left edge of the screen and 20 pixels up from the bottom edge of the screen.

**totalview\*autoLoadBreakpoints: {true | false}**

If true (default), automatically load action points from an action points file, providing the file exists. If false, you use the **STOP/EVAL/GIST \* Load All Action Points** command in the process window to load action points.

**Override with:**                    **-lb** option (overrides the **false** value)  
   **-nlb** option (overrides the **true** value)

**totalview\*autoRetraceAddresses: {on | off}**

If on (default), TotalView will retrace the sequence of dive operations performed in a variable window and recompute a new address for the variable. If off, does not retrace addresses.

**totalview\*autoSaveBreakpoints: {true | false}**

If false (default), do not automatically save action points to an action points file when you exit. You use the **STOP/EVAL/GIST \* Save All Action Points** command in the process window to save action points.

**Override with:**                    **-sb** option (overrides the **false** value)  
   **-nsb** option (overrides the **true** value)

**totalview\*blindMouse: {on | off}**

If on (default), allow “*mouse ahead*,” the queuing of mouse clicks (similar to typing ahead in a shell). If off, successive mouse clicks are ignored until TotalView responds to the first mouse click.

**totalview\*breakpointWindLocation:** =*widthxheight+x+y*

Specifies placement of the first action points window.

**Default:**

<i>width</i>	<i>height</i>	<i>x</i>	<i>y</i>
columns(70)	lines(12)	335	10

**totalview\*chaseMouse:** {on | off}

If on (default), display dialog boxes at the location of the mouse cursor.

If off, display dialog boxes centered in the upper third of the screen.

**Override with:**

–chase option (overrides the **off** value)

–nochase option (overrides the **on** value)

**totalview\*cTypeStrings:** {true | false}

If false (default), use TotalView’s type string extensions when displaying the type strings for arrays. If true, use C type string syntax when displaying arrays.

**totalview\*dataWindLocation:** =*widthxheight+x+y*

Specifies placement of the first variable window.

**Default:**

<i>width</i>	<i>height</i>	<i>x</i>	<i>y</i>
columns(72)	max(205, lines(15))	-80	320

**totalview\*displayAssemblerSymbolically:** {on | off}

If off (default), display Assembler locations as hexadecimal addresses.

If on, display Assembler locations as “label+offset.”

**totalview\*editorLaunchString:** *command\_string*

Sets the editor launch command string to the specified value. Refer to “Changing the Editor Launch String” on page 88 for more information on the format of *command\_string*.

**Default:** xterm -e %E +%N %S

**totalview\*evalWindLocation:**  $=width \times height + x + y$

Specifies placement of the first expression evaluation window.

**Default:**

<i>width</i>	<i>height</i>	<i>x</i>	<i>y</i>
columns(83)	lines(30) + 2	-1	10

**totalview\*eventLogWindLocation:**  $=width \times height + x + y$

Specifies placement of the event log window.

**Default:**

<i>width</i>	<i>height</i>	<i>x</i>	<i>y</i>
columns(75)	lines(20)	-75	-50

**totalview\*font:** *fontname*

Specifies the font used by the TotalView debugger. Use the X Windows supplied application **xlsfonts** to list the names of available fonts.

**Default:** fixed

**totalview\*frameOffsetX:** *n*

Sets the horizontal placement offset between windows of the same type as TotalView places them on the screen. This value is *added* to the default value used by TotalView. If you are using TotalView title bars, use the default.

**Default:** 0

**totalview\*frameOffsetY:** *n*

Sets the vertical placement offset between windows of the same type as TotalView places them on the screen. This value is *added* to the default value used by TotalView. If you are using TotalView title bars, use the default.

**Default:** 0

**totalview\*globalsWindLocation:** =*widthxheight+x+y*

Specifies placement of the global variables window.

**Default:**

<i>width</i>	<i>height</i>	<i>x</i>	<i>y</i>
columns(62)	max(205, lines(15))	-80	10

**totalview\*grabMouse:** {on | off}

If off (default), do not force keyboard input to dialog boxes. If you're running TotalView with a window manager that is operating in "click-to-type" mode, you should set this resource to "on" or use the **-grab** command-line option.

**totalview\*helpWindLocation:** =*widthxheight+x+y*

Specifies placement of the help window.

**Default:**

<i>width</i>	<i>height</i>	<i>x</i>	<i>y</i>
min( <i>screen_width</i> - 10, columns(84))	min( <i>screen_height</i> - 20, 606)	-1	-20

**totalview\*ignoreIncludes:** {true | false}

If true (default), ignore all source line number information in include files.

**Override with:**            **-ii** option (overrides the **false** value)  
                              **-nii** option (overrides the **true** value)

**totalview\*kernelLaunchString:** *command\_string*

Specifies the command string that TotalView uses to automatically attach to a kernel through a serial line connection when you start kernel debugging. By default, TotalView uses the following strings to start the kernel debugging:.

**For Solaris:**                            tip %K  
**For Linux cross x86:**                 kermi -l /dev/com2 -b 19200 -c  
**For Linux cross PowerPC:**            kermi -l /dev/com2 -b 9600 -c

**totalview\*mainHSplit: *n***

Same as **totalview\*mainHSplit1**.

**totalview\*mainHSplit1: *n***

Controls the height of the stack trace, stack frame and source panes in the process window. *n* specifies the pixel location of the top of the source pane.

**Default:** (*window\_height*/3)

**totalview\*mainHSplit2: *n***

Controls the height of the source pane, thread list and action point list in the process window. *n* specifies the pixel location of the top of the thread list and action point list panes.

**Default:** A function of *window\_height*: Tries to give 5 lines in the thread list and action point list panes, and the remainder, at least 20 lines, to the source pane. If it cannot give the source pane at least 20 lines, it shrinks the thread list and action point list panes to zero.

**totalview\*mainVSplit: *n***

Same as **totalview\*mainVSplit1**.

**totalview\*mainVSplit1: *n***

Controls the location of the partition between the stack trace and stack frame panes in the process window. A value of **-1** centers the partition.

**Default:** (*window\_width*/2) - 20

**totalview\*mainVSplit2: *n***

Controls the location of the partition between the thread list and action point list panes in the process window. A value of **-1** centers the partition.

**Default:** (*window\_width*/2) - 20



**totalview\*mainWindLocation:** =*width**x**height*+*x*+*y*

Specifies placement of the first main process window.

**Default:**

<i>width</i>	<i>height</i>	<i>x</i>	<i>y</i>
min(columns(94), <i>screen_width</i> - 5)	max(456, lines(45))	10	-150

**totalview\*menuCache:** {on | off}

If off (default), disables menu caching. Not all X servers support menu caching. If your X server doesn't and you have menu caching enabled (**on**), TotalView menus appear blank the second and subsequent times you display them.

**totalview\*overrideRedirect:** {on | off}

If off (default), do not create TotalView windows using the **override\_redirect** attribute. If on, use the **override\_redirect** attribute, which does not give the X window manager a chance to intercept requests.

**totalview\*ownTitles:** {on | off}

If on (default), place title bars on TotalView windows. If your window manager is a reparenting one (places its own title bars on windows), turn off this resource.

**totalview\*pullRightMenus:** {on | off}

If off (default), use walking menus. If on, use pull-right menus.

**totalview\*pvmDebugging:** {true | false}

If false (default), disables support for debugging PVM applications. If true, enables support for debugging PVM applications.

**Override with:**

- pvm** option (overrides the **false** value)
- nopvm** option (overrides the **true** value)

**totalview\*rootWindLocation:** =*width**x**height*+*x*+*y*  
Specifies placement of the root window.

**Default:**

<i>width</i>	<i>height</i>	<i>x</i>	<i>y</i>
min( <i>screen_width</i> - 10, columns(60))	max(150, lines(12))	10	10

**totalview\*scrollLineSpeed:** *n*

Specifies the maximum number of lines per second that TotalView scrolls when you click on arrows at the top and bottom of the scroll bars. To have TotalView scroll as fast as possible, set *n* to **0**.

**Default:** 40

**totalview\*scrollPageSpeed:** *n*

Specifies the maximum number of pages per second that TotalView scrolls when you click above or below the elevator box inside the scroll bars. To have TotalView scroll as fast as possible, set *n* to **0**.

**Default:** 5

**totalview\*searchCaseSensitive:** {on | off}

If off (default), searching for strings is not case-sensitive. If on, searches are case-sensitive.

**totalview\*searchPath:** *dir1*[,*dir2*,...]

Specifies a list of directories for the debugger to search when looking for source and object files. This resource serves the same purpose as the **Set Search Directory** command in the process window (see “Setting Search Paths” on page 75). If you use multiple lines, place a backslash (\) at the end of each line, except for the last line.

**totalview\*serverLaunchEnabled:** {true | false}

If true (default), TotalView automatically launches the TotalView Debugger Server (**tvdsvr**) when you start to debug a remote process.

**totalview\*serverLaunchString:** *command\_string*

Specifies the command string that TotalView uses to automatically launch the TotalView Debugger Server (**tvdsvr**) when you start to debug a remote process. By default, TotalView uses the **rsh** command to start the server, but you can use any other command that can invoke **tvdsvr** on a remote host. If you have no command available for invoking a remote process, you can't automatically launch the server; therefore, you should set **totalview\*serverLaunchEnabled** to **false**.

**Default:** `rsh %R -n "cd %D && tvdsvr -callback %L -set_pw %P"`

**totalview\*serverLaunchTimeout:** *n*

Specifies the number of seconds that TotalView waits to hear back from the TotalView Debugger Server (**tvdsvr**) that it launched successfully. The number of seconds must be between **1** and **3600** (1 hour).

**Default:** 30

**totalview\*shareActionPoint:** {**true** | **false**}

Same as **totalview\*shareActionPointInAllRelatedProcesses**.

**totalview\*shareActionPointInAllRelatedProcesses:** {**true** | **false**}

If **true** (default), the default setting for action points will be to share them in all related processes. If **false**, the default setting for action points will be to *not* share them in all related processes. See “Breakpoints for Multiple Processes” on page 136.

**totalview\*signalHandlingMode:** *action\_list*

Modifies the way in which TotalView handles signals. An *action\_list* consists of a list of *signal\_action* descriptions, separated by spaces:

*signal\_action* [ *signal\_action* ] ...

A *signal\_action* description consists of an action and a list of signals:

*action=signal\_list*

An *action* can be one of the following: **Error**, **Stop**, **Resend**, or **Discard**. For more information on the meaning of each action, refer to “Handling Signals” on page 72.

A *signal\_list* is a list of signal specifiers, separated by commas:

*signal\_specifier*[,*signal\_specifier*] ...

A *signal\_specifier* can be a signal name (such as **SIGSEGV**), a signal number (such as **11**), or a \*, which specifies all signals.

---

**Note:** Since signal numbers vary from system to system, we recommend using the signal name. For example, on BSD UNIX systems, the **SIGUSR1** signal is assigned the signal number **30**, but on System V UNIX systems, the **SIGUSR1** signal is assigned the signal number **16**.

---

The following rules apply when specifying an *action\_list*:

- If you specify an action for a signal in an *action\_list*, TotalView changes the default action for that signal.
- If you do not specify a signal in the *action\_list*, TotalView does not change its default action for the signal.
- If you specify a signal that does not exist for the platform, TotalView ignores it.
- If you specify an action for a signal twice, TotalView uses the last action specified. In other words, TotalView applies the actions from left to right.

If you need to revert the settings for signal handling to TotalView’s built-in defaults, use the **Defaults** button in the **Set Signal Handling Mode** dialog box.

For example, to set the default action for the **SIGTERM** signal to **Resend**, you specify the following action list:

**“Resend=SIGTERM”**

As another example, to set the action for **SIGSEGV** and **SIGBUS** to **Error**, the action for **SIGHUP** and **SIGTERM** to **Resend**, and all remaining signals to **Stop**, you specify the following action list:

```
“Stop=* Error=SIGSEGV,SIGBUS  
Resend=SIGHUP,SIGTERM”
```

This action list shows how TotalView applies the actions from left to right. The action list first sets the action for all signals to **Stop**. Then, the action list changes the action for **SIGSEGV** and **SIGBUS** from **Stop** to **Error** and the action for **SIGHUP** and **SIGTERM** from **Stop** to **Resend**.

**totalview\*sourcePaneTabWidth:** *n*

Sets the width of the tab character that is displayed in the source pane. For example, if your source file uses a tab width of 4, set *n* to 4.

**Default:** 8

**totalview\*spellCorrection:** {**verbose** | **brief** | **none**}

When you use the **Function or File...** or **Variable...** commands in the process window or edit a type string in a variable window, the debugger checks the spelling of your entries. By default (**verbose**), the debugger displays a dialog box before it corrects spelling. You can set this resource to **brief** to run the spelling corrector silently. (The debugger makes the spelling correction without displaying it in a dialog box first.) You can also set this resource to **none** to disable the spelling corrector.

**totalview\*splitStatics:** {**on** | **off**}

If on (default), TotalView will enter multiple globally-scoped static variables with the same name in your program into the symbol table. If off, TotalView will enter only the first occurrence of a globally-scoped static variable into the symbol table, and discard subsequent occurrences with the same name.

**totalview\*stopAll:** {**true** | **false**}

Same as **totalview\*stopAllRelatedProcessesWhenBreakpointHit**.

**totalview\*stopAllRelatedProcessesWhenBreakpointHit: {true | false}**

If **true** (default), the default setting for breakpoints will stop all related processes. If **false**, the default setting for breakpoints will *not* stop all related processes. See “Breakpoints for Multiple Processes” on page 136.

**totalview\*useTransientFor: {on | off}**

If **off**, use “override redirect” windows, which don’t let you use the window manager to perform operations, such as raise and lower, on dialog boxes. If you use an advanced window manager, you can use the **on** option (default) to specify that the debugger use “transient-for” type windows, which allow you to use the window manager to perform operations on dialog boxes. If you’re using an X11R4 or more recent server and window manager, you should use the **on** option. If you’re using the DECstation’s DEC window manager, you should use the **off** option.

## CHAPTER 9:

# TotalView Command Syntax

This chapter summarizes the syntax of the **totalview** command. For the full syntax, use the **man totalview** command to view the online version.

### Synopsis

```
totalview [filename [corefile]] [options]
```

### Description

The TotalView debugger is a source-level debugger with a graphic interface (based on the X Window System) and features for debugging distributed programs, multiprocess programs, and multithreaded programs. It is available on a number of different platforms.

<i>filename</i>	Specifies the pathname of an executable to be debugged. The executable must be compiled with the <b>-g</b> compiler switch.
<i>corefile</i>	Specifies the name of a core file.

### Options

If you specify mutually exclusive options (such as **-ii** and **-nii**) on the same command line, the last option listed is used.

<b>-a</b> <i>args</i>	Passes all subsequent arguments (specified by <i>args</i> ) to the program specified by <i>filename</i> . This option must be the <i>last</i> one on the command line.
-----------------------	--

- chase** (Default) Displays dialog boxes at the mouse pointer. To display dialog boxes centered in the upper third of the screen, use **-nochase**.
- dbfork** (Default) Catches the **fork()**, **vfork()**, and **execve()** system calls if your executable is linked with the **dbfork** library.
- debug\_file** *consoleoutputfile*  
Redirects TotalView console output to a file named *consoleoutputfile*.  
  
Default: All TotalView console output is written to **stderr**.
- demangler=compiler**  
Overrides the C++ demangler and mangler TotalView uses by default. Table 18 lists override options.

**Table 18.** Demangling Style

<b>Option</b>	<b>Meaning</b>
<b>-demangler=cset</b>	IBM x1C C++
<b>-demangler=dec</b>	Digital C++
<b>-demangler=gnu</b>	GNU C++
<b>-demangler=spro</b>	SunPro C++ 4.0 or greater
<b>-demangler=sun</b>	Sun CFRONT C++
<b>-demangler=usoft</b>	MicroSoft C++
<b>-dumpcore</b>	Allows TotalView to dump a core file when it gets an internal error. Useful for debugging TotalView itself.
<b>-dynamic</b>	(Default) Loads symbols from shared libraries. This option is available only on platforms that support shared libraries.



<b>-ext</b> <i>extension</i>	Specifies that files with the suffix <i>extension</i> are preprocessor input files. TotalView already has built-in extensions for C++ ( <b>.C</b> , <b>.cpp</b> , <b>.cc</b> , <b>.cxx</b> ), Fortran ( <b>.F</b> ), <b>lex</b> ( <b>.l</b> , <b>.lex</b> ), and <b>yacc</b> ( <b>.y</b> ) files.
<b>-font</b> <i>fontname</i>	Specifies the font to be used by TotalView. Default: <b>fixed</b>
<b>-grab</b>	Forces all keyboard input to go to an open dialog box. Use this option if your window manager uses “click-to-type” mode.
<b>-icc</b>	Ignores Control-C and prevents you from terminating the TotalView process from an <b>xterm</b> window, which is useful when your program catches the Control-C signal (SIGINT).
<b>-ii</b>	(Default) Ignores source line information for filenames ending with an <b>.h</b> or <b>.hxx</b> suffix. To disable this, use the <b>-nii</b> option.
<b>-lb</b>	(Default) Loads action points automatically from the <i>filename.TVD.breakpoints</i> file, providing the file exists. To override this, use <b>-nlb</b> .
<b>-mc</b>	Turns on menu caching. Use this option if your X server supports menu caching. If menus appear blank the second and subsequent times you display them, your X server does not support menu caching.
<b>-nicc</b>	(Default) Catches Control-C and terminates your TotalView debugging session.
<b>-nii</b>	Does <i>not</i> ignore source line information for filenames ending with an <b>.h</b> or <b>.hxx</b> suffix.
<b>-nlb</b>	Does not load action points automatically from an action points file.
<b>-nmc</b>	(Default) Turns off menu caching.

<b>-nochase</b>	Displays dialog boxes centered in the upper third of the screen.
<b>-nodbfork</b>	Does not catch <b>fork()</b> , <b>vfork()</b> , and <b>execve()</b> system calls even if your executable is linked with the <b>dbfork</b> library.
<b>-nodumpcore</b>	(Default) Does not allow TotalView to dump a core file when it gets an internal error.
<b>-nodynamic</b>	Does not load symbols from shared libraries. Setting this option can cause the <b>dbfork</b> library to fail because TotalView might not find the <b>fork()</b> , <b>vfork()</b> , and <b>execve()</b> system calls.
<b>-nograb</b>	(Default) Does not force keyboard input to an open dialog box. To override this, use <b>-grab</b> .
<b>-nopvm</b>	(Default) Disables support for debugging PVM applications. To override this option, use <b>-pvm</b> .
<b>-npr</b>	(Default) Use walking menus instead of pull-right menus. To override this, use <b>-pr</b> .
<b>-nsb</b>	(Default) Does not save action points automatically to an action points file when you exit. To override this option, use <b>-sb</b> .
<b>-pr</b>	Use pull-right menus.
<b>-pvm</b>	Enables support for debugging PVM applications.
<b>-r[emote] <i>hostname</i>[:<i>portnumber</i>]</b>	Debugs an executable that is not running on the same machine as TotalView. For <i>hostname</i> , you can specify a TCP/IP hostname, such as <b>oak.bbn.com</b> , or a TCP/IP address, such as <b>128.89.0.16</b> . Optionally, you can specify a TCP/IP port number for <i>portnumber</i> , such as <b>:4174</b> .

- sb** Saves action points automatically to an action points file when you exit TotalView.
- serial device[:options]** Debugs an executable that is not running on the same machine as TotalView. For *device*, specify the device name of a serial line, such as **/dev/com1**. Currently the only *option* you are allowed to specify is the baud rate, which defaults to **38400**. For more information on debugging over a serial line, see “Debugging Over a Serial Line” on page 65.
- shm “action\_list”** Same as **-signal\_handling\_mode**.
- signal\_handling\_mode “action\_list”** Modifies the way in which TotalView handles signals. You must enclose the *action\_list* string in quotation marks to protect it from the shell. Refer to “totalview\*signalHandlingMode: action\_list” on page 171 for a description of the *action\_list* argument.



## CHAPTER 10:

# TotalView Debugger Server Command Syntax

This chapter summarizes the syntax of the TotalView Debugger Server command, **tvdsvr**, which is used for remote debugging. For more information on remote debugging, refer to “Starting the Debugger Server for Remote Debugging” on page 57.

---

**Note:** Remote debugging support is an option that you may have to purchase separately. See your platform-specific supplement for more information.

---

### Synopsis

```
tvdsvr {-server | -callback hostname:port | -serial device} [other options]
```

### Description

The **tvdsvr** debugger server allows TotalView to control and debug a program on a remote machine. To accomplish this, the **tvdsvr** program must run on the remote machine, and it must have access to the executables to be debugged. These executables must have the same absolute pathname as the executable that TotalView is debugging, or the PATH environment variable for **tvdsvr** must include the directories containing the executables.

You must specify either the **-server**, **-callback**, or **-serial** option with the **tvdsvr** command. By default, the TotalView debugger automatically launches **tvdsvr** (known as the auto-launch feature) with the **-callback** option, and the server establishes a connection with TotalView.

If you prefer not to use the auto-launch feature, you can start **tvdsvr** manually and specify the **-server** option. Be sure to make note of the password that **tvdsvr** prints out with the message:

```
pw = hexnumhigh:hexnumlow
```

TotalView will prompt you for *hexnumhigh:hexnumlow* later. By default, **tvdsvr** automatically generates a password that is used when establishing connections. If desired, you can use the **-set\_pw** option to set a specific password.

To connect to the **tvdsvr** from TotalView, you use the New Program Window and must specify the hostname and TCP/IP port number, *hostname:portnumber* on which **tvdsvr** is running. Then, TotalView prompts you for the password for **tvdsvr**.

## Options

The following options determine the port number and password necessary for TotalView to connect with **tvdsvr**.

**-callback** *hostname:port*

(Auto-launch feature only) Immediately establishes a connection with the TotalView debugger that is running on *hostname* and listening on *port*, where *hostname* is either a hostname or TCP/IP address. If **tvdsvr** cannot connect with TotalView, it exits. If you specify the **-port**, **-search\_port**, and **-server** options with this option, **tvdsvr** ignores them.

**-debug\_file** *consoleoutputfile*

Redirects TotalView Debugger Server console output to a file named *consoleoutputfile*.

Default: All console output is written to **stderr**.

- port *number*** Sets the TCP/IP port number on which **tvdsvr** should communicate with **totalview**. If this TCP/IP port number is busy, **tvdsvr** does not select an alternate port number (that is, it communicates with nothing) unless you also specify **-search\_port**.
- Default: 4142
- pvm** Uses the Parallel Virtual Machine (PVM) library process as its input channel and registers itself as the PVM tasker.
- Note:** This option is not intended for users launching **tvdsvr** manually. When you enable PVM support within TotalView, TotalView automatically uses this option when it launches **tvdsvr**.
- search\_port** Searches for an available TCP/IP port number, beginning with the default port (4142) or the port set with the **-port** option and continuing until one is found. When the port number is set, **tvdsvr** displays the chosen port number with the following message:
- ```
port = number
```
- serial *device*[:*options*]** Waits for a serial line connection from TotalView. For *device*, specify the device name of a serial line, such as **/dev/com1**. Currently the only *option* you are allowed to specify is the baud rate, which defaults to **38400**. For more information on debugging over a serial line, see “Debugging Over a Serial Line” on page 65.

**-server** Listens for and accepts network connections on port 4142 (default). To use a different port, you must specify the **-port** or **-search\_port** options. To stop **tvdsvr** from listening and accepting network connections, you must terminate it by pressing Control-C in the terminal window from which it was started or by using the **kill** command.

**-set\_pw** *hexnumhigh:hexnumlow*

Sets the password to the 64-bit number specified by the two 32-bit numbers *hexnumhigh* and *hexnumlow*. When a connection is established between **tvdsvr** and TotalView, the 64-bit password passed by TotalView must match the password set with this option. When the password is set, **tvdsvr** displays the selected number in the following message:

```
pw = hexnumhigh:hexnumlow
```

We recommend using this option to avoid connections by other users.

---

**Note:** If necessary, you can disable password checking by specifying the **-set\_pw 0:0** option with the **tvdsvr** command. Disabling password checking is dangerous: it allows anyone to connect to your server and start programs, including shell commands, using your UID. Therefore, we don't recommend disabling password checking.

---



# Glossary

|                          |                                                                                                                                                                                                                                                                                                              |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>action point</b>      | A point in a program where a breakpoint, evaluation point, or event point has been set during a TotalView session.                                                                                                                                                                                           |
| <b>address space</b>     | A region of memory that contains code and data from a program. One or more threads can run in an address space. A process normally contains an address space.                                                                                                                                                |
| <b>breakpoint</b>        | A point in a program where execution can be conditionally suspended to permit examination and manipulation of data.                                                                                                                                                                                          |
| <b>child process</b>     | A process created by another process (see parent process) when that other process calls <code>fork()</code> .                                                                                                                                                                                                |
| <b>cluster debugging</b> | The action of debugging a program that is running on a cluster of hosts in a network. Typically, the hosts are homogeneous.                                                                                                                                                                                  |
| <b>core file</b>         | A file containing the contents of memory and a list of thread registers. The operating system dumps (creates) a core file whenever a program exits because of a severe error (such as an attempt to store data into an invalid address).                                                                     |
| <b>cross debugging</b>   | A special case of remote debugging where the host platform and the target platform are different types of machines.                                                                                                                                                                                          |
| <b>dbfork library</b>    | A library of special versions of the <code>fork()</code> and <code>execve()</code> calls used by the TotalView debugger to debug multiprocess programs. Programs that call one of the <code>fork()</code> , <code>vfork()</code> , or <code>execve()</code> routines must be linked with the dbfork library. |
| <b>debugger server</b>   | <i>See</i> the glossary entry for <b>tvdsvr process</b> .                                                                                                                                                                                                                                                    |

|                              |                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>distributed debugging</b> | The action of debugging a program that is running on more than one host in a network. The hosts can be homogeneous or heterogeneous. For example, programs written with message passing libraries such as Parallel Virtual Machine (PVM) or Parallel Macros (PARMACS) run on more than one host.                                                                                 |
| <b>dive stack</b>            | A series of nested dives that were performed in the same variable window. The number of right angle brackets (>) in the upper left hand corner of a variable window indicates the number of nested dives on the dive stack. Each time that you undive, TotalView pops a dive from the dive stack and decrements the number of right angle brackets shown in the variable window. |
| <b>diving</b>                | The action of clicking the right mouse button to display more information about an item. For example, if you dive into a variable in the TotalView debugger, a window appears with more information about the variable.                                                                                                                                                          |
| <b>editing cursor</b>        | A black rectangle that appears when a TotalView field is selected for editing. You use field editor commands to move the editing cursor.                                                                                                                                                                                                                                         |
| <b>evaluation point</b>      | A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.                                                                                                                                                                                                                                                                  |
| <b>event log</b>             | A file containing a record of events for each process in a program.                                                                                                                                                                                                                                                                                                              |
| <b>event point</b>           | A point in the program where TotalView writes an event to the event log for later analysis using the TimeScan Performance Analyzer.                                                                                                                                                                                                                                              |
| <b>extent</b>                | The number of elements in the dimension of an array. For example, a Fortran array of integer(7,8) has an extent of 7 in one dimension (7 rows) and an extent of 8 in the other dimension (8 columns).                                                                                                                                                                            |
| <b>field editor</b>          | A basic text editor that is part of TotalView's interface. The field editor supports a subset of GNU Emacs commands.                                                                                                                                                                                                                                                             |
| <b>gridget</b>               | A dotted grid in the tag field that indicates you can set an action point on the instruction.                                                                                                                                                                                                                                                                                    |
| <b>host machine</b>          | The machine on which the TotalView debugger is running.                                                                                                                                                                                                                                                                                                                          |

|                           |                                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>lower bound</b>        | The first element in the dimension of an array or the slice of an array. By default, the lower bound of an array is 0 in C and 1 in Fortran, but the lower bound can be any number, including negative numbers.                                                                                                                                                                 |
| <b>native debugging</b>   | The action of debugging a program that is running on the same machine as TotalView.                                                                                                                                                                                                                                                                                             |
| <b>nested dive window</b> | A TotalView window that results from diving into an item in a variable window. A nested dive window replaces the contents of the variable window and has an undive symbol in its title bar. Diving on the undive symbol returns the original contents of the variable window.                                                                                                   |
| <b>parcel</b>             | The number of bytes required to hold the shortest instruction for the target architecture.                                                                                                                                                                                                                                                                                      |
| <b>parent process</b>     | A process that calls <code>fork()</code> to spawn other processes (usually called child processes).                                                                                                                                                                                                                                                                             |
| <b>PARMACS library</b>    | A message passing library for creating distributed programs that was developed by the German National Research Centre for Computer Science.                                                                                                                                                                                                                                     |
| <b>process</b>            | Consists of an address space and a list of one or more threads running in that address space.                                                                                                                                                                                                                                                                                   |
| <b>process group</b>      | A group of processes associated with a multiprocess program. Includes program groups and share groups.                                                                                                                                                                                                                                                                          |
| <b>process window</b>     | The main TotalView window for a process, which consists of three panes: the stack trace, the stack frame, and the source code for the program.                                                                                                                                                                                                                                  |
| <b>program group</b>      | A group of processes that includes the parent process and all related processes. A program group includes children that were forked (processes that share the same source code as the parent) and children that were forked with a subsequent call to <code>execve()</code> (processes that do <i>not</i> share the same source code as the parent). Contrast with share group. |
| <b>PVM library</b>        | Parallel Virtual Machine library. A message passing library for creating distributed programs that was developed by the Oak Ridge National Laboratory and the University of Tennessee.                                                                                                                                                                                          |

|                              |                                                                                                                                                                                                                                                                                                                                         |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>remote debugging</b>      | The action of debugging a program that is running on a different machine than TotalView. The machine on which the program is running can be located many miles away from the machine on which TotalView is running.                                                                                                                     |
| <b>root window</b>           | A TotalView window displaying the process ID, status (e.g., at breakpoint or stopped), name, and current routine executing for each process being debugged.                                                                                                                                                                             |
| <b>serial line debugging</b> | A form of remote debugging where TotalView and the TotalView Debugger Server communicate over a serial line.                                                                                                                                                                                                                            |
| <b>share group</b>           | A group of processes that includes the parent process and any related processes that share the same source code as the parent. Contrast with program group.                                                                                                                                                                             |
| <b>signals</b>               | Messages informing processes of asynchronous events, such as serious errors. The action the process takes in response to the signal depends on the type of signal and whether or not the program includes a signal handler routine, a routine that traps certain signals and determines appropriate actions to be taken by the program. |
| <b>single step</b>           | The action of executing a single statement and stopping (as if at a breakpoint).                                                                                                                                                                                                                                                        |
| <b>slice</b>                 | A subsection of an array, which is expressed in terms of a lower bound, upper bound, and stride. Displaying a slice of an array can be useful when working with very large arrays, which is often the case in Fortran programs.                                                                                                         |
| <b>stack</b>                 | A portion of computer memory and/or registers used to hold information temporarily. The stack consists of a linked list of stack frames that holds return locations for called routines, routine arguments, local variables, and saved registers.                                                                                       |
| <b>stack frame</b>           | A section of the stack that contains the local variables, arguments, contents of the registers used by an individual routine, a frame pointer pointing to the previous stack frame, and the value of the Program Counter (PC) at the time the routine was called.                                                                       |
| <b>stack trace</b>           | A sequential list of each currently active routine called by a program and the frame pointer pointing to its stack frame.                                                                                                                                                                                                               |

|                        |                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>stride</b>          | The interval between array elements in a slice and the order in which the elements are displayed. If the stride is 1, every element between the lower bound and upper bound of the slice is displayed. If the stride is 2, every other element is displayed. If the stride is -1, every element between the upper bound and lower bound (reverse order) is displayed. |
| <b>symbol table</b>    | A table of symbolic names (such as variables or functions) used in a program and their memory locations. The symbol table is part of the executable object generated by the compiler (with the <code>-g</code> switch) and is used by debuggers to analyze the program.                                                                                               |
| <b>tag field</b>       | The left margin in the source code pane of the TotalView process window containing boxed line numbers marking the lines of source code that actually generate executable code.                                                                                                                                                                                        |
| <b>target machine</b>  | The machine on which the process to be debugged is running.                                                                                                                                                                                                                                                                                                           |
| <b>thread</b>          | An execution context that normally contains a set of private registers and a region of memory reserved for an execution stack. A thread runs in an address space.                                                                                                                                                                                                     |
| <b>tvdsrv process</b>  | The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.                                                                                                                                                                        |
| <b>undiving</b>        | The action of displaying the previous contents of a window, instead of the contents displayed for the current dive. To undive, you dive on the undive icon in the upper right-hand corner of the window.                                                                                                                                                              |
| <b>upper bound</b>     | The last element in the dimension of an array or the slice of an array.                                                                                                                                                                                                                                                                                               |
| <b>variable window</b> | A TotalView window displaying the name, address, data type, and value of a particular variable.                                                                                                                                                                                                                                                                       |



# Index

## Symbols

- , (comma), in specifying a range of addresses 112
- . (period)
  - in searches 41
  - in suffix of process names 93
- / (slash), in searching for strings 41
- : (colon), in array type strings 118
- > (right angle bracket), indicating nested dives 115
- \ (backslash), in searching backward for strings 41

## A

–a option 47, 175

### action points

- action points window 146, 165
- definition 185
- deleting 148
- disabling 148
- enabling 149
- features available 17
- loading automatically 177
- machine level 84
- saving 150, 178
- slow performance 161
- suppressing 149
- unsuppressing 149

### address space

- definition 185

### addresses

- changing 130
- of machine instructions 84, 130
- retracing 164
- specifying in variable window 112
- tracking in variable window 108

allocated arrays, displaying 125

Allows Asynchronous Stop capability 90

Allows Atomic Run capability 91

Allows Multithreaded Signal Delivery capability 91

Allows Read While Running capability 91

angle brackets, in windows 115

areas of memory, data type 123

### arguments

- for totalview command 175
- for tvdsrv command 182
- in server launch command 58, 61
- passing to program 47
- setting 77

argv array, displaying 124

### arrays

- character 123
- declared versus allocated 125
- displaying 112
- displaying argv 124
- displaying slices 125
- diving into 38
- lower bound 118
- type strings for 118

## Index

- upper bound 118
- ASM sign 134
- Assembler
  - and `-g` compiler option 38
  - constructs 158
  - display symbolically 165
  - examining 84
- Assembler Display Mode command 84
- Asynchronous Run capability 90
- Asynchronous Stop capability 90
- at breakpoint state 71
- attaching to processes 21, 50, 54
- auto-launch feature
  - (figure) 58
  - changing options 58, 170
  - description 57
  - disabling 62

## B

- B state 71
- base window, displaying a new 115
- bit fields 116
- bookmarks 146
- boxes, in tag field 29
- breakpoint state 71
- breakpoints
  - action points window 146
  - clearing 33
  - conditional 140, 154
  - countdown 154
  - definition 185
  - loading automatically 164
  - machine level 84, 134
  - saving 150, 164
  - setting 33, 134, 136
  - sharing 21, 138
  - slow performance 161
  - thread specific 153
- built-in

- intrinsic variables
  - `$clid` 153
  - `$duid` 153
  - `$nid` 153
  - `$pid` 153
  - `$processduid` 153
  - `$systid` 153
  - `$tid` 153
- statements
  - `$count` 155
  - `$countall` 154
  - `$countprocess` 154
  - `$countthread` 154
  - `$stop` 154
  - `$stopall` 154
  - `$stopprocess` 154
  - `$stopthread` 154
- type strings 121

## buttons

- EVAL 151
- go back 30, 32
- mouse, using 33
- next process 30, 32
- next thread 30, 32
- previous process 30, 32
- previous thread 30, 32
- undive 114

## C

### C

- array bounds 118
- file suffixes 86
- in evaluation points 155
- type strings
  - parameter in `.Xdefaults` file 165
  - supported 116

- C++ template functions 99

- caching variables 132

- call stack 29

- `-callback` option 58, 182

- cancelling single step operations 34

- capabilities



- Allows Asynchronous Stop 90
- Allows Atomic Run 91
- Allows Multithreaded Signal Delivery 91
- Allows Read While Running 91
- Asynchronous Run 90
- Asynchronous Stop 90
- Synchronous Run 90
- Synchronous Stop 90
- case sensitive searches 170
- casting, types of variable 116
- changing
  - auto-launch options 58
  - program groups 95
  - variables 115
- characters, arrays of 123
- chase option 176
- checking spelling 42
- child processes
  - definition 185
  - names 93
- Clear All GIST command 148
- Clear All STOP and EVAL command 148
- clearing
  - breakpoints 33, 136
  - evaluation points 33
  - event points 33
- \$clid intrinsic variable 153
- Close All Similar Windows command 113
- Close Window command 113
- closing windows 113
- cluster debugging 54
  - definition 185
- code constructs supported
  - Assembler 158
  - C 155
  - Fortran 156
- <code> data type 123
- <code> type string 130
- commands
  - arguments 77
  - Assembler Display Mode 84
  - Clear All GIST 148
  - Clear All STOP and EVAL 148
  - Close All Similar Windows 113
  - Close Window 113
  - Create Process (without starting it) 92
  - Detach from Process 55
  - Display Assembler by Address 84
  - Display Assembler Symbolically 84
  - Duplicate Window 115
  - Edit Source Text 87
  - Editor Launch String 89
  - Find Interesting Relative 96
  - for launching tvdsrv 58, 171
  - Function or File 86
  - Global Variables Window 111
  - Go Group 92
  - Go Process 91
  - Go Thread 92
  - Halt Group 105
  - Halt Process 105
  - Halt Thread 105
  - Help 37
  - input and output files 80
  - Input from File 80
  - Interleave Display Mode 84, 104
  - issuing 33
  - New Base Window 115
  - New Program Window 49, 51, 56
  - Next (instruction) 100
  - Next (source line) 100
  - Open Action Points Window 146
  - Open Expression Window 151
  - Output to File 80
  - Quit Debugger 43
  - Reexecute Last Save Window 43
  - Reexecute Last Search 41
  - Reload Executable File 50
  - Return (out of function) 101
  - rsh 60
  - Run to Selection 100
  - Save All Action Points 150
  - Save Window to File 43
  - Search Backward 41
  - Search Backward for String 41

## Index

- Search for String 41
  - Search Forward 41
  - Server Launch String 63
  - Set Command Arguments 77
  - Set Continuation Signal 55, 102
  - Set Environment Variables 79
  - Set PC to Absolute Value 104
  - Set PC to Selection 104
  - Set Process Program Group 96
  - Set Search Directory 75
  - Set Signal Handling Mode 73
  - Show All Process Groups 94
  - Show All Unattached Processes 51
  - Show Event Log Window 80
  - Single Step 98
  - Source Display Mode 84
  - Step (instruction) 99
  - Step (source line) 92, 99
  - Suppress All Action Points 149
  - Suppress All GIST 149
  - totalview, syntax and use 175
  - tvdsrv, syntax and use 181
  - Unsuppress All Action Points 149
  - Unsuppress All GIST 149
  - Update Process Info 105
  - Variable 109, 111, 128
  - common blocks, displaying 110
  - compiled expressions 144
  - compiling programs 26, 38, 46
  - conditional breakpoints 18, 140, 142, 154
  - continuing
    - with a signal 102
  - control registers, interpreting 89
  - copying text between windows 39
  - core files
    - definition 185
    - examining 56
    - in totalview command 47, 56
  - \$count statement 155
  - \$countall statement 154
  - countdown breakpoints 142, 154
  - \$countprocess statement 154
  - \$countthread statement 154
  - CPU registers 89
  - Create Process (without starting it) command 92
  - creating processes 49, 91, 160
    - using Step (source line) 92
    - without starting them 92
  - cross debugging
    - definition 13, 185
  - cursor, with arrows 30
  - customizing TotalView 163
- ## D
- dbfork library 46, 178, 185
  - dbfork option 176
  - debug\_file option 176, 182
  - debugger server 19, 57, 171, 181
  - debugging
    - multiprocess programs 20, 46
    - programs that call execve 46
    - programs that call fork 46
    - remote processes 53
  - declared arrays, displaying 125
  - deleting
    - action points 148
    - processes 106
  - demangler option 176
  - Detach from Process command 55
  - dialogs
    - action point options 137, 147
    - attach to process 52
    - behavior of 165, 177
    - change process group 96
    - debug remote process 53
    - environment variables 79
    - input from file 80
    - launch debugger server 62
    - load new program 49
    - location of 176

- output to file 80
  - serial line debugging 68
  - set command arguments 77
  - set search directory 76
  - set signal handling mode 73
  - spelling corrector 42
- directories, setting order of search 75
- disabling
  - action points 148
  - auto-launch feature 62, 170
  - PVM support 169, 178
- disassembly, in variable window 130
- Display Assembler by Address command 84
- Display Assembler Symbolically command 84
- displaying
  - areas of memory 112
  - argv array 124
  - arrays 112, 125
  - common blocks 110
  - global variables 109
  - more detail about objects 37
- distributed debugging 13, 19, 53, 57
  - definition 186
- dive stack
  - definition 186
- diving
  - definition 115, 186
  - into Fortran common blocks 111
  - into functions 86
  - into global variables 109
  - into local variables 109
  - into processes 31, 37, 94
  - into registers 108
  - into threads 32, 37
  - into variables 38
  - into windows 37
- \$duid intrinsic variable 153
- dumpcore option 176
- Duplicate Window command 115
- dynamic option 176

## E

- E state 71
- Edit Source Text command 87
- editing
  - closing editor 34
  - cursor
    - (figure) 39
    - definition 39, 186
    - selecting field for 33
    - source text 87
    - text with field editor 38
    - type strings 116
- editor launch string 88
  - default 88
- Editor Launch String command 89
- enabling
  - action points 149
  - PVM support 169, 178
- environment variables 78
- error state 71
- errors 159
- Eval button 151
- EVAL sign, for evaluation points 33
- evaluating expressions 150
- evaluation points
  - Assembler constructs 158
  - C constructs 155
  - clearing 33
  - commands 154
  - defining 140
  - definition 186
  - examples 142
  - Fortran constructs 156
  - machine level 84, 140
  - setting 33, 141
  - slow performance 161
- event log
  - definition 21, 186
  - window 80, 166
- event points

## Index

- clearing 33
  - definition 186
  - machine level 84
  - setting 33
  - slow performance 161
- examining
- core files 56
  - process groups 94
  - source and Assembler code 84
  - stack trace and stack frame 108
  - status and control registers 89
- exception enable modes 89
- executing
- to a selected line 100
  - to the completion of a function 101
- execve
- debugging programs that call 46
- execve call
- attaching to processes 50
  - failure of 160
  - setting breakpoints with 138
- exiting TotalView 34, 43
- expression evaluation window
- compiled and interpreted expressions 144
  - discussion 150
  - location 166
- ext option 177
- extent
- definition 186
- ## F
- field editor
- definition 186
  - editing text with 38
  - ending session 34
- Find Interesting Relative command 96
- finding functions 86
- font option 177
- fonts, in .Xdefaults file 166
- fork
- debugging programs that call 46
  - fork call, and setting breakpoints 138
- Fortran
- array bounds 118
  - common blocks 110
  - file suffixes 86
  - in evaluation points 156
  - type strings, supported by TotalView 116
- Function or File command 86
- functions, finding 86
- ## G
- g compiler option 38, 46
- generating a symbol table 46
- GIST sign, for event points 33
- global variables window
- discussion 111
  - location 167
- Global Variables Window command 111
- global variables, diving into 109
- go back button 30
- Go Group command 92
- Go Process command 91
- Go Thread command 92
- goto statements 140
- grab option 47, 177
- gridget 84, 134
- groups, definition 93
- ## H
- Halt Group command 105
- Halt Process command 105
- Halt Thread command 105
- handling signals 72, 171, 179
- Help command 37
- help window

- displaying 37
- features available 23
- location 167
- hexadecimal address, specifying in variable window 112
- host machine, definition 19, 186
- hostname
  - for tvdsvr 48, 53, 54, 182
  - in root and process windows 28, 69

**I**

- I state 70
- icc option 177
- idle state 70
- ignoring include files 167
- ii option 177
- inactive menu commands 33
- include files, ignoring 167
- input files, setting 80
- instructions, displaying 113, 130
- Interleave Display Mode command 84
- interpreted expressions 144
- intrinsic variables 153
  - \$clid 153
  - \$duid 153
  - \$nid 153
  - \$pid 153
  - \$processduid 153
  - \$systid 153
  - \$tid 153

**K**

- keyboard equivalents 34

**L**

- labels, for machine instructions 130
- launching tvdsvr 57, 170, 181

- lb option 177
- left mouse button 33
- lex utility, file suffixes 87
- libraries
  - dbfork 46, 178, 185
  - shared 176
- license manager 159
- line numbers, in tag field 29
- loading
  - action points 164, 177
  - new executables 48, 53
- local variables, diving into 108
- location
  - of processes 28, 69
  - of windows 170
- lower bound, of array slices 125

**M**

- M state 71
- machine instructions
  - data type 123
  - displaying 113, 130
  - single stepping 99
- mc option 177
- memory error 160
- memory, displaying areas of 38, 112
- menus
  - blank menus 177
  - caching 177
  - customizing behavior of 169
  - popping up 33
- messages 159
- middle mouse button 33
- mixed state 71
- mouse buttons, using 33
- multiple variables, with same name 109
- multiprocess programs

## Index

- and distributed debugging 19
- and signals 74
- attaching to 52
- compiling 46
- features available 20, 22
- finding active processes 96
- loading new executables 54
- PCs in 29
- process groups 93
- setting and clearing breakpoints 136

## N

- n option, of rsh command 61
- names, of processes in process groups 93
- native debugging 19
  - definition 187
- navigating
  - in the process window 30
  - in the root window 31
- nested dive window 38, 115
  - definition 187
- New Base Window command 115
- New Program Window command 49, 51, 56
- Next (instruction) command 100
- Next (source line) command 100
- nicc option 177
- \$nid intrinsic variable 153
- nii option 86, 177
- nlb option 177
- nmc option 177
- nochase option 178
- nodbfork option 178
- nodumpcore option 178
- nodynamic option 178
- nograb option 178
- nopvm option 178
- notes

- bit fields 116
- breakpoints apply to processes 134
- changing global variables 92
- copying text between windows 40
- diving into subroutines 38
- editing compound objects or arrays 120
- editing type strings 117
- how TotalView determines share group 95
- interleave display mode 84
- machine architecture and distributed debugging 19
- multiple variables with same name 109
- prefix for hexadecimal addresses 112
- specifying search directories 76
- variable window, tracking addresses 108

- npr option 178

- nsb option 178

## O

- O option 46
- offsets, for machine instructions 130
- opaque type definitions 124
- Open Action Points Window command 146
- Open Expression Window command 151
- optimizations, compiling for 46
- output files, setting 80
- override-redirect windows 169

## P

- panes
  - location and size 168
  - resizing 30
- parent processes, definition 187
- passing arguments 47
- password, generated by tvdsvr 182
- pasting text between windows 39
- patching programs 143
- PATH environment variable 75

- PC
  - See* program counter (PC)
- performance
  - action points 161
  - interpreted and compiled expressions 144
  - of remote debugging 57
- \$pid intrinsic variable 153
- pointers
  - diving into 38
  - to arrays 117
- popping up menus 33
- port number, for tvdsvr 48, 53, 54, 182
- port option 63, 183
- pr option 178
- preprocessors 87, 177
- procedures
  - attaching to processes 50, 54
  - changing
    - auto-launch options 62
    - program groups 95
    - variables 115
  - compiling multiprocess programs 46
  - compiling programs 46
  - copying text between windows 39
  - creating processes 91
  - debugging setuid programs 161
  - deleting processes 106
  - disabling the auto-launch feature 62
  - displaying
    - argv 124
    - declared and allocated arrays 125
    - global variables 109
    - machine instructions 113, 130
    - memory 112
  - diving into objects 37
  - editing
    - addresses 130
    - source text 87
    - text 38
    - type strings 116
  - evaluating expressions 150
  - examining
    - core file 56
    - source and Assembler code 84
    - stack trace and stack frame 108
  - executing
    - out of function 101
    - to a selected line 100
  - exiting from TotalView 43
  - finding
    - interesting relatives 96
    - source code for functions 86
  - issuing commands 34
  - loading new executables 48, 53
  - patching programs 143
  - reloading executables 50
  - rereading symbol tables 50
  - resizing panes 30
  - restarting processes 106
  - saving action points 150
  - setting
    - breakpoints 134, 136
    - command arguments 77
    - evaluation points 141
    - input and output files 80
    - program counter (PC) 103, 104
    - search paths 75
    - signal handling mode 73
    - thread specific breakpoints 153
  - setting editor launch string 88
  - setting environment variables 79
  - single stepping 98
    - into function calls 99
    - over function call 99
  - starting processes 91
  - starting threads 91
  - starting tvdsvr 57, 63
  - stopping processes 105
  - stopping threads 105
- process
  - definition 23
- process groups window 15, 94
- process window
  - (figure) 28
  - content of 29
  - definition 15, 187

## Index

- location 169
  - process window navigation controls
    - (figure) 31
  - process window stack 32
  - \$processuid intrinsic variable 153
  - processes
    - attaching to 50, 54
    - child, definition 185
    - creating 91
    - creating new 49
    - cross debugging 13
    - deleting 106
    - detaching from 55
    - distributed 13
    - diving into 31, 37
    - error creating 160
    - executing
      - out of function 101
    - features available for controlling 16
    - go back button 30
    - groups
      - changing 95
      - definition 21, 187
      - examining 94
      - understanding 92
    - loading new executables 48, 53
    - location of 28, 69
    - multithreaded 12
    - names 93
    - native 12
    - navigating in the process window 30
    - navigating in the root window 31
    - next process button 30
    - parent, definition 187
    - previous process button 30
    - refreshing process info 105
    - reloading 50
    - remote 12, 51
    - restarting 106
    - selecting 31
    - single stepping 98
    - starting 91
    - status of 68
    - stopping 105
    - stopping and deleting 140
  - processor number 69
  - program counter (PC) 29, 103, 145
  - program group
    - changing 95
    - definition 187
    - discussion 93
  - programs
    - compiling 46
    - cross debugging 13
    - distributed 13
    - multiprocess 12
    - multithreaded 12
    - native 12
    - remote 12
    - setuid, debugging 161
  - PVM applications
    - enabling support 169
  - pvm option 178, 183
- ## Q
- queueing mouse clicks 164
  - quitting TotalView 34, 43
- ## R
- R state 70, 71
  - raising root window 34
  - Reexecute Last Save Window command 43
  - Reexecute Last Search command 41
  - registers
    - diving into 108
    - interpreting 89
  - relatives, definition 93
  - Reload Executable File command 50
  - remote debugging 57
    - (figure) 12, 58, 64
    - attaching to a process 54
    - definition 19, 188
    - launching tvdsvr 57



- loading a new executable 53
  - process location 28, 69
  - tvdsrv command syntax 181
  - remote option 48, 178
  - repainting windows 34
  - rereading symbol tables 50
  - resizing panes 30
  - resources, for .Xdefaults file 163
  - restarting processes 106
  - resuming
    - execution 91
    - processes with a signal 102
  - retracing addresses 164
  - Return (out of function) command 101
  - right mouse button 33
  - root window
    - (figure) 27
    - content of 27, 69
    - definition 15, 188
    - location 170
    - raising 34
  - rounding modes 89
  - rsh command 60
  - Run to Selection command 100
  - running state 71
- S**
- S state 70
  - same name, multiple variables with 109
  - Save All Action Points command 150
  - Save Window to File command 43
  - saving
    - action points 150, 164, 178
    - contents of windows 42
  - sb option 179
  - scroll bar
    - (figure) 35
  - scrollable multiline field
    - (figure) 36
  - scrolling
    - multiline fields 36
    - using the keyboard 35
    - windows 33, 170
    - windows and fields 34
  - Search Backward command 41
  - Search Backward for String command 41
  - Search for String command 41
  - Search Forward command 41
  - search paths
    - in .Xdefaults file 170
    - setting 75
  - search\_port option 63, 183
  - searching for strings 41, 170
  - searching for text strings 41
  - selecting
    - a line of source code 104
    - commands 33
    - Eval button 151
    - items 33
  - selecting processes 31
  - selecting threads 31
  - sending signals to program 74
  - serial line debugging
    - definition 188
  - serial option 179, 183
  - server launch command 171
  - Server Launch String command 63
  - server option 63, 184
  - Set Command Arguments command 77
  - Set Continuation Signal command 55, 102
  - Set Environment Variables command 79
  - Set PC to Absolute Value command 104
  - Set PC to Selection command 104
  - Set Process Program Group command 96

## Index

- set\_pw option 58, 184
- setting
  - breakpoints 33, 134, 136
  - command arguments 77
  - environment variables 78
  - evaluation points 33, 141
  - event points 33
  - input and output files 80
  - program counter (PC) 103
  - search path 75, 170
- setuid programs 161
- shaded box, in tag field 84
- share group
  - definition 188
  - determining members of 95
  - discussion 93
- shared libraries 176
- sharing action points 21, 138
- shm option 179
- Show All Process Groups command 94
- Show All Unattached Processes command 51
- Show Event Log Window command 80
- showing areas of memory 112
- signal\_handling\_mode option 179
- signals
  - continuing execution with 102
  - definition 188
  - handling in TotalView 72, 171, 179
- single process group window 95
- single stepping
  - cancelling 34
  - definition 188
  - high-level* single stepping commands 98
  - into function calls 99
  - over function call 99
  - slow performance 161
  - to the next instruction 98
- sizing cursor
  - (figure) 30
- sleeping state 70
- slices, of arrays 125
- source code pane 29, 159, 168, 173
- source code, examining 84
- Source Display Mode command 84
- speed when scrolling 170
- spelling corrector 42, 173
- stack
  - definition 188
  - frame
    - definition 188
    - examining 108
    - pane 29, 110
  - trace
    - definition 188
    - diving into 37
    - examining 108
    - pane 29
- standard input, and launching tvdsrvr 61
- starting
  - processes 91
  - threads 91
  - TotalView 26, 47
  - tvdsrvr 48, 57, 63
- status registers, interpreting 89
- status, of processes 68
- status, of threads 68
- Step (instruction) command 99
- Step (source line) command 99
- stepping 98
  - See also* single stepping
- STOP sign, for breakpoints 33, 134
- \$stop statement 154
- \$stopall statement 154
- stopped state 70, 71
- stopping processes 105, 140
- stopping threads 105
- \$stopprocess statement 154

- \$stopthread statement 154
  - stride, in array slices 125
  - <string> data type 123
  - strings, searching for 41, 170
  - structures 38, 119
  - subroutines, diving into 38
  - suffixes
    - of preprocessor input files 87
    - of processes in process groups 93
    - of source files 86
  - Suppress All Action Points command 149
  - Suppress All GIST command 149
  - suspended windows 152
  - swap space 160
  - symbol table, definition 189
  - symbol tables
    - rereading 50
  - Synchronous Run capability 90
  - Synchronous Stop capability 90
  - \$ystid intrinsic variable 153
- T**
- T state 70, 71
  - tab character 173
  - tag field 29, 189
  - target machine, definition 19, 189
  - template functions 99
  - text strings, searching for 41
  - thread
    - definition 23
  - thread list 26, 27, 29
  - thread specific breakpoints 153
  - threads
    - diving into 32, 37
    - executing
      - out of function 101
      - go back button 30
      - navigating in the process window 30
      - navigating in the root window 31
      - next thread button 30
      - previous thread button 30
      - selecting 31
      - single stepping 98
      - starting 91
      - status of 68
      - stopping 105
  - \$tid intrinsic variable 153
  - timeout, for launching tvdsvr 61, 171
  - totalview command 26, 47, 175
    - environment variables 78
  - transient-for windows 174
  - troubleshooting 9, 159
  - tvdsvr command 19, 181, 189
    - auto-launch feature 57
    - enabling launch of 170
    - environment variables 78
    - starting 57, 171
    - use with PVM applications 183
  - type casting 116
  - type strings
    - built-in 121
    - editing 116
    - for opaque types 124
    - parameter in .Xdefaults file 165
  - typedef datatype 119
- U**
- unattached processes window
    - content of 69
    - discussion 51
  - undive icon 114
  - undiving
    - definition 115, 189
    - from windows 115
  - unions 119
  - Unsuppress All Action Points command 149

## Index

Unsuppress All GIST command 149  
unwinding the stack 104  
Update Process Info command 105  
upper bound, of array slices 125  
using  
    mouse buttons 33

## V

Value field 151  
Variable command 109, 111, 128  
variable window  
    definition 15, 189  
    discussion 108  
    displaying 108  
    duplicating 115  
    location 165  
    to display area of memory 112  
    tracking addresses 108  
variables  
    caching 132  
    changing the value 115  
    diving into 37  
    features available 18  
    intrinsic 153  
    local, diving into 108  
<void> data type 123  
void data type 123  
void type 123

## W

windows  
    action points 146, 165  
    closing 113  
    diving into 37  
    evaluation 166  
        *See also* expression evaluation window  
    event log 80, 166  
    expression 151  
    global variables 111, 167  
    help 167

machine instructions in 113  
offset between 166  
override-redirect 169  
problems with 159  
process 15, 28, 169  
process groups 15, 94  
repainting 34  
root 15, 27, 170  
saving contents of 42  
single process group 95  
suspended 152  
transient-for 174  
unattached processes 51  
variable 15, 108, 109, 112, 113, 165

## X

X Window System 39  
.Xdefaults file 163  
xterm  
    launching tvdsvr from 61  
    problems with 159

## Y

yacc utility, file suffixes 87

## Z

Z state 70  
zombie state 70