TotalView

User Guide

June 2004

Version 6.5

Copyright © 1999–2004 by Etnus LLC. All rights reserved.
Copyright © 1998–1999 by Etnus, Inc.
Copyright © 1996–1998 by Dolphin Interconnect Solutions, Inc.
Copyright © 1993–1996 by BBN Systems and Technologies, a division of BBN Corporation.

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

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

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

TotalView and Etnus are registered trademarks of Etnus LLC.

TotalView uses a modified version of the Microline widget library. Under the terms of its license, you are entitled to use these modifications. The source code is available at http://www.etnus.com/Products/TotalView/developers.

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

Book Overview

Part I – Introduction

1 Discovering TotalView 3
2 About Threads, Processes, and Groups 15

Part II – Setting Up

3 Setting Up a Debugging Session 35
4 Setting Up Remote Debugging Sessions 63
5 Setting Up Parallel Debugging Sessions 77

Part III – Using the GUI

6 Using TotalView Windows 123
7 Visualizing Programs and Data 137

Part IV – Using the CLI

8 Seeing the CLI at Work 157
9 Using the CLI 165

Part V – Debugging

10 Debugging Programs 179
11 Using Groups, Processes, and Threads 205
12 Examining and Changing Data 235
13 Examining Arrays 279
14 Setting Action Points 293

Glossary 331

Contents

About This Book

How to Use This Book xxi
Using the CLI xxii
Audience xxii
Conventions xxiii
TotalView Documentation xxiv
Contacting Us xxiv

Part I – Introduction

1 Discovering TotalView

Getting Started 3
Starting TotalView 4
What About Print Statements? 4
Examining Data 7
Examining Arrays 8
Seeing Groups of Variables 10
Setting Watchpoints 10
Debugging Multiprocess and Multithreaded Programs 10
Supporting Multiprocess and Multithreaded Programs 12
Using Groups and Barriers 13
Introducing the CLI 13
What's Next 14

2 About Threads, Processes, and Groups

A Couple of Processes 15
Threads 17
Complicated Programming Models 18
Types of Threads 20
Organizing Chaos 22
Creating Groups 26
Simplifying What You're Debugging 30

Part II – Setting Up

3 Setting Up a Debugging Session

Compiling Programs 35
Using File Extensions 36
Starting TotalView 36
Initializing TotalView 38
Exiting from TotalView 40
Loading Executables 40
Loading Remote Executables 42
Attaching to Processes 42
Attaching Using the Unattached Page 43
Attaching Using File > New Program or dattach 44
Detaching from Processes 45
Examining Core Files 45
Viewing Process and Thread States 46
Seeing Attached Process States 47
Seeing Unattached Process States 48
Handling Signals 48
Setting Search Paths 50
Setting Command Arguments 53
Setting Input and Output Files 53
Setting Preferences 54
Setting Preferences, Options, and X Resources 59
Setting Environment Variables 60
Monitoring TotalView Sessions 61

4 Setting Up Remote Debugging Sessions

Setting Up and Starting the TotalView Debugger Server 63
Setting Single-Process Server Launch Options 64
Setting Bulk Launch Window Options 65
Starting the Debugger Server Manually 67
Using the Single-Process Server Launch Command 68
Bulk Server Launch Setting on an SGI MIPs Computer 69
Setting Bulk Server Launch on an HP Alpha Computer 70
Setting Bulk Server Launch on an IBM RS/6000 AIX Computer 71
Disabling Autolaunch 71
Changing the Remote Shell Command 71
Changing Arguments 72
Autolaunching Sequence 72
Debugging Over a Serial Line 74
Starting the TotalView Debugger Server 74
Starting TotalView on a Serial Line 75
Using the New Program Window 75

5 Setting Up Parallel Debugging Sessions

Debugging MPICH Applications 78
Starting TotalView on an MPICH Job 78
Attaching to an MPICH Job 80
Using MPICH P4 procgroup Files 81

Debugging HP Tru64 Alpha MPI Applications 81
Starting TotalView on an HP Alpha MPI Job 81
Attaching to an HP Alpha MPI Job 82
Debugging HP MPI Applications 82
Starting TotalView on an HP MPI Job 82
Attaching to an HP MPI Job 83
Debugging IBM MPI Parallel Environment (PE) Applications 83
Preparing to Debug a PE Application 83
Using Switch-Based Communications 83
Performing a Remote Login 84
Setting Timeouts 84
Starting TotalView on a PE Program 84
Setting Breakpoints 84
Starting Parallel Tasks 85
Attaching to a PE Job 85
Attaching from a Node Running poe 85
Attaching from a Node Not Running poe 86
Debugging LAM/MPI Applications 86
Debugging QSW RMS Applications 87
Starting TotalView on an RMS Job 87
Attaching to an RMS Job 87
Debugging SGI MPI Applications 88
Starting TotalView on an SGI MPI Job 88
Attaching to an SGI MPI Job 88
Debugging Sun MPI Applications 89
Attaching to a Sun MPI Job 89
Displaying the Message Queue Graph Window 90
Displaying the Message Queue 91
About the Message Queue Display 91
Using Message Operations 92
Diving on MPI Processes 92
Diving on MPI Buffers 93
About Pending Receive Operations 93
About Unexpected Messages 93
About Pending Send Operations 94
Troubleshooting MPI Applications 94
Debugging OpenMP Applications 95
Debugging OpenMP Programs 95
About TotalView OpenMP Features 96
About OpenMP Platform Differences 97
Viewing OpenMP Private and Shared Variables 97
Viewing OpenMP THREADPRIVATE Common Blocks 98
Viewing the OpenMP Stack Parent Token Line 100
Debugging Global Arrays Applications 100
Debugging PVM (Parallel Virtual Machine) and DPVM Applications 103
Supporting Multiple Sessions 104
Setting Up ORNL PVM Debugging 104
Starting an ORNL PVM Session 104
Starting a DPVM Session 105
Automatically Acquiring PVM/DPVM Processes 106
Attaching to PVM/DPVM Tasks 107

About Reserved Message Tags 108
Cleaning Up Processes 108
Debugging Shared Memory (SHMEM) Code 109
Debugging UPC Programs 110
Invoking TotalView 110
Viewing Shared Objects 110
Displaying Pointer to Shared Variables 112
Debugging Parallel Applications Tips 113
Attaching to Processes 113
Parallel Debugging Tips 116
MPICH Debugging Tips 118
IBM PE Debugging Tips 118

Part III – Using the GUI

6 Using TotalView Windows

Using the Mouse Buttons 123
Using the Root Window 124
Using the Attached Page 124
Using the Unattached Page 127
Using the Groups Page 128
Using the Log Page 128
Using the Process Window 129
Viewing the Assembler Version of Your Code 131
Diving into Objects 132
Resizing and Positioning Windows and Dialog Boxes 134
Editing Text 135
Saving the Contents of Windows 136

7 Visualizing Programs and Data

Displaying Your Program's Call Tree 137
Visualizing Array Data 139
How the Visualizer Works 139
Configuring TotalView to Launch the Visualizer 140
Setting the Visualizer Launch Command 141
Viewing Data Types in the Visualizer 142
Viewing Data 142
Visualizing Data Manually 143
Visualizing Data Programmatically 143
Using the Visualizer 145
Using Directory Window Commands 145
Using Data Windows Commands 146
Using the Graph Window 147
Displaying Graphs 148
Manipulating Graphs 149
Using the Surface Window 149
Displaying Surface Data 151
Manipulating Surface Data 152
Launching the Visualizer from the Command Line 152

Part IV – Using the CLI

8 Seeing the CLI at Work

Setting the CLI EXECUTABLE_PATH Variable 157
Initializing an Array Slice 158
Printing an Array Slice 159
Writing an Array Variable to a File 160
Automatically Setting Breakpoints 161

9 Using the CLI

About the Tcl and the CLI 165
About The CLI and TotalView 166
Using the CLI Interface 166
Starting the CLI 167
Startup Example 168
Starting Your Program 168
About CLI Output 170
'more' Processing 171
Using Command Arguments 171
Using Namespaces 172
About CLI Prompt 172
Using Built-in and Group Aliases 173
How Parallelism Affects Behavior 174
Types of IDs 174
Controlling Program Execution 175
Advancing Program Execution 175
Using Action Points 176

Part V – Debugging

10 Debugging Programs

Searching and Looking For Program Elements 179
Searching for Text 180
Looking for Functions and Variables 180
Finding the Source Code for Functions 181
Resolving Ambiguous Names 182
Finding the Source Code for Files 182
Resetting the Stack Frame 182
Editing Source Text 183
Manipulating Processes and Threads 183
Using the Toolbar to Select a Target 184
Stopping Processes and Threads 184
Updating Process Information 185
Holding and Releasing Processes and Threads 185
Using Barrier Points 187
Holding Problems 188
Examining Groups 188
Displaying Groups 190

Placing Processes in Groups 190
Starting Processes and Threads 190
Creating a Process Without Starting It 191
Creating a Process by Single-Stepping 191
Stepping and Setting Breakpoints 192
Using Stepping Commands 193
Stepping into Function Calls 194
Stepping Over Function Calls 195
Executing to a Selected Line 195
Executing Out of a Function 196
Continuing with a Specific Signal 196
Deleting (Killing) Programs 197
Restarting Programs 197
Checkpointing 197
Fine-Tuning Shared Library Use 198
Preloading Shared Libraries 198
Controlling Which Symbols TotalView Reads 200
Specifying Which Libraries are Read 200
Reading Excluded Information 201
Setting the Program Counter 202
Interpreting the Status and Control Registers 203

11 Using Groups, Processes, and Threads

Defining the GOI, POI, and TOI 205
Setting a Breakpoint 206
Stepping (Part I) 207
Understanding Group Widths 208
Understanding Process Width 208
Understanding Thread Width 208
Using Run To and duntil Commands 209
Using P/T Set Controls 210
Setting Process and Thread Focus 211
Understanding Process/Thread Sets 211
Specifying Arenas 213
Specifying Processes and Threads 213
Defining the Thread of Interest (TOI) 213
About Process and Thread Widths 214
Specifier Examples 215
Setting Group Focus 216
Specifying Groups in P/T Sets 217
About Arena Specifier Combinations 218
'All' Does Not Always Mean 'All' 220
Setting Groups 222
Using the g Specifier: An Extended Example 222
Merging Focuses 225
Naming Incomplete Arenas 225
Naming Lists with Inconsistent Widths 226
Stepping (Part II): Examples 227
Using P/T Set Operators 228
Using the P/T Set Browser 229
Using the Group Editor 231

12 Examining and Changing Data

Changing How Data is Displayed 235
Displaying STL Variables 235
Changing Size and Precision 238
Displaying Variables 238
Displaying Program Variables 239
Seeing Structure Information 240
Seeing More Information 241
Displaying Variables in the Current Block 241
Viewing a Variable in Different Scopes as Your Program Executes 242
Scoping Issues 242
Browsing for Variables 243
Displaying Local Variables and Registers 244
Dereferencing Variables Automatically 245
Displaying Areas of Memory 246
Changing Types to Display Machine Instructions 247
Displaying Machine Instructions 248
Rebinding the Variable Window 248
Closing Variable Windows 249
Diving in Variable Windows 249
Displaying an Array of Structure's Elements 251
Changing What the Variable Window Displays 253
Viewing a List of Variables 254
Entering Variables and Expressions 254
Entering Expressions into the Expression Column 255
Using the Expression List with Multiprocess/Multithreaded Programs 256
Reevaluating, Reopening, Rebinding, and Restarting 257
Seeing More Information 257
Sorting, Reordering, and Editing 258
Changing the Values of Variables 259
Changing a Variable's Data Type 260
Displaying C Data Types 260
Viewing Pointers to Arrays 261
Viewing Arrays 261
Viewing typedef Types 262
Viewing Structures 262
Viewing Unions 263
Viewing Built-In Types 263
Viewing Opaque Data (<opaque> Data Type) 265
Viewing Character Arrays (<string> Data Type) 265
Viewing Areas of Memory (<void> Data Type) 265
Viewing Instructions (<code> Data Type) 266
Type-Casting Examples 266
Displaying Declared Arrays 266
Displaying Allocated Arrays 266
Displaying the argv Array 266
Changing the Address of Variables 267
Displaying C++ Types 268
Viewing Classes 268
Displaying Fortran Types 269

Displaying Fortran Common Blocks 269
Displaying Fortran Module Data 269
Debugging Fortran 90 Modules 271
Viewing Fortran 90 User-Defined Types 272
Viewing Fortran 90 Deferred Shape Array Types 272
Viewing Fortran 90 Pointer Types 273
Displaying Fortran Parameters 273
Displaying Thread Objects 274
Scoping and Symbol Names 274
Qualifying Symbol Names 276

13 Examining Arrays

Examining and Analyzing Arrays 279
Displaying Array Slices 279
Using Slices and Strides 280
Using Slices in the Lookup Variable Command 282
Filtering Array Data Overview 283
Filtering Array Data 284
Filtering by Comparison 284
Filtering for IEEE Values 285
Filtering a Range of Values 286
Creating Array Filter Expressions 287
Using Filter Comparisons 288
Sorting Array Data 288
Obtaining Array Statistics 289
Displaying a Variable in all Processes or Threads 290
Visualizing Array Data 292
Visualizing a Laminated Variable Window 292

14 Setting Action Points

About Action Points 293
Setting Breakpoints and Barriers 295
Setting Source-Level Breakpoints 295
Choosing Source Lines 295
Setting and Deleting Breakpoints at Locations 295
Displaying and Controlling Action Points 297
Disabling Action Points 298
Deleting Action Points 298
Enabling Action Points 298
Suppressing Action Points 299
Setting Machine-Level Breakpoints 299
Setting Breakpoints for Multiple Processes 300
Setting Breakpoints When Using the fork()/execve() Functions 302
Debugging Processes That Call the fork() Function 302
Debugging Processes that Call the execve() Function 302
Example: Multiprocess Breakpoint 303
Setting Barrier Points 303
About Barrier Breakpoint States 303
Setting a Barrier Breakpoint 304
Creating a Satisfaction Set 305
Hitting a Barrier Point 305

Releasing Processes from Barrier Points 306
Deleting a Barrier Point 306
Changing Settings and Disabling a Barrier Point 306
Defining Eval Points and Conditional Breakpoints 306
Setting Eval Points 308
Creating Conditional Breakpoint Examples 308
Patching Programs 309
Branching Around Code 309
Adding a Function Call 309
Correcting Code 310
About Interpreted Compiled Expressions 310
About Interpreted Expressions 310
About Compiled Expressions 311
Allocating Patch Space for Compiled Expressions 311
Allocating Dynamic Patch Space 311
Allocating Static Patch Space 312
Using Watchpoints 313
Using Watchpoints on Different Architectures 314
Creating Watchpoints 315
Displaying Watchpoints 315
Watching Memory 316
Triggering Watchpoints 316
Using Multiple Watchpoints 316
Copying Previous Data Values 317
Using Conditional Watchpoints 317
Saving Action Points to a File 319
Evaluating Expressions 319
Writing Code Fragments 322
Using TotalView Variables 322
Using Built-In Statements 323
About Supported C Constructs 324
Data Types and Declarations 324
Statements 325
Supported Fortran Constructs 325
Fortran Data Types and Declarations 326
Fortran Statements 326
Writing Assembler Code 326

Glossary 331

Index 343

Figures

Chapter 1: Discovering TotalView

1. The Process Window 4
2. Action Point Properties Dialog Box 5
3. Setting Conditions 6
4. Patching Using an Eval Point 7
5. Diving on a Structure and an Array 8
6. Slicing and Filtering Arrays 9
7. Visualizing an Array 9
8. Tools > Expression List Window 10
9. The Root Window 11
10. Process and Thread Switching Buttons 11
11. A Laminated Variable Window 12
12. A Message Queue Graph 12
13. Toolbar With Pulldown 13
14. The Root Window Group Page 13

Chapter 2: About Threads, Processes, and Groups

15. A Uniprocessor 16
16. A Program and Daemons 16
17. Mail Using Daemons to Communicate 16
18. Two Computers Working on One Problem 17
19. Threads 18
20. Four-Processor Computer 19
21. Four Processors on a Network 19
22. Threads (again) 20
23. User and Service Threads 21
24. User, Service, and Manager Threads 21
25. Five-Processes: Their Control and Share Groups 23
26. Five Processes: Adding Workers and Lockstep Groups 24
27. Five Processes and Their Groups on Two Computers 25
28. Step 1: A Program Starts 26
29. Step 2: Forking a Process 26
30. Step 3: Creating a Process using exec() 27
31. Step 5: Creating a Second Version 28

32. Step 6: Creating a Remote Process 28
33. Step 7: Creating a Thread 29
34. Step 8: Hitting a Breakpoint 29
35. Step 9: Stepping the Lockstep Group 30

Chapter 3: Setting Up a Debugging Session

36. Startup and Initialization Sequence 39
37. File > Exit Dialog Box 40
38. File > New Program Dialog Box 41
39. Unattached Page 43
40. File > New Program Dialog Box 44
41. Thread > Continuation Signal Dialog Box 45
42. Attached Page Showing Process and Thread Status 47
43. Process and Thread Labels in the Process Window 47
44. File > Signals Dialog Box 49
45. File > Search Path Dialog Box 51
46. Select Directory Dialog Box 52
47. Process > Startup Parameters Dialog Box: Arguments Page 53
48. Process > Startup Parameters Dialog Box: Standard I/O Page 54
49. File > Preferences Dialog Box: Options Page 55
50. File > Preferences Dialog Box: Action Points Page 55
51. File > Preferences Dialog Box: Launch Strings Page 56
52. File > Preferences Dialog Box: Bulk Launch Page 56
53. File > Preferences Dialog Box: Dynamic Libraries Page 57
54. File > Preferences Dialog Box: Parallel Page 57
55. File > Preferences Dialog Box: Fonts Page 58
56. File Preferences Dialog Box: Formatting Page 58
57. File > Preferences Dialog Box: Pointer Dive Page 59
58. Process > Startup Parameters Dialog Box: Environment Page 60
59. Root Window Log Page 61

Chapter 4: Setting Up Remote Debugging Sessions

60. File > Preferences: Launch Strings Page 64
61. File > Preferences: Bulk Launch Page 66
62. Manual Launching of Debugger Server 68
63. Launching tvdsvr 73
64. Multiple tvdsvr Processes 73
65. TotalView Debugging Session Over a Serial Line 74
66. File > New Program Dialog Box 75

Chapter 5: Setting Up Parallel Debugging Sessions

67. Root Window: Unattached Page 80
68. Tools > Message Queue Graph Window 90
69. Message Queue Window 92
70. Message Queue Window Showing Pending Receive Operation 93
71. Sample OpenMP Debugging Session 96
72. OpenMP Shared Variable 98
73. OpenMP THREADPRIVATE Common Block Variables 99
74. OpenMP Stack Parent Token Line 100

75. Question Window for Global Arrays Program 101
76. Tools > Global Arrays Window 102
77. Fortran and C Variable Windows 102
78. Casting in Fortran 103
79. PVM Tasks and Configuration Window 108
80. SHMEM Sample Session 109
81. A Sliced UPC Array 111
82. UPC Variable Window Showing Nodes 111
83. Visualizing a UPC Variable 112
84. A Pointer to a Shared Variable 112
85. Pointer to a Shared Variable 113
86. Group > Attach Subset Dialog Box 114
87. Stop Before Going Parallel Question Box 115
88. File > Preferences: Parallel Page 115

Chapter 6: Using TotalView Windows

89. Root Window Attached Page 125
90. Root Window Showing Two Host Computers 125
91. Two Views of the Root Window 126
92. Sorted and Aggregated Root Window 127
93. Root Window Unattached Page 127
94. Root Window Groups Page 128
95. Root Window Log Page 128
96. A Process Window 129
97. Line Numbers with Stop Icon and PC Arrow 130
98. Address Only (Absolute Addresses) 131
99. Assembly Only (Symbolic Addresses) 132
100. Both Source and Assembler (Symbolic Addresses) 132
101. Nested Dive 133
102. Backward and Forward Buttons 134
103. Resizing (and Its Consequences) 135
104. Editing Cursor 136
105. File > Save Pane Dialog Box 136

Chapter 7: Visualizing Programs and Data

106. Tools > Call Tree Dialog Box 138
107. TotalView Visualizer Relationships 139
108. TotalView Visualizer Connection 140
109. File > Preferences Launch Strings Page 141
110. A Three-Dimensional Array Sliced into Two Dimensions 142
111. A Variable Window 143
112. Sample Visualizer Windows 145
113. A 2D and a 3D Visualizer Window 146
114. Rotating and Querying 147
115. Visualizer Graph Data Window 148
116. Graph Options Dialog Box 148
117. A 2D Point Display 149
118. A 2D Surface Display 150
119. A 3D Surface Display 150

120. Surface Options Dialog Box 151
121. Zooming and Rotating About an Axis 153

Chapter 9: Using the CLI

122. The CLI and TotalView 166
123. CLI xterm Window 167

Chapter 10: Debugging Programs

124. Edit > Find Dialog Box 180
125. View > Lookup Variable Dialog Box 180
126. Ambiguous Function Dialog Box 181
127. View > Lookup Function Dialog Box 181
128. Undive/Dive Controls 181
129. Ambiguous Function Dialog Box 182
130. View > Lookup Function Dialog Box 183
131. The Toolbar 184
132. Running To Barriers 188
133. Control and Share Groups Example 189
134. Root Window: Groups Page 190
135. Action Point and Addresses Dialog Boxes 192
136. Ambiguous Address Dialog Box 193
137. Stepping Illustrated 193
138. Thread > Continuation Signal Dialog Box 196
139. Create Checkpoint and Restart Checkpoint Dialog Boxes 198
140. Tools > Debugger Loaded Libraries Dialog Box 199
141. Stopping to Set a Breakpoint Question Box 199
142. File > Preferences: Dynamic Libraries Page 200
143. Load All Symbols in Stack Context menu 202

Chapter 11: Using Groups, Processes, and Threads

144. The P/T Set Control in the Process Window 210
145. The P/T Set Control in the Tools > Evaluate Window 210
146. Width Specifiers 215
147. A P/T Set Browser Window 230
148. P/T Set Browser Windows 232
149. Group > Edit Group Dialog Box 233

Chapter 12: Examining and Changing Data

150. An Untransformed Map 236
151. A Transformed Map 236
152. List and Vector Transformations 237
153. File > Preferences Formatting Page 238
154. Variable Window for a Global Variable 240
155. The Details Dialog Box 241
156. Displaying Scoped Variables 241
157. Program Browser and Variable Windows (Part 1) 243
158. Program Browser and Variable Window (Part 2) 244
159. Diving on Local Variables and Registers 245
160. File > Preferences Pointer Dive Page 246
161. Variable Window for an Area of Memory 247

162. Variable Window with Machine Instructions 248
163. Casting to Code 249
164. Undive/Redive Buttons 250
165. Nested Dives 250
166. Displaying a Fortran Structure 251
167. Displaying C Structures and Arrays 252
168. Dive in All 253
169. The Tools > Expression List Window 254
170. A Context Menu 254
171. Expression List Window Context Menu 255
172. The Tools > Expression List Window 255
173. The Tools > Expression List Window Showing Column Selector 257
174. Using an Expression to Change a Value 259
175. Displaying a Union 263
176. Editing the argv Argument 267
177. Displaying C++ Classes That Use Inheritance 268
178. Diving on a Common Block List in the Stack Frame Pane 270
179. Fortran Modules Window 271
180. Fortran 90 User-Defined Type 272
181. Fortran 90 Pointer Value 273
182. Thread Objects Page on an IBM AIX Computer 275
183. Details Dialog Box 276

Chapter 13: Examining Arrays

184. Stride Displaying the Four Corners of an Array 281
185. Fortran Array with Inverse Order and Limited Extent 282
186. Variable Window for *small_array* 283
187. Array Data Filtering by Comparison 285
188. Array Data Filtering for IEEE Values 286
189. Array Data Filtering by Range of Values 287
190. Sorted Variable Window 288
191. Array Statistics Window 289
192. Laminated Scalar Variable 291
193. Laminated Array and Structure 292

Chapter 14: Setting Action Points

194. Action Point Symbols 294
195. Setting Breakpoints on Multiple Similar Addresses and on Processes 296
196. Action Point > At Location Dialog Box 296
197. Ambiguous Function Dialog Box 297
198. Action Point > Properties Dialog Box 297
199. Breakpoint at Assembler Instruction 299
200. PC Arrow Over a Stop Icon 300
201. Action Point > Properties Dialog Box 300
202. File > Preferences: Action Points Page 301
203. Action Point > Properties Dialog Box 304
204. Stopped Execution of Compiled Expressions 311
205. Tools > Watchpoint Dialog Boxes 315
206. Tools > Evaluate Dialog Box 320

207. Waiting to Complete Message Box 321
208. Evaluating Information in Multiple Processes 321
209. Using Assembler Expressions 327

About This Book

This book describes how to use TotalView®, a source- and machine-level debugger for multiprocess, multithreaded programs. It assumes that you are familiar with programming languages, the UNIX operating systems, and the processor architecture of the platform on which you are running TotalView and your program.

This user guide combines information for two TotalView debuggers. One uses Motif to present windows and dialog boxes. The other runs in an xterm-like window and requires you to type commands. This book emphasizes the Motif interface, as it is easier to use. After you see what you can do using Motif, you will know what you can do using the command interface.

TotalView doesn't change much from platform to platform. Differences between platforms are mentioned.

How to Use This Book

The information in this book is presented in five parts:

- **I: Introduction**

  This part contains an overview of some of TotalView features and an introduction to the TotalView process/thread model. This section gives you a feel for what TotalView can do.

- **II: Setting Up**

  This part describes how to configure TotalView. No one will ever use all of the information in this part.

  Chapter 3 contains general information. Chapters 4 and 5 tell you how to get your programs running under the control of TotalView. Chapter 4 explains how to get the TotalView Debugger Server (**tvdsvr**) running and if you're reconfiguring how the **tvdsvr** gets launched. In most cases, the TotalView default works fine and you won't need this information.

Chapter 5 looks at the high performance computing environments such as MPICH, OpenMP, global arrays, and the like. You should go to the table of contents and find the section that has the information you need.

- **III: Using the GUI**

  The chapters in this section describe some of the TotalView windows and how you use them. They also describe tools such as the **Visualizer** and the **Call Tree** that help you analyze what your program is doing.

- **IV: Using the CLI**

  The chapters in this section explain the basics of using the Command Line Interface (CLI) for debugging.

- **V: Debugging**

  In many ways, most of what precedes this part of the book is introductory material. This part explains how to examine your program and its data. It contains information on setting the action points that allow you to stop and monitor your program's execution.

  Chapter 11 is a detailed examination of the TotalView group, process, and thread model. Having a better understanding of this model makes it easier to debug multiprocess and multithreaded programs.

Using the CLI

To use the Command Line Interface (CLI), you need to be familiar with and have experience debugging programs with the TotalView GUI. CLI commands are embedded within a Tcl interpreter, so you get better results if you are familiar with Tcl. If you don't know Tcl, you can still use the CLI, but you lose the ability to program actions that Tcl provides; for example, CLI commands operate on a set of processes and threads. By using Tcl commands, you can save this set and apply this saved set to other commands.

The following books are excellent sources of Tcl information:

- Ousterhout, John K. *Tcl and the Tk Toolkit*. Reading, Mass.: Addison Wesley, 1997.
- Welch, Brent B. *Practical Programming in Tcl & Tk*. Upper Saddle River, N.J.: Prentice Hall PTR, 1997.

There is also a rich set of resources available on the Web. A very good s starting point is **http://www.tcltk.com**.

The fastest way to gain an appreciation of the actions performed by CLI commands is to review Chapter 1 of the *TotalView Reference Guide*, which contains an overview of CLI commands.

Audience

Many of you are very sophisticated programmers, having a tremendous knowledge of programming and its methodologies, and almost all of you have used other debuggers and have developed your own techniques for debugging the programs that you write.

We know you are an expert in your area, whether it be threading, high-performance computing, client/server interactions, and the like. So, rather than telling you about what you're doing, this book tells you about TotalView.

TotalView is a rather easy-to-use product. Nonetheless, we can't tell you how to use TotalView to solve your problems because your programs are unique and complex, and we can't anticipate what you want to do.

Information about what you do with a dialog box or the kinds of data you can type is in the online Help. If you prefer, an HTML version of this information is available on our Web site. If you have purchased TotalView, you can also post this HTML documentation on your intranet.

Conventions

The following table describes the conventions used in this book:

Convention	Meaning
[ ]	Brackets are used when describing parts of a command that are optional.
arguments	In a command description, text in italics represents information you type. Elsewhere, italics is used for emphasis.
Dark text	In a command description, **dark text** represents keywords or options that you must type exactly as displayed. Elsewhere, it represents words that are used in a programmatic way rather than their normal way.
`Example text`	In program listings, this indicates that you are seeing a program or something you'd type in response to a shell or CLI prompt. If this text is in bold, it's indicating that what you're seeing is what you'll be typing. If you're viewing this information online, example text is in color.
	This graphic symbol indicates that the information that follows—*which is printed in italics*—is a note. This information is an important qualifier to what you just read.
	This graphic symbol indicates that a feature is only available in the GUI. If you see it on the first line of a section, all the information in the section is just for GUI users. When it is next to a paragraph, just that sentence or paragraph applies to the GUI.
CLI:	The primary emphasis of this book is the GUI. It shows the windows and dialog boxes that you use. This symbol tells you the CLI command you use to do the same thing.

TotalView Documentation

The following table describes other TotalView documentation:

Title	Contents	Online Help	HTML	PDF	Print
TotalView User Guide	Describes how to use the TotalView GUI and the CLI; this is the most used of all the TotalView books.	✔	✔	✔	✔
TotalView New Features	Describes new features added to TotalView.	✔	✔	✔	
Debugging Memory Using TotalView	Is a combined user and reference guide describing how to find your program's memory problems.	✔	✔	✔	✔
TotalView Reference Guide	Contains descriptions of CLI commands, how you run TotalView, and platform-specific information.	✔	✔	✔	✔
TotalView QuickView	Presents what you need to know to get started using TotalView.		✔		✔
TotalView Commands	Defines all TotalView GUI commands—this is the online Help.	✔	✔	✔	
TotalView Installation Guide	Contains the procedures to install TotalView and the FLEXlm license manager.	✔	✔	✔	
TotalView Release Notes	Lists known bugs and other information related to the current release.		✔	✔	
Platforms and System Requirements	Lists the platforms upon which TotalView runs and the compilers it supports.	✔	✔	✔	

Contacting Us

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

Our Internet E-Mail address for support issues is:

support@etnus.com

For documentation issues, the address is:

documentation@etnus.com

Our phone numbers are:

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

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

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

Part I: Introduction

This part of the *TotalView Users Guide* contains two chapters.

Chapter 1: Discovering TotalView
Presents an overview of what TotalView is and the ways in which it can help you debug programs. If you haven't used TotalView before, reading this chapter lets you know what TotalView can do for you.

Chapter 2: About Threads, Processes, and Groups
Defines the TotalView model for organizing processes and threads. While most programmers have an intuitive understanding of what their programs are doing, debugging multiprocess and multithreaded programs requires an exact knowledge of what's being done. This chapter begins a two-part look at the TotalView process/thread model. This chapter contains introductory information. Chapter 11: "*Using Groups, Processes, and Threads*" on page 205 contains information on actually using these concepts.

Discovering TotalView

1

The Etnus TotalView debugger is a powerful, sophisticated, and programmable tool that lets you debug, analyze, and tune the performance of complex serial, multiprocessor, and multithreaded programs.

If you want to jump in and get started quickly, go to our web site at http://www.etnus.com and select the "Getting Started" link. (It's in the Quick Links area on the right near the bottom.)

This chapter contains the following sections:

- "*Getting Started*" on page 3
- "*Debugging Multiprocess and Multithreaded Programs*" on page 10
- "*Using Groups and Barriers*" on page 13
- "*Introducing the* CLI" on page 13
- "*What's Next*" on page 14

Getting Started

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

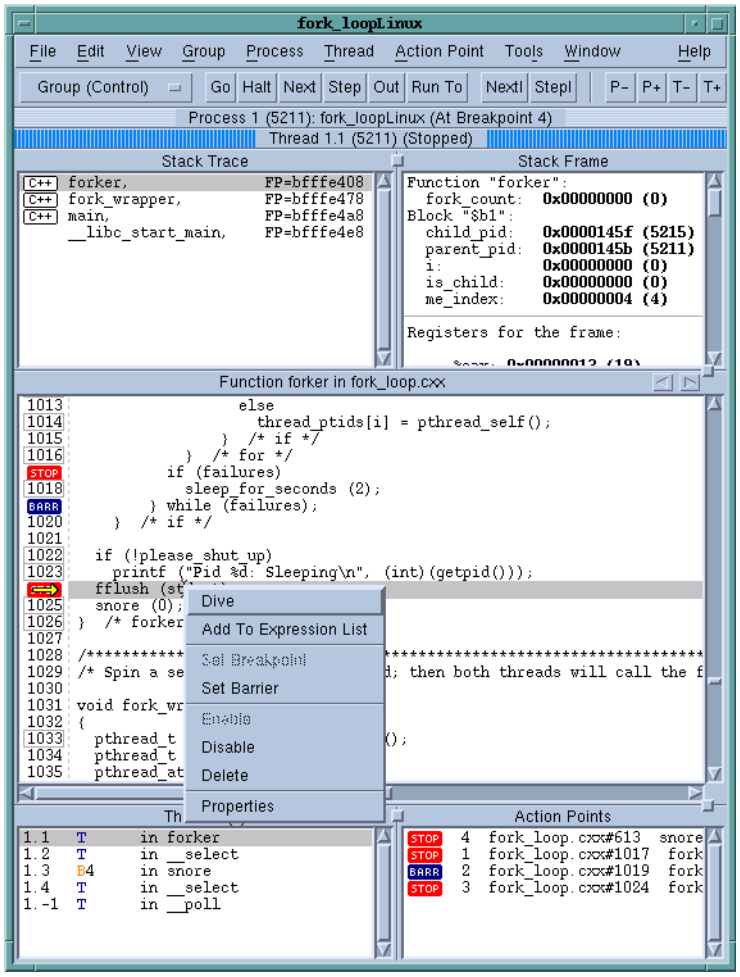
- You use the **–g** option when you compile your program.
- You start your program under the debugger's control.
- You set a breakpoint.
- You examine data.

The way you do these things is similar to the way you do things in other debuggers. Where TotalView differs from what you're used to is in its raw power, the breadth of commands available, and its native ability to handle multiprocess, multithreaded programs.

Starting TotalView

After execution begins—by typing something like **totalview** *programname*—the TotalView five-paned Process Window appears.

Figure 1: The Process Window



You can start program execution in several ways. Perhaps the easiest way is to click the **Step** button in the toolbar. This gets your program started, which means that the initialization performed by the program gets done but no statements are executed.

You can scroll your program to find where you want it to run to, select the line, then click on **Run To** in the toolbar. Or you can click on the line number, which tells TotalView to create a breakpoint on that line, and then click the **Go** button in the toolbar.

If your program is large, and usually it will be, you can use **Edit > Find** to locate the line for you. Or, if you want to stop execution when your program reaches a subroutine, use **Action Point > At Location** to set a breakpoint before you click **Go**.

What About Print Statements?

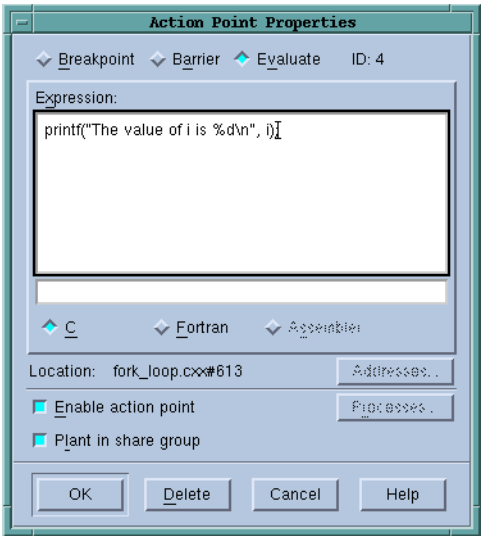
Most programmers learn to debug by using print statements. That is, you insert lots of **printf()** or **PRINT** statements in your code and then inspect what gets written. The problem with this is that every time you want to add

a new statement, you need to recompile your program. Even worse is that in a multiprocess, multithreaded program, what gets printed is probably not in the right order. While TotalView is much more sophisticated than this, you can still use **printf()** statements if that's your style.

If you don't want to change the way you've been debugging, you can add a breakpoint that prints information. Do this by right-clicking on a breakpoint and then select **Properties** from the context menu. (This context menu was shown in the previous figure.) TotalView then opens its **Action Point Properties** Dialog Box, which is shown in next figure.

*In TotalView, a breakpoint is called an **action point**. This is because these breakpoints can be much more powerful than the breakpoints you've used in other debuggers.*

Figure 2: Action Point Properties Dialog Box

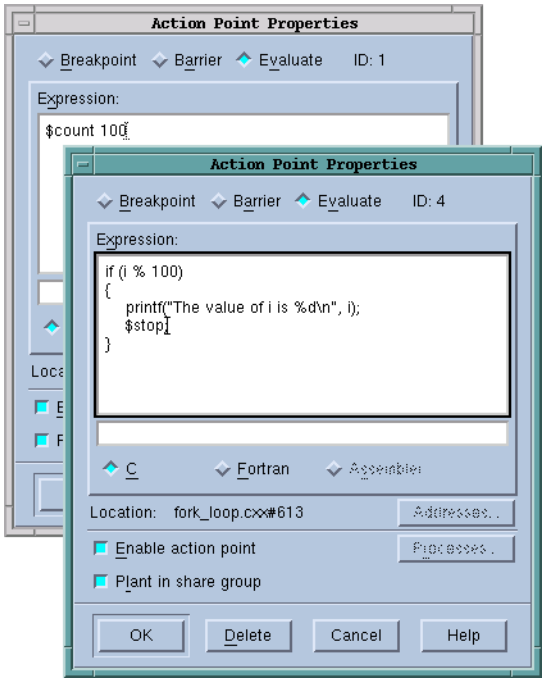


You can add any code you want to a breakpoint. Because there's code associated with this breakpoint, it is called an *eval point*. Here's where TotalView does things a little differently. When your program reaches this eval point, TotalView executes the code you've entered. In this case, TotalView prints the value of **i**.

Eval points do exactly what you tell them to do. In this case, because you didn't tell TotalView to stop executing, it keeps on going. In other words, you don't have to stop program execution just to see data. You can, of

course, tell TotalView to stop. The following figure shows two eval points that stop execution. (One of them does something else as well.)

Figure 3: Setting Conditions

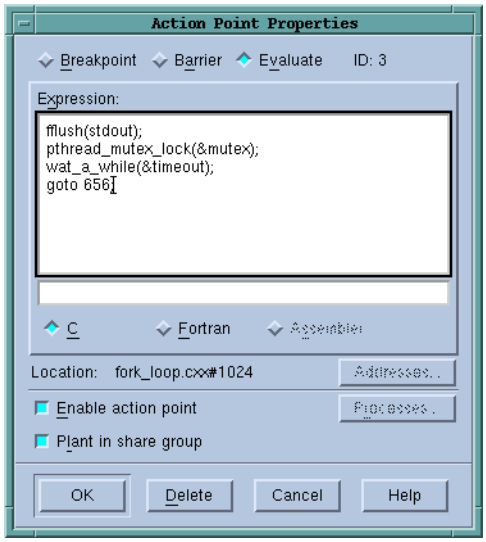


The eval point in the foreground uses programming language statements and a built-in TotalView function to stop a loop every 100 iterations. It also prints the value of **i**. In contrast, the eval point in the background just stops the program every 100 times a statement gets executed.

Eval points let you patch your programs and route around code that you want replaced. For example, suppose you need to change a bunch of statements. Just add these statements to an action point, then add a **goto** statement that jumps over the code you no longer want executed. For

example, the eval point shown in the following figure tells TotalView to execute three statements and then skip to line 656.

Figure 4: Patching Using an Eval Point



Examining Data

Programmers use print statements as an easy way to examine data. They usually do this because their debugger doesn't have sophisticated ways of showing information. In contrast, Chapter 12, "*Examining and Changing Data*," on page 235 and Chapter 13, "*Examining Arrays*," on page 279 explain how you can display data values with TotalView. In addition, Chapter 7, "*Visualizing Programs and Data*," on page 137 describes how to visualize your data in a graphical way.

Because data is difficult to see, the Stack Frame Pane (the pane in the upper right corner of the Process Window has a list of all variables that exist in your current routine. If the value is simple, you can see its value in this pane.

If the value isn't simple, just dive on the variable to get more information.

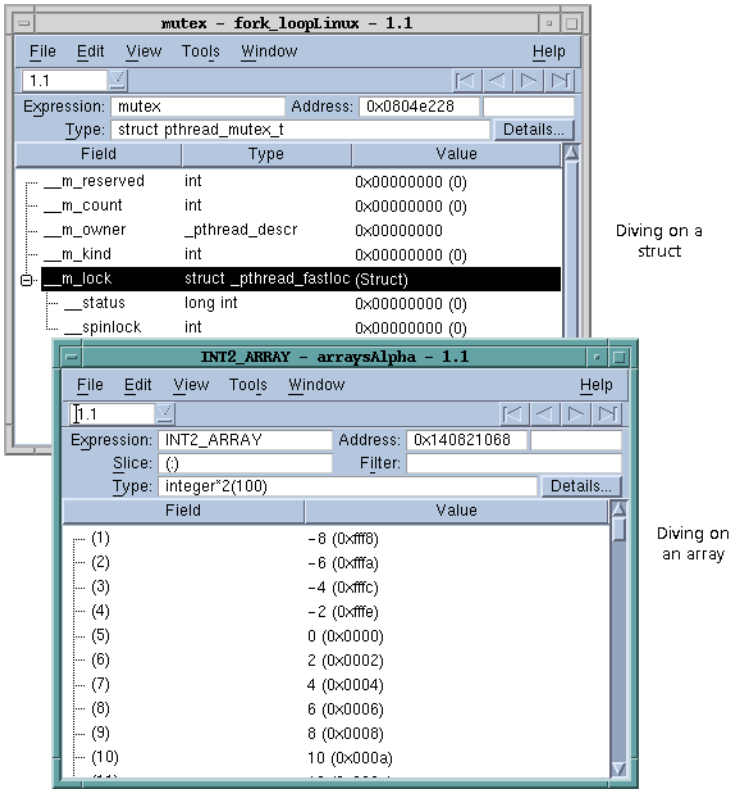
Diving *is something you can do almost everywhere in TotalView. What happens depends on where you are. In general, it either brings you to a different place in your program or shows you more information about what your diving on. To dive on something, position the cursor over the item and click your middle mouse button. If you have a two-button mouse, double-click your left mouse button.*

Diving on a variable tells TotalView to display a window that contains information about the variable. (As you read this manual, you'll come across many other types of diving.)

Some of the values in the Stack Frame Pane are in **bold** type. This lets you know that you can click on the value and then edit it.

The following figure shows two Variable Windows. One window was created by diving on a structure and the second by diving on an array.

Figure 5: Diving on a Structure and an Array



Because the data displayed in a Variable Window might not be simple, you can *also* dive on data in the Variable Window. When you dive in a Variable Window, TotalView replaces the window's contents with the new information. If this isn't what you want, you can use the **View > Dive in New Window** command to display this information in a separate window.

If the data being displayed is a pointer, diving on the variable dereferences the pointer and then displays the data that is being pointed to. In this way, you can follow linked lists. The upper right corner of a Variable Window has arrow buttons ( ). Selecting these buttons lets you undive and redive. For example, if you're following a pointer chain, click the center-left-pointing arrow to go back to where you just were. Click the center-right-pointing arrow to go forward to the place you previously dove on. The outermost two arrows do undive all and redive all operations.

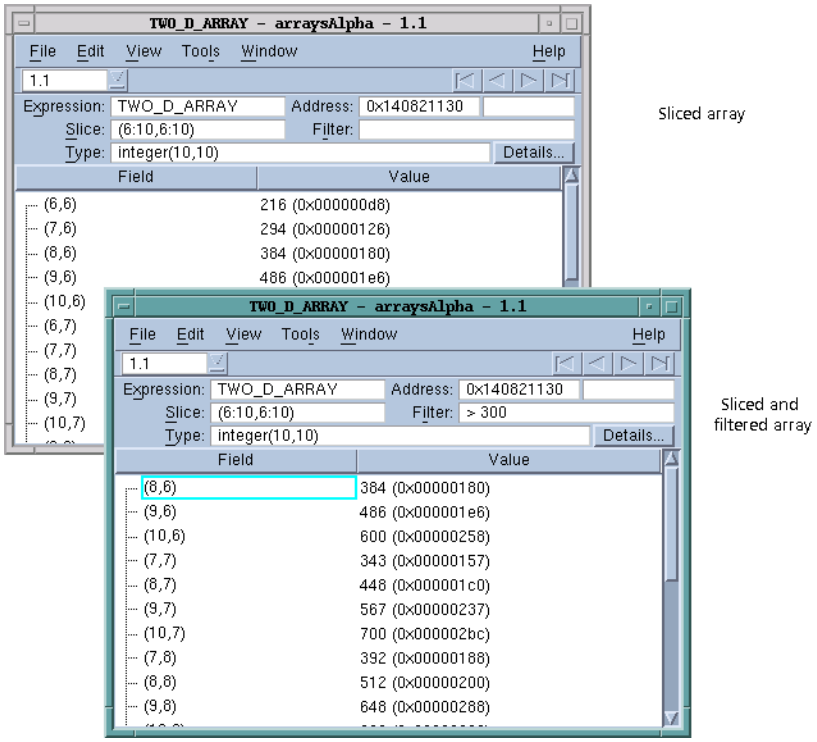
Examining Arrays

Because arrays almost always have copious amounts of data, TotalView has a variety of ways to simplify how it should display this data.

The Variable Window in the upper left corner of the figure on the next page shows a basic *slice* operation. Slicing tells TotalView to display array elements whose positions are named within the slice. In this case, TotalView is displaying elements 6 through 10 in each of the array's two dimensions.

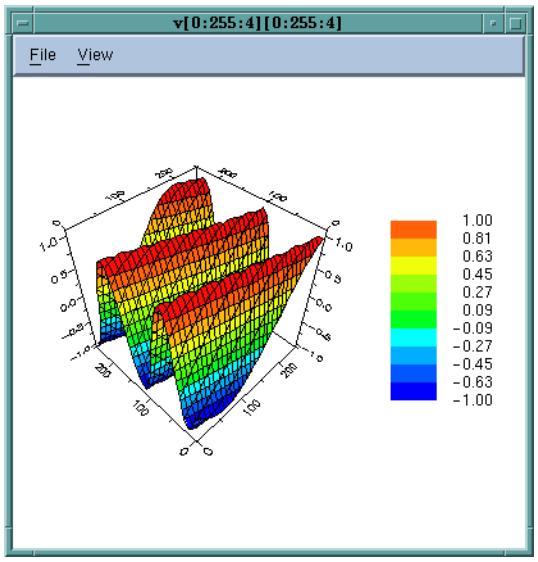
The other Variable Window in this figure combines a *filter* with a slice. A filter tells TotalView to display data if it meets some criteria that you specify. Here, the filter says "of the array elements that could be displayed, only display elements whose value is greater than 300."

Figure 6: Slicing and Filtering Arrays



While slicing and filtering let you reduce the amount of data that TotalView displays, you might want to see the shape of the data. If you select the **Tools > Visualize** command, TotalView shows a graphic representation of the information in the Variable Window; for example:

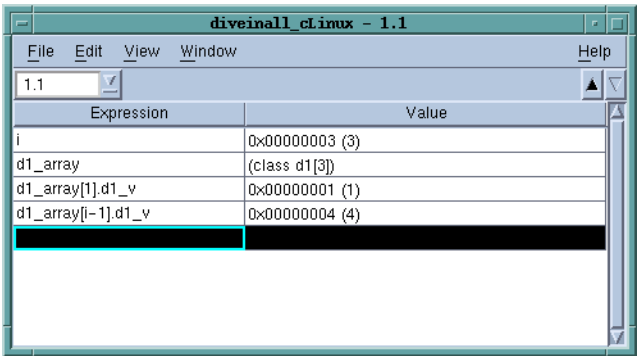
Figure 7: Visualizing an Array



Seeing Groups of Variables

Variable Windows let you critically examine many aspects of your data. In many cases, you're not interested in much of this information. Instead, all you're interested in is the variable's value. This is what the Expression List Window is for. It differs from the Variable Window in that it lets you see the values of many variables at the same time. For example:

Figure 8: Tools > Expression List Window



You can add variables to this window in several ways, such as:

- Type the variable's name in the Expression column.
- Select the variable in the Source or Stack Frame Panes or in a Variable Window, right-click, then select **Add to Expression List** from the context menu.

For more information, see "*Viewing a List of Variables*" on page 254.

Setting Watchpoints

Using watchpoints is yet another way to look at data. A TotalView watchpoint lets you see when a variable's data changes. Watchpoints work in a different way than other action points. A watchpoint stops execution whenever a data value changes, no matter what instruction changed the data. That is, if you change data from 30 different statements, the watchpoint stops execution right after any of these 30 statements make a change. Another example is if something is trashing a memory location. You put a watchpoint on that location and then wait until TotalView stops execution because the watchpoint was executed.

To create a watchpoint for a variable, select **Tools > Watchpoint** from the variable's Variable Window.

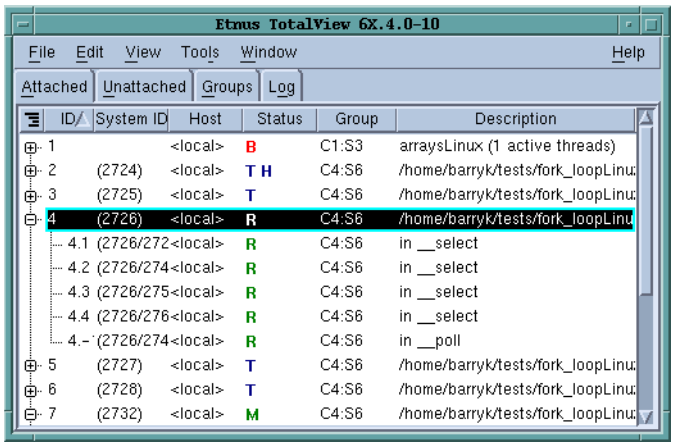
Debugging Multiprocess and Multithreaded Programs

When your program creates processes and threads, TotalView can automatically bring them under its control. If the processes are already running, TotalView can acquire them, too. You don't need to have multiple debuggers running. One TotalView is all you need.

The processes that your program creates can be local or remote. Both are presented to you in the same way. You can display them in the current Process Window or display them in an additional window.

The Root Window, which automatically appears after you start TotalView, contains an overview of all processes and threads being debugged. Diving on a process or a thread listed in the Root Window takes you quickly to the information you want to see.

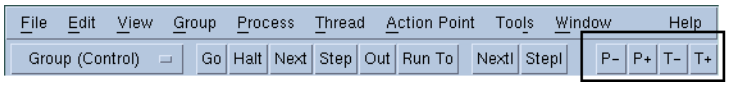
Figure 9: The Root Window



If you need to debug processes that are already running, the Unattached Page lets you dive on other processes you own. After diving on them, you can debug them in the same way as any other process or thread.

In the Process Window, you can switch between processes and threads by clicking the process and thread switching buttons in the toolbar. These are the four buttons on the right side of the toolbar in the following figure.

Figure 10: Process and Thread Switching Buttons



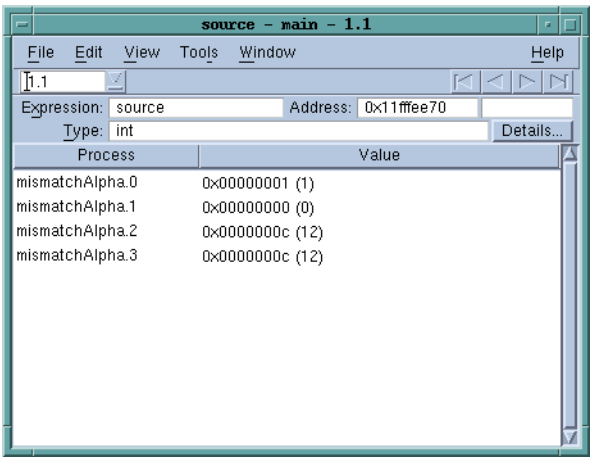
Every time you click one of these buttons, TotalView switches contexts. The switching order is the order in which you see things in the Root Window.

In many cases, you'll be using one of the popular parallel execution models. TotalView supports MPI and MPICH, OpenMP, ORNL PVM (and HP Alpha DPVM), SGI shared memory (shmem), Global Arrays, and UPC. You could be using threading in your programs. Or, you can compile your programs using products provided by your hardware vendor or third-party compilers, such as those from Intel and the Free Software Foundation (the GNU compilers).

Supporting Multiprocess and Multithreaded Programs

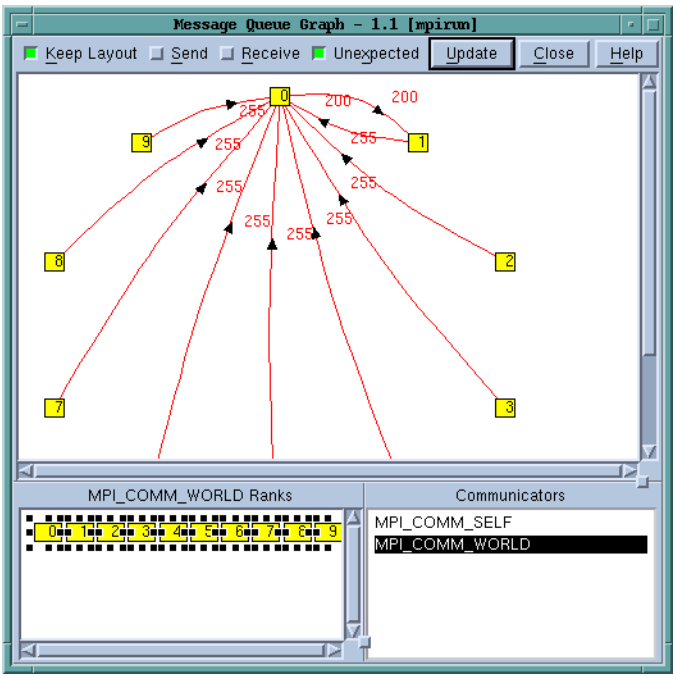
When debugging multiprocess, multithreaded programs, you often want to see the value of a variable in each process or thread simultaneously. The TotalView laminated data view does this for you. The following figure shows an example of what you see after you laminate a multithreaded program.

Figure 11: A Laminated Variable Window



If you're debugging an MPI program, the **Tools > Message Queue Graph** Window graphically displays the program's message queues.

Figure 12: A Message Queue Graph



Clicking on the boxed numbers tells TotalView to place the associated process into a Process Window. Clicking on a number next to the arrow tells TotalView to display more information about that message queue.

This book contains many additional examples.

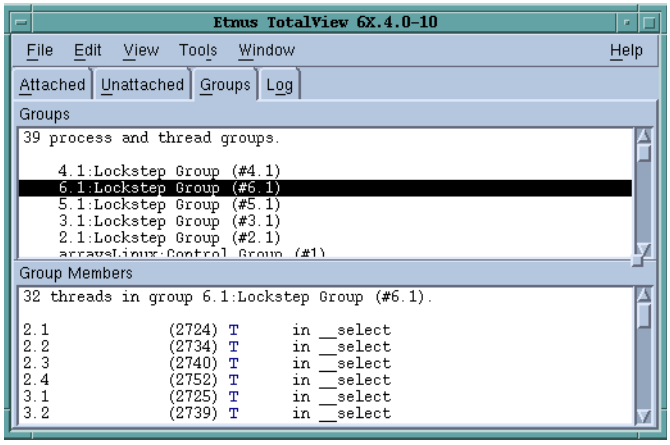
Using Groups and Barriers

When running a multiprocess and multithreaded program, TotalView tries to automatically place your executing processes into different groups. While you can always individually stop, start, step, and examine any thread or process, TotalView lets you perform these actions on groups of threads and processes. In most cases, you do the same kinds of operations on the same kinds of things. The toolbar's pulldown menu lets you select the target of your action. Here's the toolbar

Figure 13: Toolbar With Pulldown

For example, if you are debugging an MPI program, you might set the pulldown to **Process (Workers)**. (Chapter 11 describes the reasons for setting the pulldown this way.) When you want to see which threads are in a group, you can select the Groups tab from the Root Window.

Figure 14: The Root Window Group Page



Introducing the CLI

The TotalView Command Line Interpreter, or CLI, contains an extensive set of commands that you can type into a command window. These commands are embedded in a version of the Tcl command interpreter. When you open a CLI window, you can enter any Tcl statements that you could enter in any version of Tcl. You can also enter commands that Etnus has added to Tcl that allow you to debug your program. Because these debugging com-

mands are native to the TotalView Tcl, you can also use Tcl to manipulate the program being debugged. This means that you can use the CLI to create your own commands or perform any kind of repetitive operation. For example, the following code shows how to set a breakpoint at line 1038 using the CLI:

```
dbreak 1038
```

When you combine Tcl and TotalView, you can simplify what you are doing. For example, the following code shows how to set a group of breakpoints:

```
foreach i {1038 1043 1045} {
   dbreak $i
}
```

Chapter 8 presents more realistic examples.

Information about the CLI is scattered throughout this book. Chapter 2 of the *TotalView Reference Guide* contains descriptions of most CLI commands.

What's Next

This chapter has presented just a few TotalView highlights. The rest of this book tells you more about all TotalView features.

All TotalView documentation is available on our Web site at **http://www.etnus.com/Support/docs** in PDF and HTML formats. You can also find this information in the online Help.

About Threads, Processes, and Groups

2

While the specifics of how multiprocess, multithreaded programs execute differ greatly from one hardware platform to another, from one operating system to another, and from one compiler to another, all share some general characteristics. This chapter defines a general model for conceptualizing the way processes and threads execute.

This chapter presents the concepts of *threads*, *processes*, and *groups*. Chapter 11, "*Using Groups, Processes, and Threads*," on page 205 is a more exacting and comprehensive look at these topics.

This chapter contains the following sections:

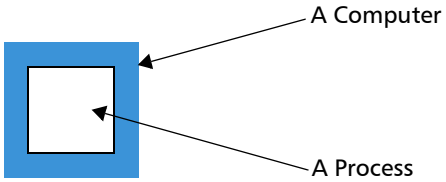
- "*A Couple of Processes*" on page 15
- "*Threads*" on page 17
- "*Complicated Programming Models*" on page 18
- "*Types of Threads*" on page 20
- "*Organizing Chaos*" on page 22
- "*Creating Groups*" on page 26
- "*Simplifying What You're Debugging*" on page 30

A Couple of Processes

When programmers write single-threaded, single-process programs, they can almost always answer the question "Do you know where your program is?" These types of programs are rather simple, looking something like what's shown in the figure on the next page.

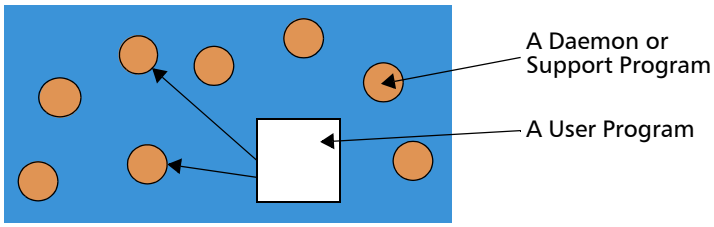
If you use any debugger on these types of programs, you can almost always figure out what's going on. Before the program begins executing, you set a breakpoint, let the program run until it hits the breakpoint, and then inspect variables to see their values. If you suspect that there's a logic problem, you can step the program through its statements, seeing what happens and where things are going wrong.

Figure 15: A Uniprocessor



What is actually occurring, however, is a lot more complicated, since a number of programs are always executing on your computer. For example, your computing environment could have daemons and other support programs executing, and your program can interact with them.

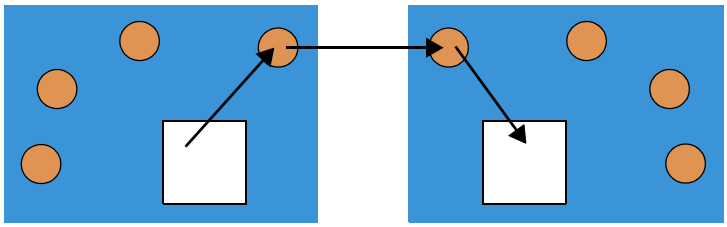
Figure 16: A Program and Daemons



These additional processes can simplify your program because it no longer has to do everything itself. It can hand off some tasks and not have to focus on how that work gets done.

The preceding figure shows an architecture where the application program just sends requests to a daemon. This architecture is very simple. The type of architecture shown in the next figure is more typical. In this example, an email program communicates with a daemon on one computer. After receiving a request, this daemon sends data to an email daemon on another computer, which then delivers the data to another mail program.

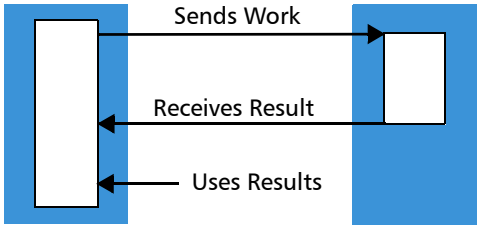
Figure 17: Mail Using Daemons to Communicate



This architecture has one program handing off work to another. After the handoff, the programs do not interact. The program handing off the work just assumes that the work gets done. Some programs can work well like this. Most don't. Most computational jobs do better with a model that allows a program to divide its work into smaller jobs, and parcel this work to other computers. Said in a different way, this model has other machines do some of the first program's work. To gain any advantage, however, the work a program parcels out must be work that it doesn't need right away. In

this model, the two computers act more or less independently. And, because the first computer doesn't have to do all the work, the program can complete its work faster.

Figure 18: Two Computers Working on One Problem



Using more than one computer doesn't mean that less computer time is being used. Overhead due to sending data across the network and overhead for coordinating multiprocessing always means more work is being done. It does mean, however, that your program finishes sooner than if only one computer were working on the problem.

One problem with this model is how a programmer debugs what's happening on the second computer. One solution is to have a debugger running on each computer. The TotalView solution to this debugging problem places a server on each remote processor as it is launched. These servers then communicate with the main TotalView. This debugging architecture gives you one central location from which you can manage and examine all aspects of your program.

You can also have TotalView attach to programs that are already running on other computers. In other words, programs don't have to be started from within TotalView to be debugged by TotalView.

In all cases, it is far easier to write your program so that it only uses one computer at first. After you have it working, you can split up its work so that it uses other computers. It is likely that any problems you find will occur in the code that splits up the program or in the way the programs manipulate shared data, or in some other area related to the use of more than one thread or process. This assumes, of course, that it is practical to write your program as a single-process program. For some algorithms, executing a program on one computer means that it will take weeks to execute.

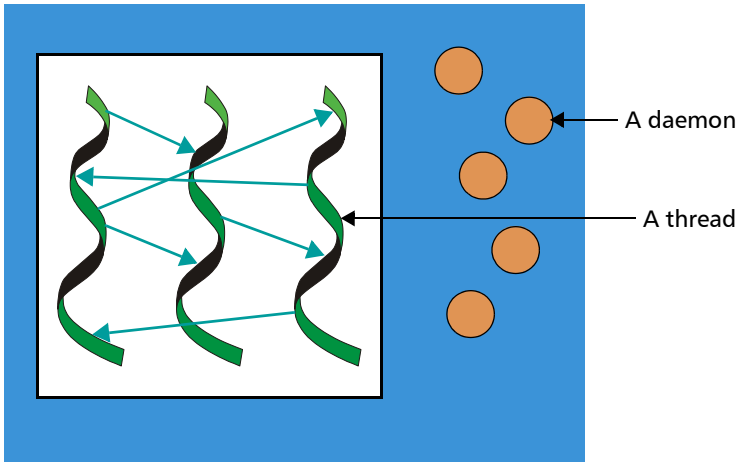
Threads

The operating system owns the daemon programs discussed in the previous section. These daemons perform a variety of activities, from managing computer resources to providing standard services, such as printing.

If operating systems can have many independently executing components, why can't a program? Obviously, a program can and there are various ways

to do this. One programming model splits the work off into somewhat independent tasks within the same process. This is the *threads* model.

Figure 19: Threads



This figure also shows the daemon processes that are executing. (The figures in the rest of this chapter don't show these daemons.)

In this computing model, a program (the main thread) creates threads. If they need to, these newly created threads can also create threads. Each thread executes relatively independently from other threads. You can, of course, program them to share data and to synchronize how they execute.

The debugging issue here is similar to the problem of processes running on different machines. In both, a debugger must intervene with more than one executing entity. It has to understand multiple address spaces and multiple contexts.

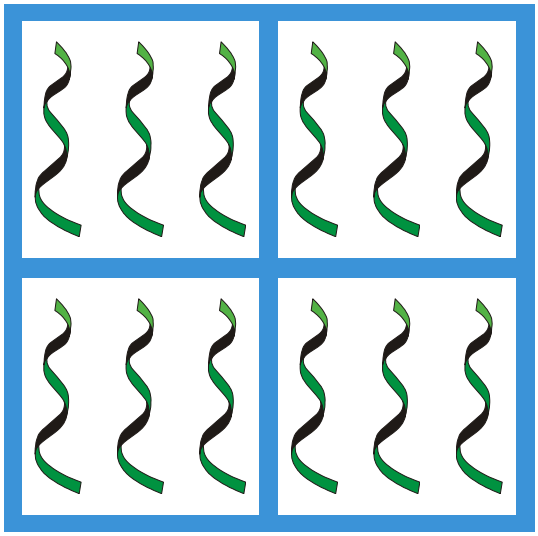
There's not a lot of difference between a multithreaded or a multiprocess program when you are using TotalView. The way in which TotalView displays process information is very similar to how it displays thread information.

Complicated Programming Models

While most computers have one processor, high-performance computing often uses computers that have more than one. And as hardware prices decrease, this model is starting to become more widespread. Having more

than one processor means that the threads model shown in the figure in the previous section changes to look something like this.

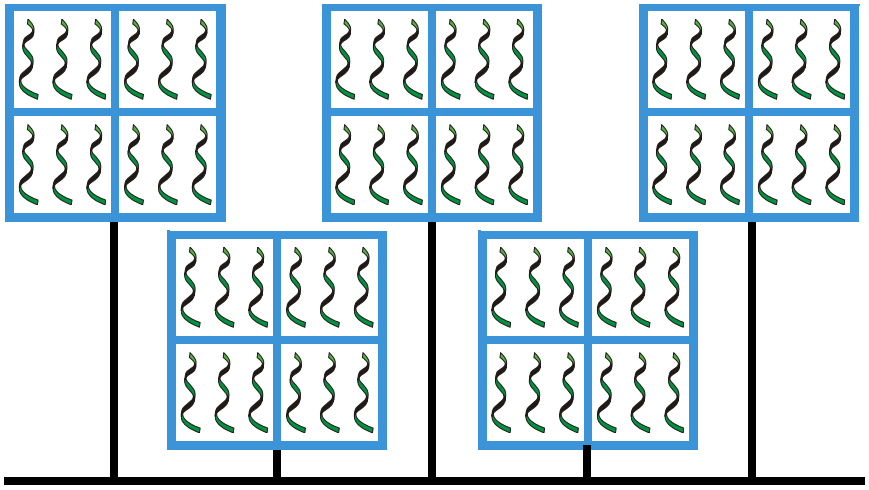
Figure 20: Four-Processor Computer



This figure shows four linked processors in one computer, each of which has three threads. This architecture is an extension to the model that links more than one computer together. Its advantage is that the processor doesn't need to communicate with other processors over a network, as it is completely self-contained.

The next step is to join many multiprocessor computers together. The following figure shows five computers, each with four processors, with each processsor running three threads. If this figure shows the execution of one program, then the program is using 60 threads.

Figure 21: Four Processors on a Network



This figure depicts only processors and threads. It doesn't have any information about the nature of the programs and threads or even whether the programs are copies of one another or represent different executables.

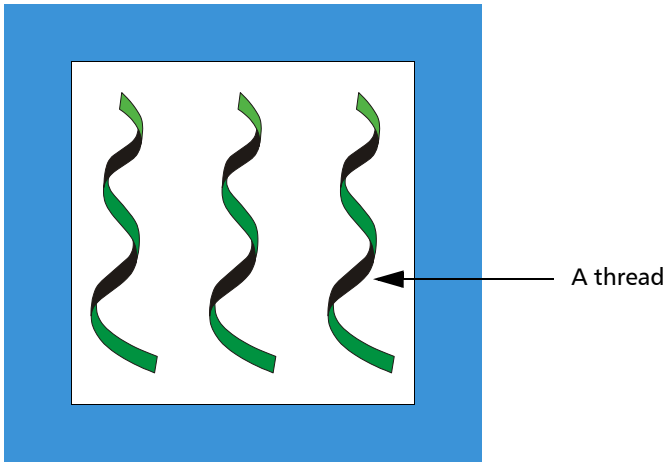
At any time, it is next to impossible to guess which threads are executing and what a thread is actually doing. To make matters worse, many multiprocessor programs begin by invoking a process such as **mpirun** or IBM **poe**, whose function is to distribute and control the work being performed. In this kind of environment, a program (or the program in a library) is using another program to control the workflow across processors.

When there are problems working this way, traditional debuggers and solutions don't work. TotalView, on the other hand, organizes this mass of executing procedures for you and lets you distinguish between threads and processes that the operating system uses from those that your program uses.

Types of Threads

All threads aren't the same. The following figure shows a program with three threads.

Figure 22: Threads (again)



Assume that all of these threads are *user threads*; that is, they are threads that perform some activity that you've programmed.

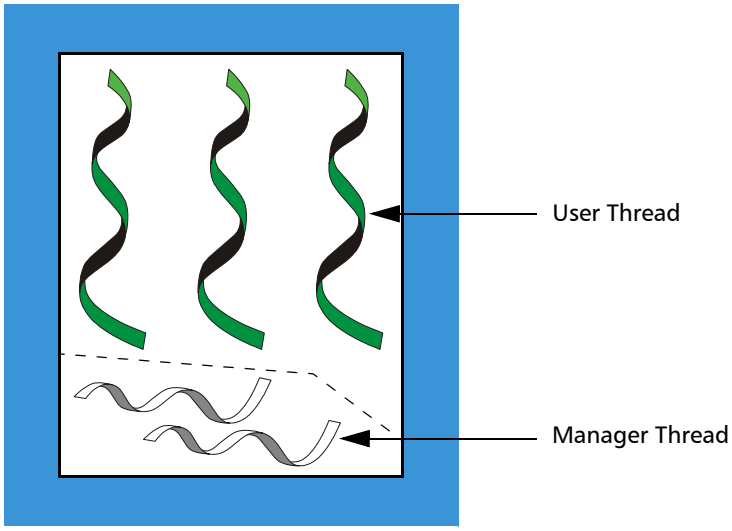
Many computer architectures have something called user mode, user space, or something similar. User threads means something else. The TotalView definition of a user thread is simply a unit of execution created by a program.

Because your program creates user threads to do the work of your program, they are also called *worker threads*.

Other threads can also be executing. For example, the threads that are part of the operating environment are manager threads. These are threads that

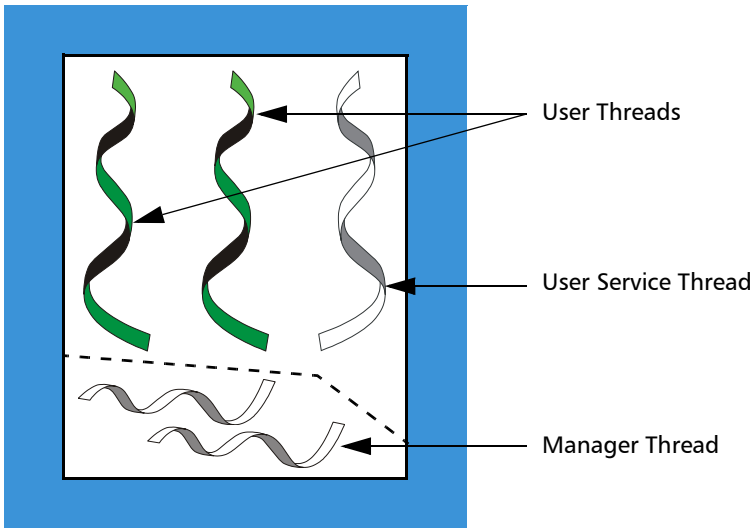
your environment or operating system adds to your program to help it get work done. In the following two figures, the horizontal threads at the bottom are user-created manager threads.

Figure 23: User and Service Threads



All threads are not created equal and all threads do not execute equally. In most cases, a program also creates manager-like threads. Since these user-created manager threads perform services for other threads, they are called *service threads*.

Figure 24: User, Service, and Manager Threads



These service threads are also worker threads. For example, the sole function of a user service thread might be to send data to a printer in response to a request from the other two threads.

One reason you need to know which of your threads are service threads is that a service thread performs different types of activities than your other

threads. Because their activities are different, they are usually developed separately and, in many cases, are not involved with the fundamental problems being solved by the program. Here are two examples:

- The code that sends messages between processes is far different than the code that performs fast Fourier transforms. Its bugs are quite different than the bugs that create the data that is being transformed.
- A service thread that queues and dispatches messages sent from other threads might have bugs, but the bugs are different than the rest of your code and you can handle them separately from the bugs that occur in nonservice user threads.

In contrast, your user threads are the agents that perform the program's work. Being able to distinguish between the two kinds of threads means that you can focus on the threads and processes that actively participate in an activity, rather than on threads performing subordinate tasks.

Although this last figure shows five threads, most of your debugging effort will focus on just two threads.

Organizing Chaos

It is possible to debug programs that are running thousands of processes and threads across hundreds of computers by individually looking at each. However, this is almost always impractical. The only workable approach is to organize your processes and threads into groups and then debug your program by using these groups. In other words, in a multiprocess, multithreaded program, you are most often not programming each process or thread individually. Instead, most high-performance computing programs perform the same or similar activities on different sets of data.

TotalView cannot know your program's architecture; however, it can make some intelligent guesses based on what your program is executing and where the program counter is. Using this information, TotalView automatically organizes your processes and threads into the following predefined groups:

- **Control Group**: All the processes that a program creates. These processes can be local or remote. If your program uses processes that it did not create, TotalView places them in separate control groups. For example, a client/server program that has two distinct executables that run independently of one another has each executable in a separate control group. In contrast, processes created by **fork()** are in the same control group.
- **Share Group**: All the processes within a control group that share the same code. S*ame code* means that the processes have the same executable file name and path. In most cases, your program has more than one share group. Share groups, like control groups, can be local or remote.
- **Workers Group**: All the worker threads within a control group. These threads can reside in more than one share group.

- **Lockstep Group**: All threads that are at the same PC (program counter). This group is a subset of a workers group. A lockstep group only exists for stopped threads. By definition, all members of a lockstep group are within the same workers group. That is, a lockstep group cannot have members in more than one workers group or more than one control group.

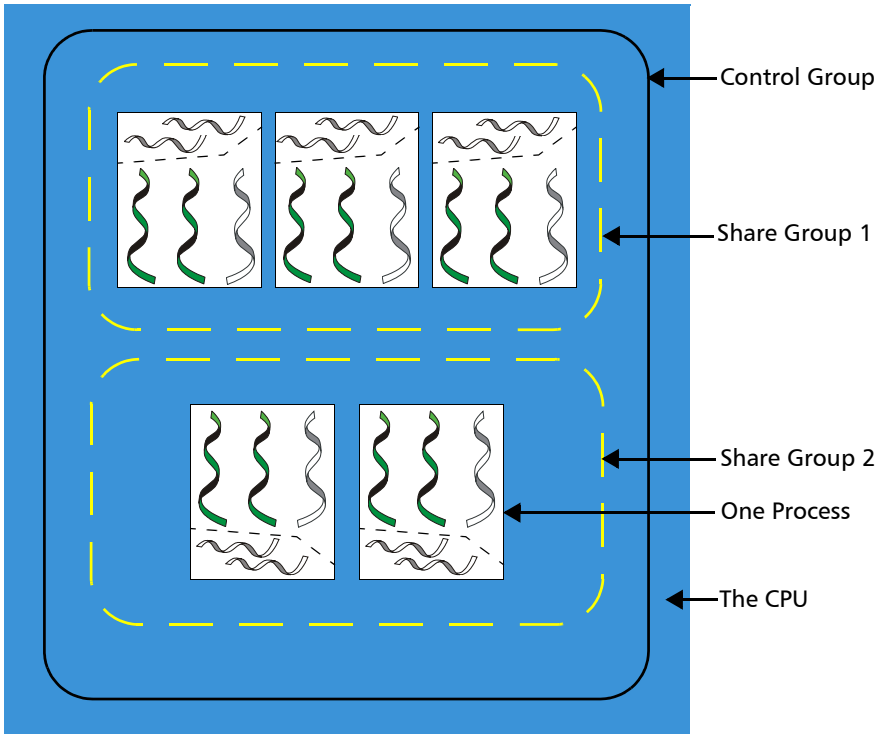
The control and share groups only contain processes; the workers and lockstep groups only contain threads.

TotalView lets you manipulate processes and threads individually and by groups. In addition, you can create your own groups and manipulate a group's contents (to some extent). For more information, see Chapter 11, "*Using Groups, Processes, and Threads*," on page 205.

Not all operating systems let you individually manipulate threads. If TotalView cannot manipulate your program's threads, it dims the commands in its menu and toolbar.

The following figure shows a processor running five processes (ignoring daemons and other programs not related to your program) and the threads within the processes. This figure shows a control group and two share groups within the control group.

Figure 25: Five-Processes: Their Control and Share Groups

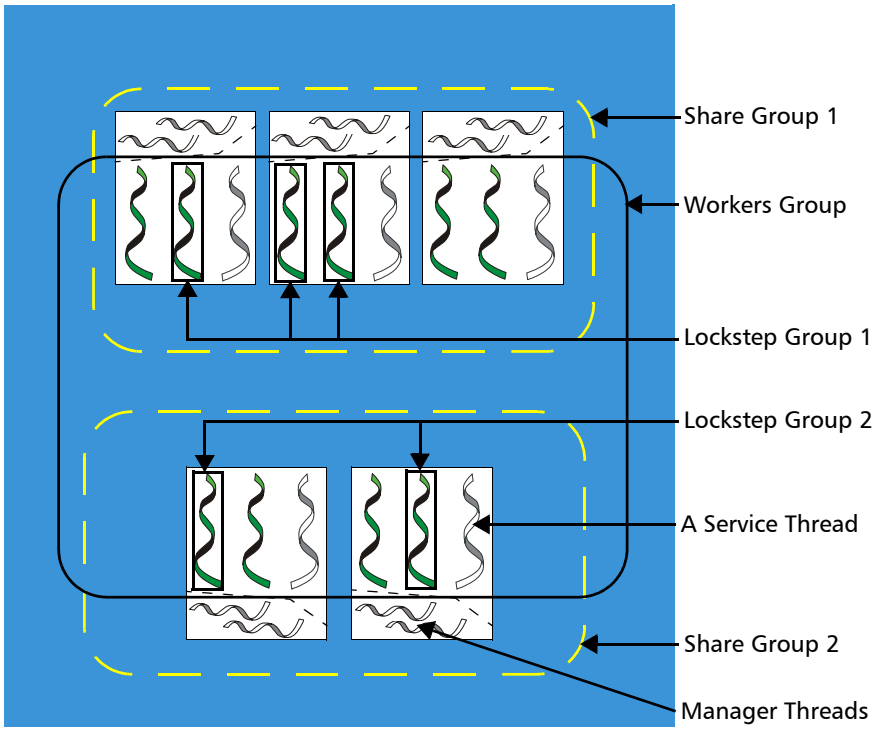


Many of the elements in this figure are used in other figures in this book. These elements are as follows:

CPU	The one outer square represents the CPU. All elements in the drawing operate within one CPU.
Processes	The five white inner squares represent processes being executed by the CPU.
Control Group	The large rounded rectangle that surrounds the five processes shows one control group. This diagram doesn't indicate which process is the main procedure.
Share Groups	The two smaller rounded rectangles having white dashed lines surround processes in a share group. This drawing shows two share groups within one control group. The three processes in the first share group have the same executable. The two processes in the second share group share a second executable.

The control group and the share group only contain processes. The next figure shows how TotalView organizes the threads in the previous figure. This figure adds a workers group and two lockstep groups.

Figure 26: Five Processes: Adding Workers and Lockstep Groups



This figure doesn't show the control group since it encompasses everything in this figure. That is, this example's control group contains all of the program's lockstep, share, and worker group's processes and threads.

The additional elements in this figure are as follows:

Workers Group All nonmanager threads within the control group make up the workers group. This group includes service threads.

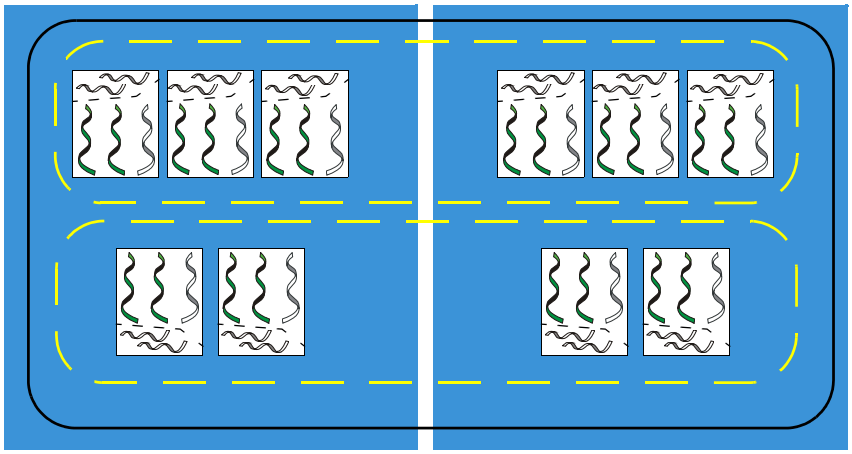
Lockstep Groups Each share group has its own lockstep group. The previous figure shows two lockstep groups, one in each share group.

Service Threads Each process has one service thread. A process can have any number of service threads, but this figure only shows one.

Manager Threads The ten manager threads are the only threads that do not participate in the workers group.

The following figure extends the previous figure to show the same kinds of information executing on two processors.

Figure 27: Five Processes and Their Groups on Two Computers



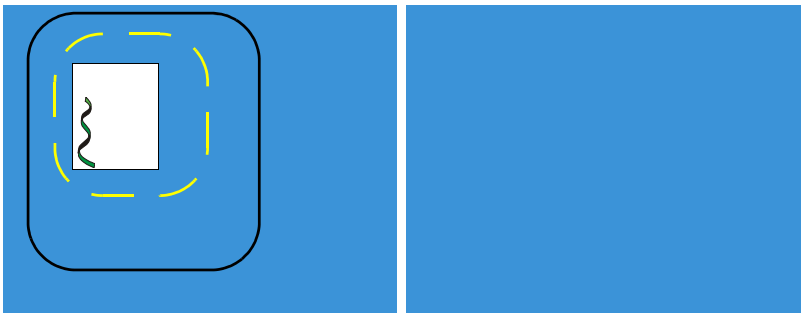
This figure differs from the other ones in this section because it shows ten processes executing within two processors rather than five processes within one processor. Although the number of processors has changed, the number of control and share groups is unchanged. This makes a nice example. However, most programs are not this regular.

Creating Groups

TotalView places processes and threads in groups as your program creates them. The exception is the lockstep groups that are created or changed whenever a process or thread hits an action point or is stopped for any reason. There are many ways to build this type of organization. The following steps indicate the beginning of how you might do this.

Step 1 TotalView and your program are launched and your program begins executing.

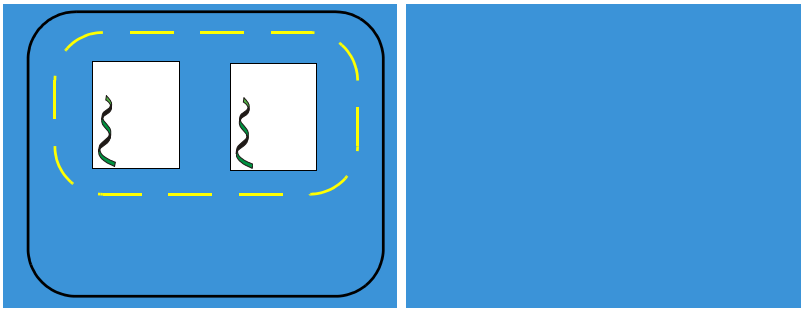
Figure 28: Step 1: A Program Starts



- **Control group**: The program is loaded and creates a group.
- **Share group**: The program begins executing and creates a group.
- **Workers group**: The thread in the main() routine is the workers group.
- **Lockstep group**: There is no lockstep group because the thread is running. (Lockstep groups only contain stopped threads.)

Step 2 The program forks a process.

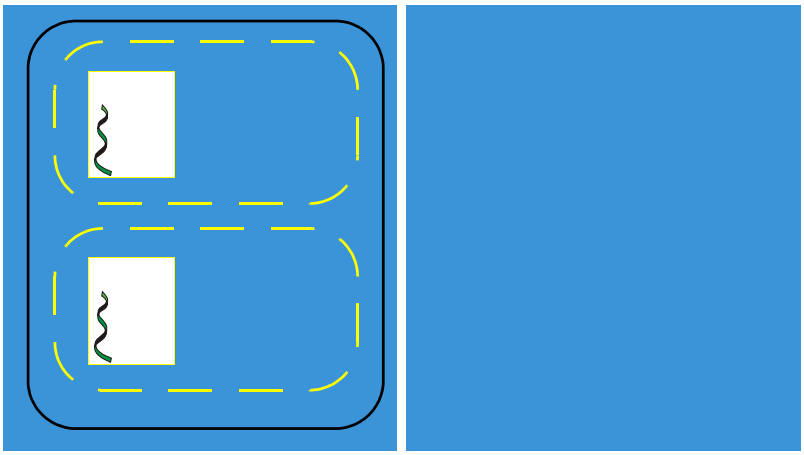
Figure 29: Step 2: Forking a Process



- **Control group**: TotalView adds a second process to the existing group.
- **Share group**: TotalView adds a second process to the existing group.
- **Workers group**: TotalView adds the thread in the second process to the existing group.
- **Lockstep group**: There are no lockstep groups because the threads are running.

Step 3 The first process uses the **exec()** function to create a second process.

Figure 30: Step 3: Creating a Process using exec()



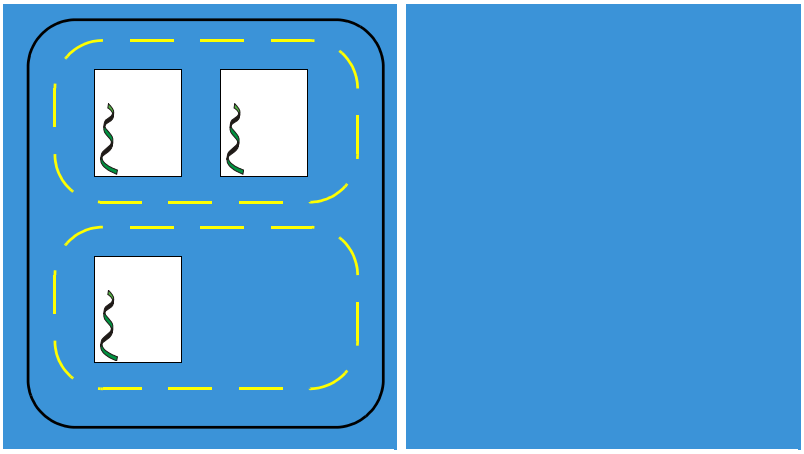
- **Control group**: The group is unchanged.
- **Share group**: TotalView creates a second share group with the process created by the **exec()** function as a member. TotalView removes this process from the first share group.
- **Workers group**: Both threads are in the workers group.
- **Lockstep group**: There are no lockstep groups because the threads are running.

Step 4 The first process hits a break point.

- **Control group**: The group is unchanged.
- **Share group**: The groups are unchanged.
- **Workers group**: The group is unchanged.
- **Lockstep group**: TotalView creates a lockstep group whose member is the thread of the current process. (In this example, each thread is its own lockstep group.)

Step 5 The program is continued and TotalView starts a second version of your program from the shell. You attach to it within TotalView and put it in the same control group as your first process.

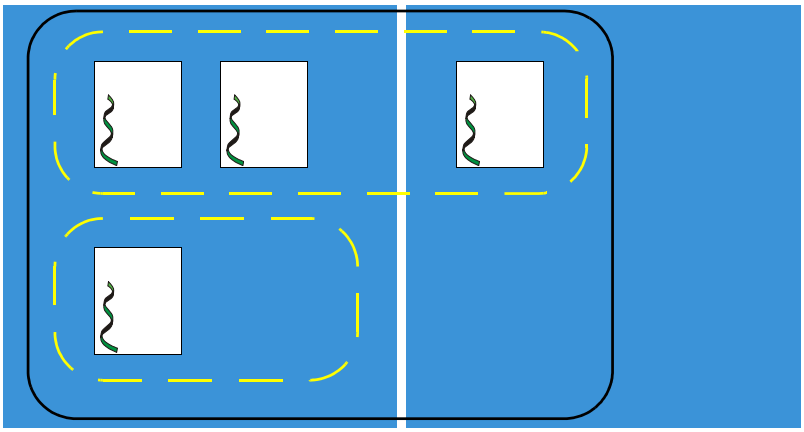
Figure 31: Step 5: Creating a Second Version



- **Control group**: TotalView adds a third process.
- **Share group**: TotalView adds this third process to the first share group.
- **Workers group**: TotalView adds the thread in this third process to the group.
- **Lockstep group**: There are no lockstep groups because the threads are running.

Step 6 Your program creates a process on another computer.

Figure 32: Step 6: Creating a Remote Process

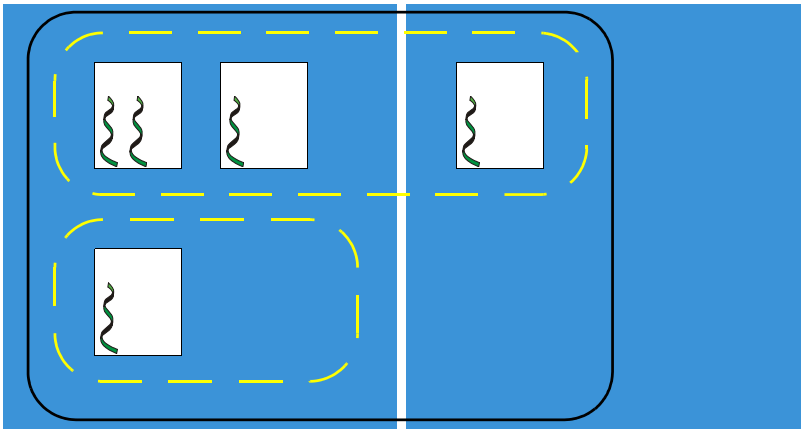


- **Control group**: TotalView extends the control group so that it contains the fourth process, which is running on the second computer.
- **Share group**: The first share group now contains this newly created process, even though it is running on the second computer.

- **Workers group**: TotalView adds the thread within this fourth process to the workers group.
- **Lockstep group**: There are no lockstep groups because the threads are running.

Step 7 A process within the control group creates a thread. This adds a second thread to one of the processes.

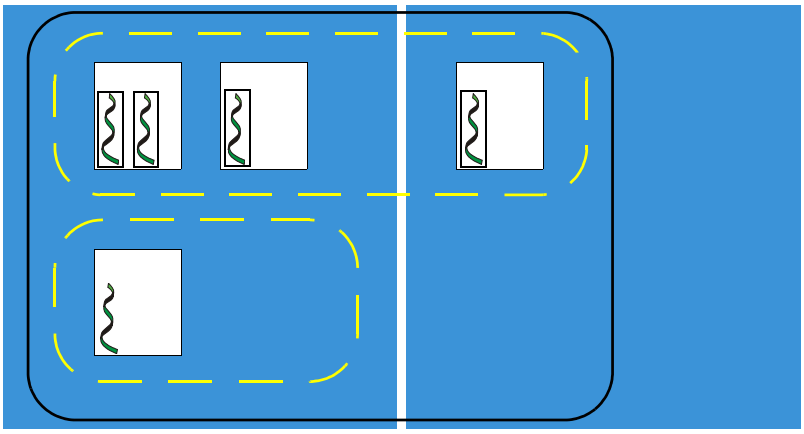
Figure 33: Step 7: Creating a Thread



- **Control group**: The group is unchanged.
- **Share group**: The group is unchanged.
- **Workers group**: TotalView adds a fifth thread to this group.
- **Lockstep group**: There are no lockstep groups because the threads are running.

Step 8 A breakpoint is set on a line in a process executing in the first share group. By default, TotalView shares the breakpoint. The program executes until all three processes are at the breakpoint.

Figure 34: Step 8: Hitting a Breakpoint

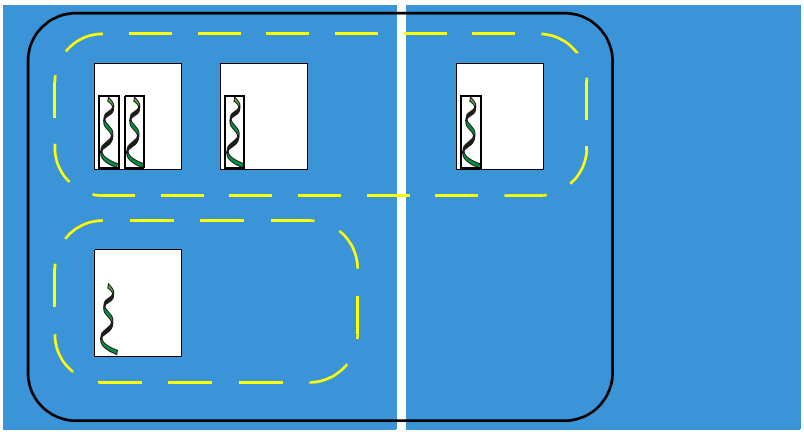


- **Control group**: The group is unchanged.
- **Share group**: The groups are unchanged.

- **Workers group**: The group is unchanged.
- **Lockstep groups**: TotalView creates a lockstep group whose members are the four threads in the first share group.

Step 9 You tell TotalView to step the lockstep group.

Figure 35: Step 9: Stepping the Lockstep Group



- **Control group**: The group is unchanged.
- **Share group**: The groups are unchanged.
- **Workers group**: The group is unchanged.
- **Lockstep group**: The lockstep groups are unchanged. (There are other lockstep groups; this is explained in Chapter 11.)

What Comes Next This example could keep on going to create a more complicated system of processes and threads. However, adding more processes and threads won't change the basics of what has been covered.

Simplifying What You're Debugging

The reason you're using a debugger is because your program isn't operating correctly and the way you think you're going to solve the problem (unless it is a &%$# operating system problem, which, of course, it usually is) is by stopping your program's threads, examining the values assigned to variables, and stepping your program so you can see what's happening as it executes.

Unfortunately, your multiprocess, multithreaded program and the computers upon which it is executing have lots of things executing that you want TotalView to ignore. For example, you don't want to be examining manager and service threads that the operating system, your programming environment, and your program create.

Also, most of us are incapable of understanding exactly how a program is acting when perhaps thousands of processes are executing asynchronously. Fortunately, there are only a few problems that require full asynchronous behavior at all times.

One of the first simplifications you can make is to change the number of processes. For example, suppose you have a buggy MPI program running on 128 processors. Your first step might be to have it execute in an 8-processor environment.

After you get the program running under the debugger's control, run the process being debugged to an action point so that you can inspect the program's state at that point. In many cases, because your program has places where processes are forced to wait for an interaction with other processes, you can ignore what they are doing.

TotalView lets you control as many groups, processes, and threads as you need to control. Although you can control each one individually, you might have problems remembering what you're doing if you're controlling large numbers of these things independently. TotalView creates and manages groups so that you can focus on portions of your program.

In most cases, you don't need to interact with everything that is executing. Instead, you want to focus on one process and the data that this process manipulates. Things get complicated when the process being investigated is using data created by other processes, and these processes might be dependent on other processes.

The following is a typical way to use TotalView to locate problems:

1. At some point, make sure that the groups you are manipulating do not contain service or manager threads. (You can remove processes and threads from a group by using the **Group > Edit Group** command.)

   CLI: **dgroups –remove**

2. Place a breakpoint in a process or thread and begin investigating the problem. In many cases, you are setting a breakpoint at a place where you hope the program is still executing correctly. Because you are debugging a multiprocess, multithreaded program, set a *barrier point* so that all threads and process will stop at the same place.

   Don't step your program except where you need to individually look at what occurs in a thread. Using barrier points is much more efficient. Barrier points are discussed in Chapter 14 and online within the Action Point area within the Tip of the Week archive at http://www.etnus.com/Support/Tips/.

3. After execution stops at a barrier point, look at the contents of your variables. Verify that your program state is actually correct.
4. Begin stepping your program through its code. In most cases, step your program synchronously or set barriers so that everything isn't running freely.

   Things begin to get complicated at this point. You've been focusing on one process or thread. If another process or thread modifies the data and you become convinced that this is the problem, you need to go off to it and see what's going on.

You need to keep your focus narrow so that you're only investigating a limited number of behaviors. This is where debugging becomes an art. A multiprocess, multithreaded program can be doing a great number of things. Understanding where to look when problems occur is the art.

For example, you most often execute commands at the default focus. Only when you think that the problem is occurring in another process do you change to that process. You still execute in the default focus, but this time the default focus changes to another process.

Although it seems like you're often shifting from one focus to another, you probably will do the following:

- Modify the focus so that it affects just the next command. If you are using the GUI, you might select this process and thread from the list displayed in the Root Window. If you are using the CLI, you use the **dfocus** command to limit the scope of a future command. For example, the following is the CLI command that steps thread 7 in process 3:

  ```
  dfocus t3.7 dstep
  ```

- Use the **dfocus** command to change focus temporarily, execute a few commands, and then return to the original focus.

This chapter is just an overview of the threads, processes, and groups. Chapter 11, "*Using Groups, Processes, and Threads*," on page 205 contains the details.

Part II: Setting Up

This section of the TotalView Users Guide contains information about running TotalView in the different types of environments in which you execute your program.

Chapter 3: Setting Up a Debugging Session

You configure your TotalView environment the same way on all operating systems and in all environments. This chapter tells you what you need to know to start TotalView and tailor how it works.

At a minimum, glance at this chapter to see what it contains so that you can come back at a later time, if you need to.

Chapter 4: Setting Up Remote Debugging Sessions

When you are debugging a program that has processes executing on a remote computer, TotalView launches server processes for these remote processes. Usually, you don't need to know much about this. The primary focus of this chapter is what to do when you have problems.

If you aren't having problems, you probably won't ever look at the information in this chapter.

Chapter 5: Setting Up Parallel Debugging Sessions

TotalView lets you debug programs created using many different parallel environments, such as OpenMP, MPI, MPICH, UPC, and the like. This chapter discusses how to set up these environments.

Because each environment's discussion is self-contained, use the table of contents to locate the information you need.

Setting Up a Debugging Session

3

This chapter explains how to set up a TotalView session. It also describes some of the most-used commands and procedures. For information on setting up remote debugging, see Chapter 4, "*Setting Up Remote Debugging Sessions*," on page 63. For information on setting up parallel debugging sessions, see Chapter 5, "*Setting Up Parallel Debugging Sessions*," on page 77.

This chapter contains the following sections:

- "*Compiling Programs*" on page 35
- "*Starting TotalView*" on page 36
- "*Exiting from TotalView*" on page 40
- "*Loading Executables*" on page 40
- "*Attaching to Processes*" on page 42
- "*Detaching from Processes*" on page 45
- "*Examining Core Files*" on page 45
- "*Viewing Process and Thread States*" on page 46
- "*Handling Signals*" on page 48
- "*Setting Search Paths*" on page 50
- "*Setting Command Arguments*" on page 53
- "*Setting Input and Output Files*" on page 53
- "*Setting Preferences*" on page 54
- "*Setting Environment Variables*" on page 60
- "*Monitoring TotalView Sessions*" on page 61

Compiling Programs

The first step in getting a program ready for debugging is to add your compiler's –g debugging command-line option to all the other options you use when compiling your program. This option tells your compiler to generate symbol table debugging information; for example:

```
cc –g –o executable source_program
```

You can also debug programs that you did not compile using the **–g** option, or programs for which you do not have source code. For more information, see "*Viewing the Assembler Version of Your Code*" on page 131.

The following table presents some general considerations "*Compilers and Platforms* in the *TotalView Reference Guide* contains additional considerations.

Compiler Option or Library	What It Does	When to Use It
Debugging symbols option (usually **–g**)	Generates debugging information in the symbol table.	Before debugging *any* program with TotalView.
Optimization option (usually **–O**)	Rearranges code to optimize your program's execution. Some compilers won't let you use the **–O** option and the **–g** option at the same time. Even if your compiler lets you use the **–O** option, don't use it when debugging your program, since strange results often occur.	After you finish debugging your program with TotalView.
Multiprocess programming library (usually **dbfork**)	Uses special versions of the **fork()** and **execve()** system calls. In some cases, you need to use the **–lpthread** option. For more information about dbfork, see "*Linking with the dbfork Library*" contained in the "*Compilers and Platforms*" Chapter of the *TotalView Reference Guide*.	Before debugging a multiprocess program that explicitly calls **fork()** or **execve()**. See "*Debugging Processes That Call the fork() Function*" on page 302 and "*Debugging Processes that Call the execve() Function*" on page 302.

Using File Extensions

When TotalView opens a file, it uses the file's extension to determine which programming language you used. If you are using an unusual extension, you can manually associate your extension with a programming language by setting the **TV::suffixes** variable in a startup file. For more information, see the "*TotalView Variables*" chapter in the *TotalView Reference Guide*.

Starting TotalView

TotalView can debug programs that run in many different computing environments and that use a variety of parallel processing modes. This section looks at few of the ways you can start TotalView. See the "*TotalView Command Syntax*" chapter in the *TotalView Reference Guide* for more detailed information.

In most cases, the command for starting TotalView looks like the following:

totalview [*executable* [*corefile*]] [*options*]

where *executable* is the name of the executable file to debug and *corefile* is the name of the core file to examine.

CLI: **totalviewcli** [*executable* [*corefile*]] [*options*]

Your environment may require you to start TotalView in another way. For example, if you are debugging an MPI program, you must invoke TotalView on **mpirun**. For details, see Chapter 5, "*Setting Up Parallel Debugging Sessions*," on page 77.

You can use the GUI and the CLI at the same time. Use the **Tools** > **Command Line** command to display the CLI's window.

The following examples show different ways of that you might begin debugging a program:

Starting TotalView

totalview

Starts TotalView without loading a program or core file. After TotalView starts, you can load a program by using the **File** > **New Program** command.

CLI: **totalviewcli** then **dload** *executable*

Debugging a program

totalview *executable*

Starts TotalView and loads the *executable* program.

CLI: **totalviewcli** *executable*

Debugging a core file

totalview *executable corefile*

Starts TotalView and loads the *executable* program and the *corefile* core file.

CLI: **dattach –c** *corefile* **–e** *executable*

Passing arguments to the program being debugged

totalview *executable* **–a** *args*

Starts TotalView and passes all the arguments following the **–a** option to the *executable* program. When you use the **–a** option, you must enter it as the last TotalView option on the command line.

CLI: **totalviewcli** *executable* **–a** *args*

If you don't use the **–a** option and you want to add arguments after TotalView loads your program, use the **Process** > **Startup Parameters** command.

CLI: **dset ARGS_DEFAULT** {*value*}

Debugging a program that runs on another computer

totalview *executable* –remote *hostname_or_address*[:*port*]

Starts TotalView on your local host and the TotalView Debugger Server (**tvdsvr**) on a remote host. After TotalView begins executing, it loads the program specified by *executable* for remote debugging. You can specify a host name or a TCP/IP address. If you need to, you can also enter the TCP/IP port number.

CLI: totalviewcli *executable* –r *hostname_or_address*[:*port*]

Debugging a MPICH Program

mpirun -np *count* -tv *executable*

The MPICH mpirun command obtains information from the **TOTALVIEW** environment variable and then uses this information when it starts the first process in the parallel job.

Here's where you can find more information:

- Debugging parallel programs such as MPI, PVM, or UPC, see Chapter 5, "*Setting Up Parallel Debugging Sessions*," on page 77.
- The **totalview** command, see "*TotalView Command Syntax*" in the *TotalView Reference Guide*.
- Remote debugging, see "*Setting Up and Starting the TotalView Debugger Server*" on page 63 and "*TotalView Debugger Server (tvdsvr) Command Syntax*" in the *TotalView Reference Guide*.

Initializing TotalView

When TotalView begins executing, it can read initialization and startup information from a number of files. The two most commonly used are initialization files that you create and preference files that TotalView creates.

An *initialization file* is a place where you can store CLI functions, set variables, and execute actions. TotalView interprets the information in this file whenever it begins executing. This file, which you must name **tvdrc**, resides in a **.totalview** subdirectory contained in your home directory. TotalView creates this directory for you the first time it executes.

TotalView can actually read more than one initialization file. You can place these files in your installation directory, the **.totalview** subdirectory, or the directory in which you invoke TotalView. If an initialization file is present in one or all of these places, TotalView reads and executes each. Only the initialization file in your **.totalview** directory has the name **tvdrc**. The other initialization files have the name **.tvdrc**. A dot precedes the file name.

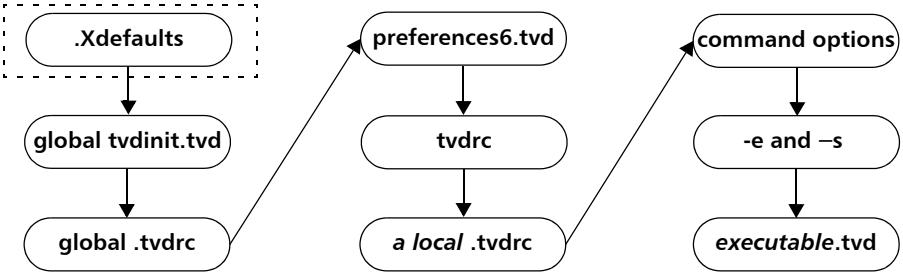
Before Version 6.0, you placed your personal **.tvdrc** *file in your home directory. If you do not move this file to the* **.totalview** *directory, TotalView will still find it. However, if you also have a* **tvdrc** *file in the* **.totalview** *directory, TotalView ignores the* **.tvdrc** *file in your home directory.*

TotalView automatically writes your *preferences file* to your **.totalview** subdirectory. Its name is **preferences6.tvd**. Do not modify this file as TotalView overwrites it when it saves your preferences.

If you add the **–s** *filename* option to either the **totalview** or **totalviewcli** shell command, TotalView executes the CLI commands contained in *filename*. This startup file executes after a **tvdrc** file executes. The **–s** option lets you, for example, initialize the debugging state of your program, run the program you're debugging until it reaches some point where you're ready to begin debugging, and even create a shell command that starts the CLI.

The following figure shows the order in which TotalView executes initialization and startup files.

Figure 36: Startup and Initialization Sequence



The **.Xdefaults** file, which is actually read by the server when you start X Windows, is only used by the GUI. The CLI ignores it. Prior to TotalView release 6.0, the **.Xdefaults** file was used extensively. TotalView 6.0 only obtains Visualizer and window hints from this file.

If you have an X resources file, TotalView reads it the first time Release 6.0 starts executing. It then writes any TotalView resources it finds to your **preferences6.tvd** *file. If you change a value within your resources file after TotalView writes your preferences file, TotalView ignores your change. You can force TotalView to reread your X resources file by deleting your preferences file.*

The **tvdinit.tvd** file is in the TotalView **lib** directory. It contains startup macros that TotalView requires. Do not edit this file. Instead, if you want to set a TotalView variable or define or run a CLI macro, create a file named **.tvdrc** and place it in the TotalView **lib** directory.

As part of the initialization process, TotalView exports three environment variables into your environment: **LM_LICENSE_FILE**, **TVROOT**, and either **SHLIB_PATH** or **LD_LIBRARY_PATH**.

If you have saved an action point file to the same subdirectory as your program, TotalView automatically reads the information in this file when it loads your program.

The format of a release 6.0 action point file differs from that used in earlier releases. While TotalView 6.0 can read an action point file created by an earlier version, earlier versions cannot read a release 6.0 or later action point file.

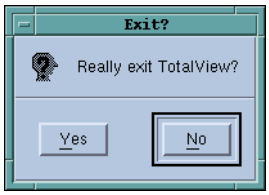
You can also invoke scripts by naming them in the **TV::process_load_callbacks** list. For information on using this variable, see the "*Variables*" chapter of the *TotalView Reference Guide*.

If you are debugging multiprocess programs that run on more than one computer, TotalView caches library information in the **.totalview** subdirectory. If you want to move this cache to another location, set **TV::library_cache_directory** to this location. TotalView can share the files in this cache directory among users.

Exiting from TotalView

To exit from TotalView, select **File > Exit**. You can select this command in the Root, Process, and Variable Windows. After selecting this command, TotalView displays the following dialog box:

Figure 37: File > Exit Dialog Box



Select **Yes** to exit. As TotalView exits, it kills all programs and processes that it started. However, programs and processes that TotalView did not start continue to execute.

If you have a CLI *window open, TotalView also closes this window. Similarly, if you type* **exit** *in the* CLI*, the* CLI *closes* GUI *windows. If you type* **exit** *in the* CLI *and you have a* GUI *window open, TotalView still displays this dialog box.*

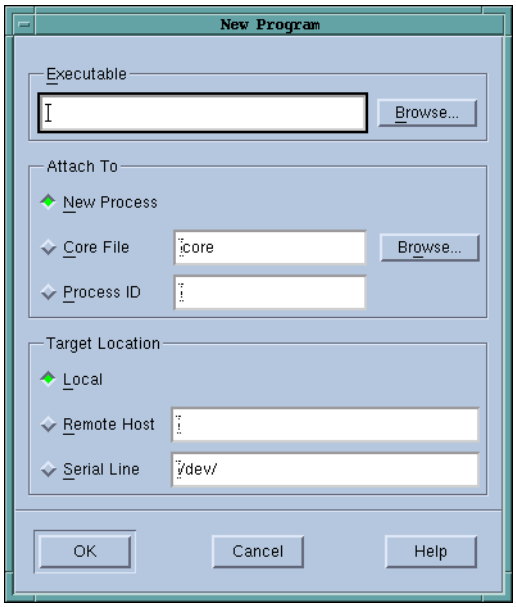
CLI: **exit**

If you have both the CLI and the GUI open, type Ctrl+D to close the CLI window and not exit from TotalView.

Loading Executables

TotalView can debug programs on local and remote hosts, and programs that you access over networks and serial lines. The **File > New Program** command, which is located in the Root and Process Windows, loads local and remote programs, core files, and processes that are already running. After you select this command, TotalView displays the following dialog box.

Figure 38: File > New Program Dialog Box



You can do the following with the controls in this dialog box:

- **Load a new executable**

  Type the path name in the **Executable** field.

  CLI: dload –e *executable*

- **Load a core file**

  Type the name in the **Core File** field. You must also type the path name of the executable associated with this core file in the **Executable** field.

  CLI: dattach –c *corefile* –e *executable*

- **Load a program using process ID**

  Type a process ID in the **Process ID** field *and* type the associated executable's path name in the **Executable** field.

  CLI: dattach *executable pid*

If you need to debug a program on a remote machine, type the machine's host name or IP address in the **Remote Host** field. If the program is local, make sure that you have selected the **Local** button.

CLI: dload *executable* –r *hostname*

You can use a full or relative path name in the **Executable** and **Core File** fields. If you enter a file name, TotalView searches for it in the list of direc-

tories named using the **File > Search Path** command or listed in your **PATH** environment variable.

CLI: dset EXECUTABLE_PATH

If you select **New Process**, TotalView always loads a new copy of the program named in the **Executable** field. Even if the program is already loaded, TotalView still loads another copy.

For information about debugging programs over a serial line, see "*Debugging Over a Serial Line*" on page 74.

Loading Remote Executables

If TotalView fails to automatically load a remote executable, you may need to disable *autolaunching* for this connection and manually start the TotalView Debugger Server (**tvdsvr**). (*Autolaunching* is the process of automatically launching **tvdsvr** processes.) You can disable autolaunching by adding the *hostname*:*portnumber* suffix to the name you type in the **Remote Host** field. As always, the *portnumber* is the TCP/IP port number on which the debugger server is communicating with TotalView. See "*Setting Up and Starting the TotalView Debugger Server*" on page 63 for more information.

You can connect to a remote machine in the following ways:

- Use the **–remote** command-line option when you start TotalView. For details on the syntax for this option, see "*Starting TotalView*" on page 36.
- Use the **File > New Program** command after you start TotalView.

  CLI: dload *executable* –r *hostname*

- Connect to a remote host using the **File > New Program** command and then display the Unattached Page of the Root Window. You can now attach to these programs by diving into them.

  CLI: If you're using the CLI, you need to know the file's name so that you can attach to the program using the dattach command.

If TotalView supports your program's parallel process runtime library (for example, MPI, PVM, *or* UPC), *it automatically connects to remote hosts. For more information, see Chapter* 5, "*Setting Up Parallel Debugging Sessions,*" *on page* 77.

Attaching to Processes

If a program you're testing is hung or looping (or misbehaving in some other way), you can attach to it while it is running. You can attach to single and multiprocess programs, and these programs can be running remotely.

To attach to a process, either use the Unattached Page in the Root Window or use the **File > New Program** command, which you can select from the Root and Process Windows. (Using the Unattached Page is easier if the pro-

cess is listed. If it's not listed, you must use the **File** > **New Program** command.)

CLI: **dattach** *executable pid*

If the process or any of its children calls the **execve()** routine, you might need to attach to it by creating a new Process Window. This is because TotalView relies on the **ps** command to obtain the process name, and the name the operating system knows for the process might be different from what TotalView thinks it is.

When you exit from TotalView, TotalView kills all programs and processes that it started. However, programs and processes that were executing before you brought them under the debugger's control continue to execute.

Attaching Using the Unattached Page

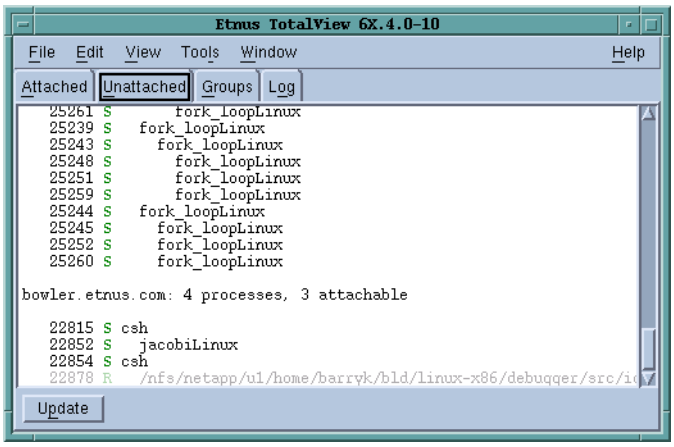
GUI

To use the Unattached Page to attach a process:

1. Go to the Root Window and select the Unattached tab.

   This page lists the process ID, status, and name of each process associated with your username. The processes that appear dimmed are those that are being debugged or those that TotalView won't allow you to debug. For example, you can't debug the TotalView process itself. The processes at the top of this figure are local; the remaining processes are remote.

Figure 39: Unattached Page



If you're debugging a remote process, the Unattached Page also shows processes running under your username on each remote host name. You can attach to any of these remote processes. For more information on remote debugging, see "*Setting Up and Starting the TotalView Debugger Server*" on page 63 and "*TotalView Debugger Server (tvdsvr) Command Syntax*" in the *TotalView Reference Guide*.

2. Dive into the process you wish to debug by double-clicking on it.

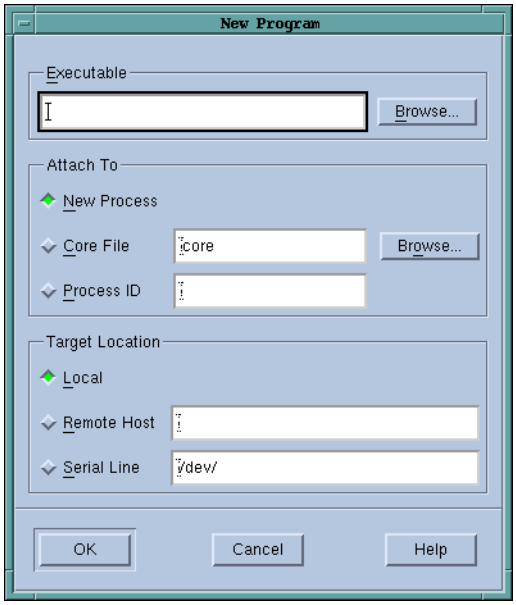
   A Process Window appears. The right arrow points to the current program counter (PC), indicating where the program was executing when TotalView attached to it.

Attaching Using File > New Program or dattach

Here's the procedure for using the **File > New Program** command to attach to a process:

1. Use the **ps** shell command to obtain the process ID (PID) of the process.
2. Select the **File > New Program** command. TotalView displays the dialog box shown in the following dialog box.

Figure 40: File > New Program Dialog Box



Enter a file name in the **Executable** field. This name can be a full or relative path name. If you supply a simple file name, TotalView searches for it in the directories named using the **File > Search Path** command or listed in your **PATH** environment variable.

3. Enter the process ID (PID) of the *unattached* process in the **Process ID** field.

CLI:	dattach *pid*
	dset EXECUTABLE_PATH

4. Select **OK**.

If the executable is a multiprocess program, TotalView asks if you want to attach to all relatives of the process. If this is what you want, choose **Yes**.

If the process has children that call **execve()**, TotalView tries to determine each child's executable. If TotalView has trouble attaching, you must delete (*kill*) the parent process and start it again using TotalView.

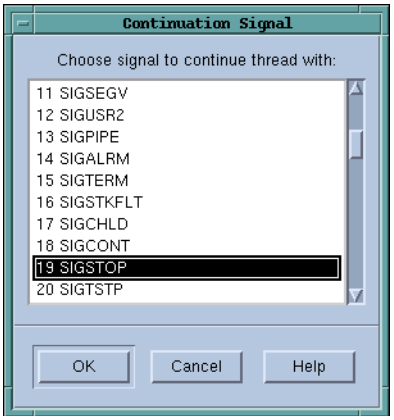
A Process Window appears. In this window, the right arrow points to the current program counter (PC), which is where the program was executing when TotalView attached to it.

Detaching from Processes

To detach from processes that TotalView did not create:

1 (Optional) After opening a Process Window on the process, select the **Thread** > **Continuation Signal** command to display the following dialog box.

Figure 41: Thread > Continuation Signal Dialog Box



CLI: No equivalent to Thread > Continuation Signal exists.

Choose the signal that TotalView sends to the process when it detaches from the process. For example, to detach from a process and leave it stopped, set the continuation signal to **SIGSTOP**.

2 Select **OK**.

3 Select the **Process** > **Detach** command.

CLI: ddetach

When you detach from a process, TotalView removes all breakpoints that you have set in it.

Examining Core Files

If a process encounters a serious error and dumps a core file, you can look at this file using one of the following methods:

- Start TotalView as follows:
  totalview *filename corefile* [*options*]

CLI: **totalviewcli** *filename corefile* [*options*]

- Select the **File** > **New Program** command from the Root Window. In the middle section of the dialog box, type the name of the core file in the **Core File** field, and then select **OK**. Next, enter the *executable*'s name.

> CLI: dattach –c *corefile* –e *executable*

If your operating system can create multithreaded core files, TotalView can examine the thread in which the problem occurred. It can also show you information about other threads in your program.

The Process Window displays the core file, with the Stack Trace, Stack Frame, and Source Panes showing the state of the process when it dumped core. The title bar of the Process Window names the signal that caused the core dump. The right arrow in the line number area of the Source Pane indicates the value of the program counter (PC) when the process encountered the error.

You can examine the state of all variables at the time the error occurred. Chapter 12, "*Examining and Changing Data*," on page 235 contains more information.

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

Viewing Process and Thread States

Process and thread states are displayed in the following:

- The Attached Page of the Root Window, for processes and threads.
- The Unattached Page of the Root Window, for processes.
- The process and thread status bars of the Process Window.
- The Threads Pane of the Process Window.
- The P/T Set Browser.

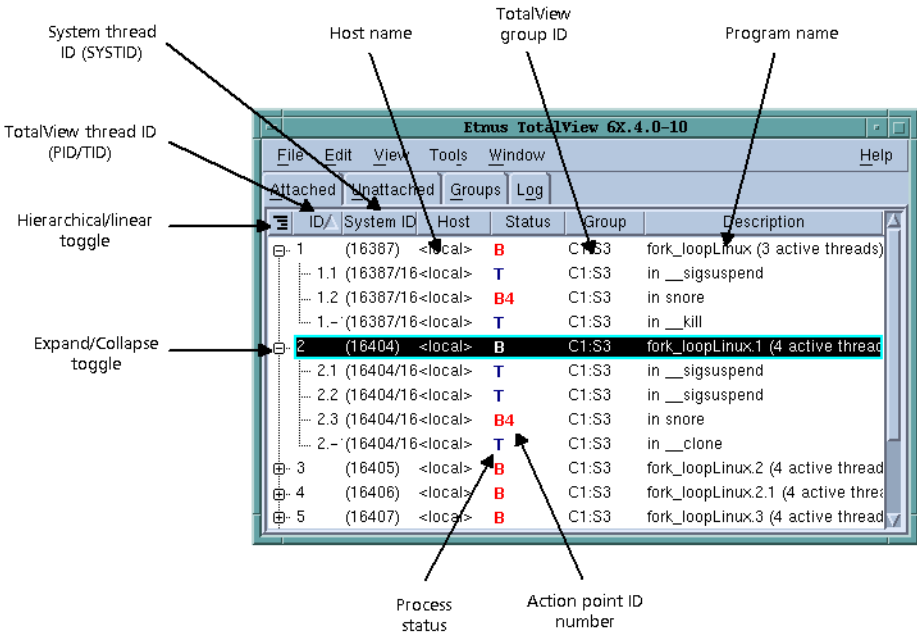
The figure on the next page shows TotalView displaying process state information in the Attached Page.

> CLI: dstatus and dptsets
>
> When you use either of these commands, TotalView also displays state information.

The **Status** of a process includes the process location, the process ID, and the state of the process. (These characters are explained in "*Seeing Attached Process States*" on page 47.)

The Unatttached Page lists all processes associated with your username. The information in this page is similar to the information in the Attached

Figure 42: Attached Page Showing Process and Thread Status



Page, differing only in that TotalView dims the processes being debugged. The status bars in the Process Window display similar information.

Figure 43: Process and Thread Labels in the Process Window

If the thread ID *that TotalView assigns is the same as the operating system thread* ID *your, TotalView only displays* ID.

Seeing Attached Process States

TotalView uses the letters shown in the following table to indicate process and thread state. (These letters are in the **Status** column in the Root Window's Attached Page, as the figure in the previous section shows.)

State Code	State Description
blank	Exited or never created
B	At breakpoint
E	Error reason
H	Held
K	In kernel
M	Mixed
R	Running
T	Stopped reason
W	At watchpoint

The error state usually indicates that your program received a fatal signal, such as **SIGSEGV**, **SIGBUS**, or **SIGFPE**, from the operating system. See "*Han-*

dling Signals" on page 48 for information on controlling how TotalView handles signals that your program receives.

CLI: The CLI prints out a word indicating the state; for example, "breakpoint."

Seeing Unattached Process States

TotalView derives the state information for a process displayed in the Unattached Page from the operating system. The state characters TotalView uses to summarize the state of an unattached process do not necessarily match those used by the operating system. The following table describes the state indicators that TotalView displays:

State Code	State Description
I	Idle
R	Running
S	Sleeping
T	Stopped
Z	Zombie (no apparent owner)

Handling Signals

If your program contains a signal handler routine, you may need to adjust the way TotalView handles signals. The following table shows how TotalView handles UNIX signals if you do not tell it how to handle them:

Signals that TotalView Passes Back to Your Program		Signals that TotalView Treats as Errors	
SIGHUP	SIGIO	SIGILL	SIGPIPE
SIGINT	SIGIO	SIGTRAP	SIGTERM
SIGQUIT	SIGPROF	SIGIOT	SIGTSTP
SIGKILL	SIGWINCH	SIGEMT	SIGTTIN
SIGALRM	SIGLOST	SIGFPE	SIGTTOU
SIGURG	SIGUSR1	SIGBUS	SIGXCPU
SIGCONT	SIGUSR2	SIGSEGV	SIGXFSZ
SIGCHLD		SIGSYS	

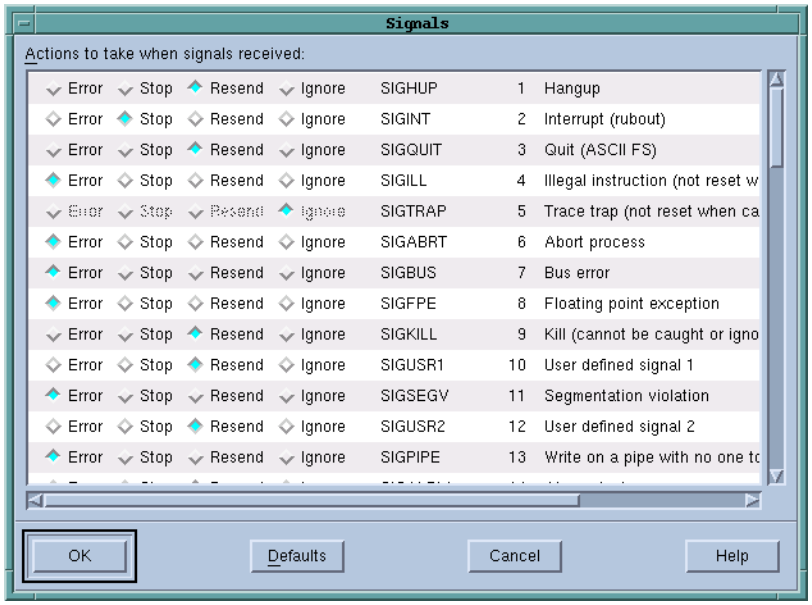
TotalView uses the SIGTRAP *and* SIGSTOP *signals internally. If a process receives either of these signals, TotalView neither stops the process with an error nor passes the signal back to your program. You cannot alter the way TotalView uses these signals.*

On some systems, hardware registers affect how TotalView and your program handle signals such as SIGFPE. For more information, see "*Interpreting the Status and Control Registers*" on page 203 and "*Architectures*" in the *TotalView Reference Guide*.

On an SGI computer, setting the **TRAP_FPE** *environment variable to any value indicates that your program traps underflow errors. If you set this variable, however, you also need to use the controls in the* **File > Signals** *Dialog Box to indicate what TotalView should do with* **SIGFPE** *errors. (In most cases, you set* **SIGFPE** *to* **Resend**.)

You can change the signal handling mode using the **File > Signals** command.

Figure 44: File > Signals Dialog Box



CLI: **dset TV::signal_handling_mode**

The signal names and numbers that TotalView displays are platform-specific. That is, what you see in this box depends on the computer and operating system in which your program is executing.

You can change the default way in which TotalView handles a signal by setting the **TV::signal_handling_mode** variable in a **.tvdrc** startup file. For more information, see Chapter 4 of the "*TotalView Reference Guide*."

When your program receives a signal, TotalView stops all related processes. If you don't want this behavior, clear the **Stop control group on error signal** check box on the Options Page of the **File > Preferences** Dialog Box.

CLI: **dset TV::warn_step_throw**

When your program encounters an error signal, TotalView opens or raises the Process Window. Clearing the **Open process window on error signal** check box, also found on the Options Page in the **File > Preferences** Dialog Box, tells TotalView not to open or raise windows.

CLI: **dset TV::GUI::pop_on_error**

If processes in a multiprocess program encounter an error, TotalView only opens a Process Window for the first process that encounters an error. (If it did it for all of them, TotalView would quickly fill up your screen with Process Windows.)

If you select the **Open process window at breakpoint** check box on the **File > Preferences** Action Points Page, TotalView opens or raises the Process Window when your program reaches a breakpoint.

CLI: dset TV::GUI::pop_at_breakpoint

Make your changes by selecting one of the radio buttons described in the following table.

Button	Description
Error	Stops the process, places it in the *error* state, and displays an error in the title bar of the Process Window. If you have also selected the **Stop control group on error signal** check box, TotalView also stops all related processes. Select this button for severe error conditions, such as SIGSEGV and SIGBUS.
Stop	Stops the process and places it in the *stopped* state. Select this button if you want TotalView to handle this signal as it would a SIGSTOP signal.
Resend	Sends the signal back to the process. This setting lets you test your program's signal handling routines. TotalView sets the SIGKILL and SIGHUP signals to **Resend** since most programs have handlers to handle program termination.
Ignore	Discards the signal and continues the process. The process does not know that something raised a signal.

Do not use Ignore for fatal signals such as SIGSEGV *and* SIGBUS. *If you do, TotalView can get caught in a signal/resignal loop with your program; the signal immediately reoccurs because the failing instruction repeatedly re-executes.*

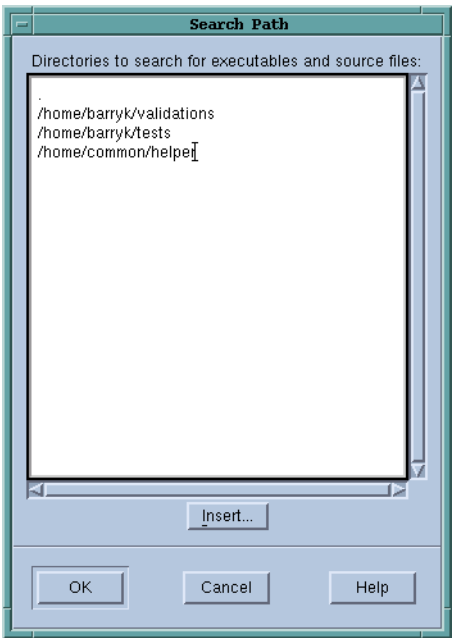
Setting Search Paths

If your source code, executable, and object files reside in different directories, set search paths for these directories with the **File > Search Path** command. You do not need to use this command if these directories are already named in your environment's PATH variable.

CLI: dset EXECUTABLE_PATH

These search paths apply to *all* processes that you're debugging.

Figure 45: File > Search Path Dialog Box



TotalView searches the following directories in order:

1. The current working directory (.) and the directories you specify with the **File > Search Path** command, in the exact order you enter them.
2. The directory name hint. This is the directory that is within the debugging information generated by your compiler.
3. If you entered a full path name for the executable when you started TotalView, TotalView searches this directory.
4. If your executable is a symbolic link, TotalView looks in the directory in which your executable actually resides for the new file.

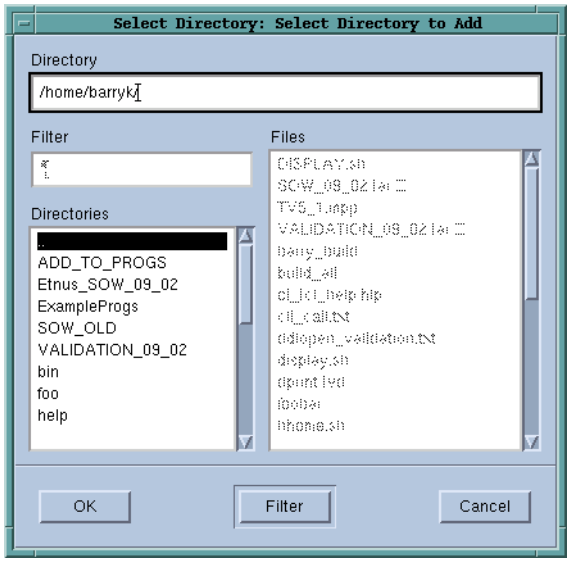
   Since you can have multiple levels of symbolic links, TotalView continues following links until it finds the actual file. After it finds the current executable, it looks in its directory for your file. If the file isn't there, TotalView backs up the chain of links until either it finds the file or determines that the file can't be found.
5. The directories specified in your **PATH** environment variable.
6. The **src** directory within your TotalView installation directory.

When you enter directories into this dialog box, you must enter them in the order you want them searched, and you must enter each on its own line. You can enter directories in the following ways:

- You can type path names directly.
- You can cut and paste directory information.

- You can click the **Insert** button to display the **Select Directory** dialog box that lets you browse through the file system, interactively selecting directories. Here is the dialog box:

Figure 46: Select Directory Dialog Box



The current working directory (.) in the **File > Search Path** Dialog Box is the first directory listed in the window. TotalView interprets relative path names as being *relative* to the current working directory.

If you remove the current working directory from this list, TotalView reinserts it at the top of the list.

After you change this list of directories, TotalView again searches for the source file of the routine being displayed in the Process Window.

You can also specify search directories using the **EXECUTABLE_PATH** environment variable.

The TotalView search path is not usually passed to other processes. For example, it does not affect the **PATH** of a starter process such as **poe**. Suppose that "." is in your TotalView path, but it is not in your **PATH** environment variable. In addition, the executable named **prog_name** is listed in your **PWD** environment variable. In this case, the following command works:

```
totalview prog_name
```

However, the following command does not:

```
totalview poe -a prog_name
```

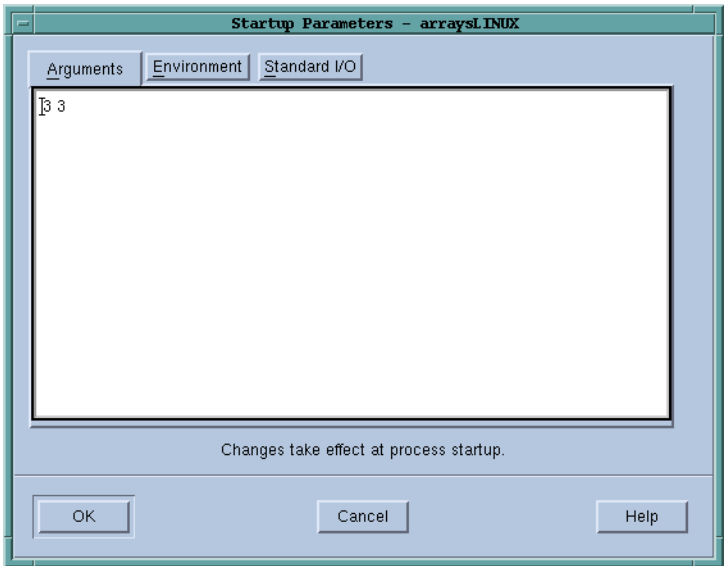
Setting Command Arguments

When TotalView creates a process, it uses the name of the file that contain the executable code as the process's program name. If your program requires command-line arguments and you didn't enter them using the debugger's **–a** command-line option, you can set these arguments *before* you start the process.

To set these arguments:

1 Select the Arguments Tab in the **Process** > **Startup Parameters** Dialog Box.

Figure 47: Process > Startup Parameters Dialog Box: Arguments Page



CLI: dset ARGS_DEFAULT {*value*}

2 Type the arguments you want TotalView to pass to your program. Either separate each argument with a space or place each one on a separate line. If an argument contains spaces, enclose the entire argument in double-quotation marks.

3 When you're done, select **OK**.

Setting Input and Output Files

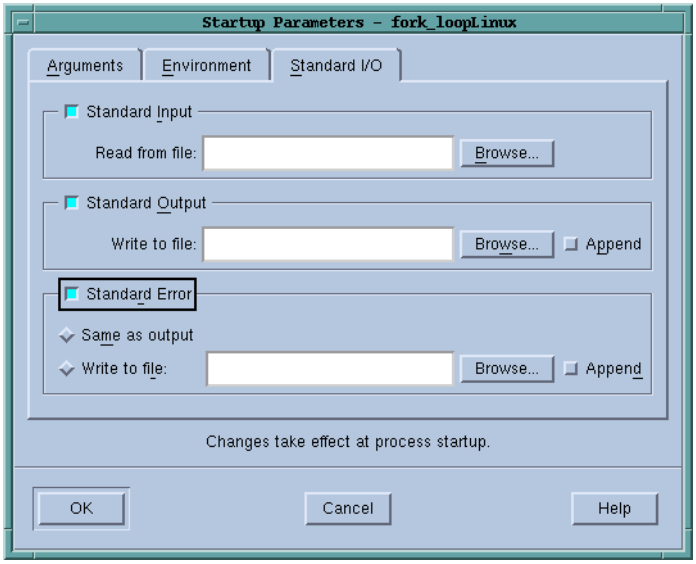
Before your program begins executing, TotalView defines how it manages standard input (**stdin**) and standard output (**stdout**). Unless you tell TotalView otherwise, **stdin** and **stdout** use the shell window from which you invoked TotalView.

The **Process** > **Startup Parameters** command lets you redirect **stdin** or **stdout**. You can only do this before your program begins executing.

To redirect **stdin** or **stdout**:

1 Select the Standard I/O Tab from the dialog box displayed when you invoke the **Process** > **Startup Parameters** command. Here is this page:

Figure 48: Process > Startup Parameters Dialog Box: Standard I/O Page



2 Type the name of the file, relative to your current working directory. Entering names in these text boxes is equivalent to using <, >, or >& symbols in most shells. and then select **OK**.

If you select the **Append** check box, TotalView appends new information to the end of the file if the file already exists. If the **Append** check box isn't checked, TotalView overwrites the file's contents.

If you select the **Same as output** radio button, TotalView writes **Standard Error** information to the same location as **Standard Output**.

CLI: **drun and drerun have arguments that let you reset stdin, stdout, and stderr.**

Setting Preferences

The **File** > **Preferences** command lets you tailor many of the debugger's behaviors. This section contains an overview of these preferences. See the online Help for detailed explanations.

Some settings, such as the prefixes and suffixes examined before loading dynamic libraries, can differ between operating systems. If they can differ, TotalView can store a unique value for each. TotalView does this transparently, which means that you only see an operating system's values when you are running TotalView on that operating system. For example, if you set a server launch string on an SGI computer, it does not affect the value stored for other systems. Generally, this occurs for server launch strings and dynamic library paths.

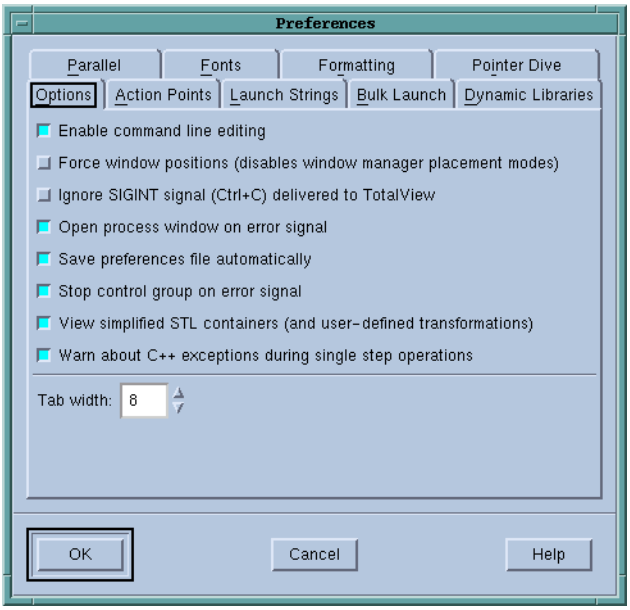
Every preference has a variable that you can set using the CLI. These variables are described in the" *Variables*" chapter of the *TotalView Reference Guide*.

The rest of this section is an overview of these TotalView preferences.

Options

This page contains check boxes that are either general in nature or that influence different parts of the system. See the online Help for information on using these check boxes.

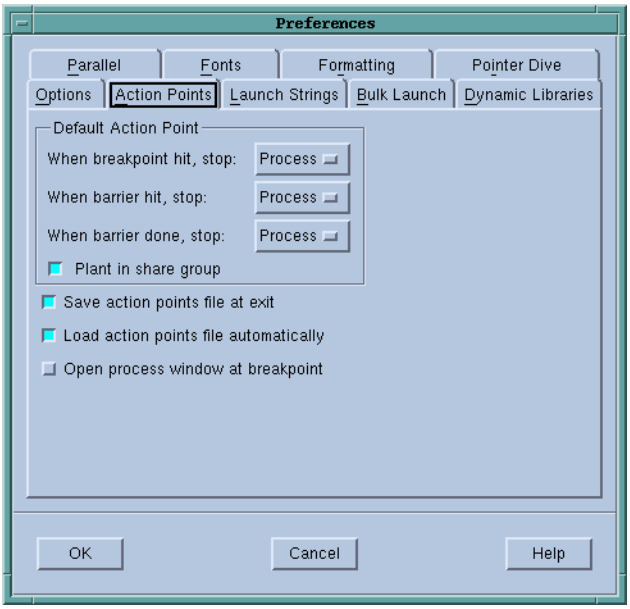
Figure 49: File > Preferences Dialog Box: Options Page



Action Points

The commands on this page indicate whether TotalView should stop anything else when it encounters an action point, the scope of the action point, automatic saving and loading of action points, and if TotalView should open a Process Window for the process encountering a breakpoint.

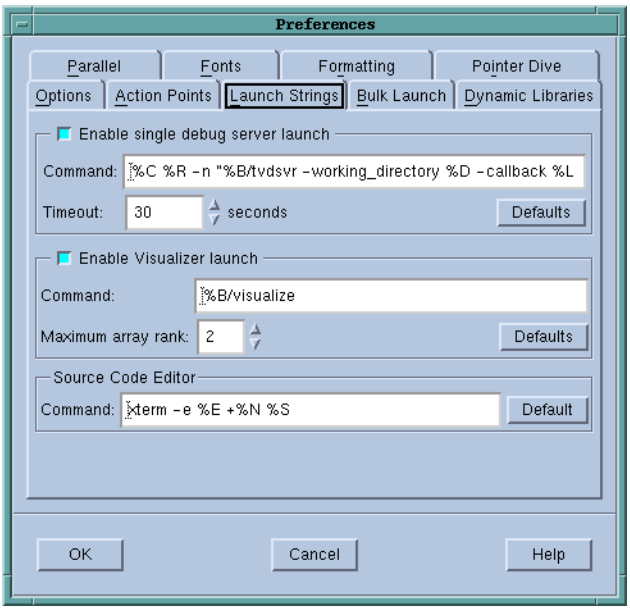
Figure 50: File > Preferences Dialog Box: Action Points Page



Launch Strings

The three areas of this page let you set the launch string that TotalView uses when it launches the **tvdsvr** remote debugging server, the Visualizer, and a source code editor. The values you initially see in the page are default values that TotalView supplies.

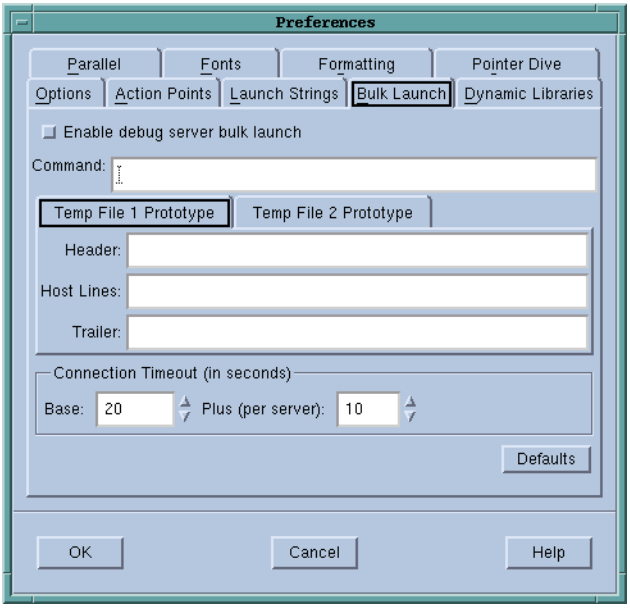
Figure 51: File > Preferences Dialog Box: Launch Strings Page



Bulk Launch

The fields and commands on this page configure the TotalView bulk launch system. (The bulk launch system launches groups of processes simultaneously.) See Chapter 4 for more information.

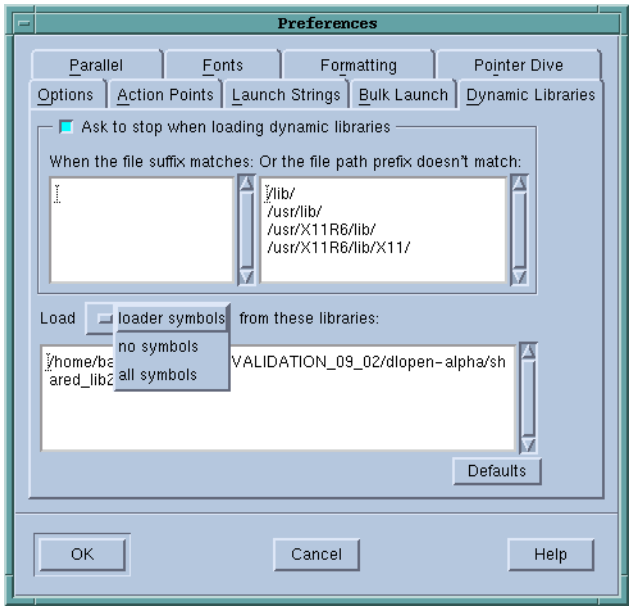
Figure 52: File > Preferences Dialog Box: Bulk Launch Page



Dynamic Libraries

When debugging large programs, you can sometimes increase performance by telling TotalView that it should load an process debugging symbols. This page lets you control which symbols are added to TotalView when it loads a dynamic library, and how many of a library's symbols are read in.

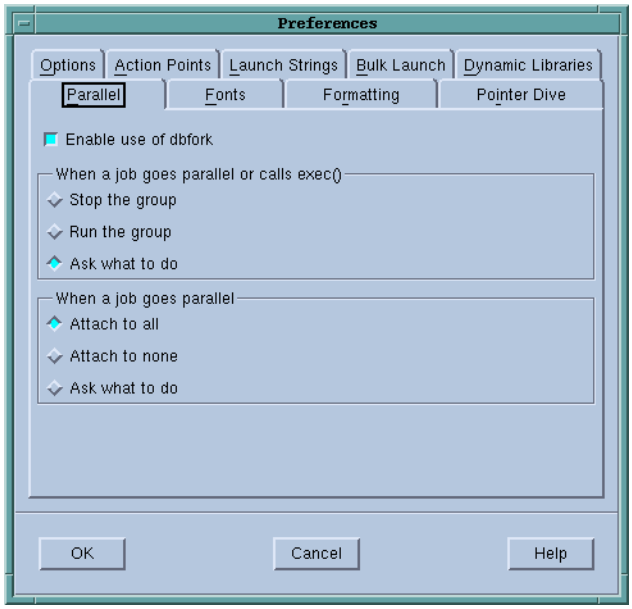
Figure 53: File > Preferences Dialog Box: Dynamic Libraries Page



Parallel

This options on this page let you control if TotalView will stop or continue executing when a processes creates a thread or goes parallel. By telling your program to stop, you can set breakpoints and examine code before execution begins.

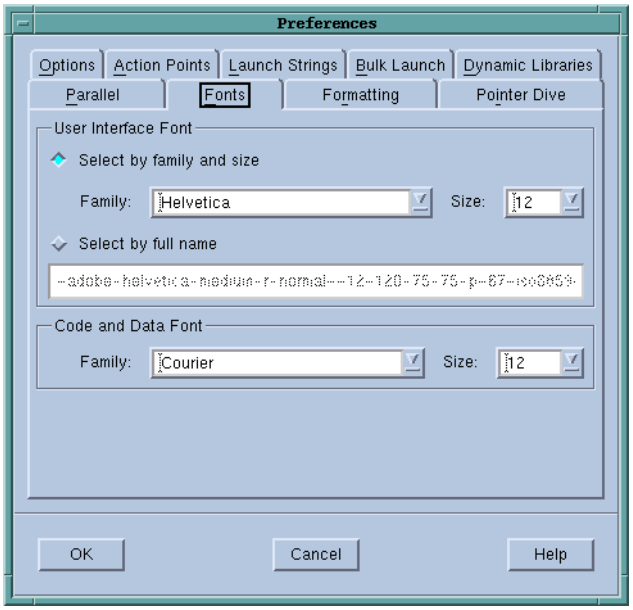
Figure 54: File > Preferences Dialog Box: Parallel Page



Fonts

The options on this page lets you specify the fonts TotalView uses in the user interface and how TotalView displays your code.

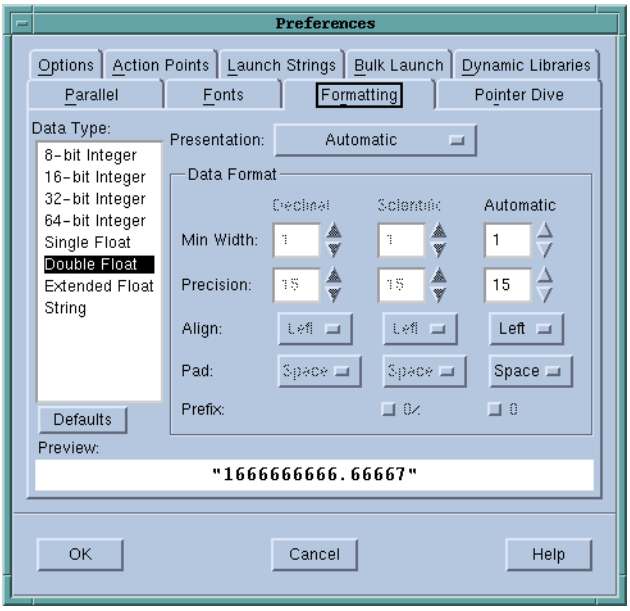
Figure 55: File > Preferences Dialog Box: Fonts Page



Formatting

The options on this page control how TotalView displays your program's variables.

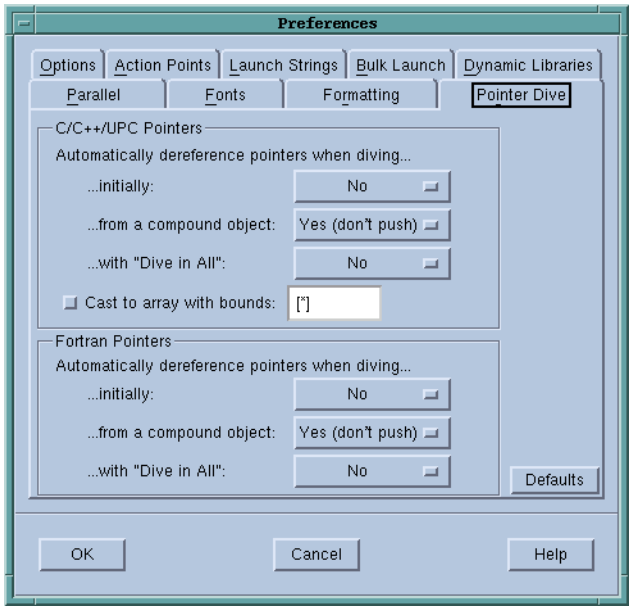
Figure 56: File Preferences Dialog Box: Formatting Page



Pointer Dive

The options on this page control how TotalView dereferences pointers and how it casts pointers to arrays are cast.

Figure 57: File > Preferences Dialog Box: Pointer Dive Page



Setting Preferences, Options, and X Resources

In most cases, preferences are the best way to set many features and characteristics. In some cases, you need have more control. When these situations occur, you can the preferences and other TotalView attributes using variables and command-line options.

Older versions of TotalView did not have a preference system. Instead, you needed to set values in your **.Xdefaults** file or using a command-line option. For example, setting **totalview*autoLoadBreakpoints** to true tells TotalView to automatically load an executable's breakpoint file when it loads an executable. Because you can also set this option as a preference and set it using the CLI **dset** command, this X resource has been *deprecated*.

Deprecated means that while the feature still exists in the current release, there's no guarantee that it will continue to work at a later time. We have deprecated all "totalview" X default options. TotalView still fully supports Visualizer resources. Information on these Visualizer settings is at **http://www.etnus.com/Support/docs/xresources/XResources.html**.

Similarly, documentation for earlier releases told you how to use a command-line option to tell TotalView to automatically load breakpoints, and there were two different command-line options to perform this action. While these methods still work, they are also deprecated.

In some cases, you might set a state for one session or you might override one of your preferences. (A *preference* indicates a behavior that you want to occur in all of your TotalView sessions.) This is the function of the command-line options described in "*TotalView Command Syntax*" in the *TotalView Reference Guide*.

For example, you can use the **–bg** command-line option to set the background color for TotalView windows in the TotalView session just being invoked. TotalView does not remember changes to its default behavior that you make using command-line options. You have to set them again when you start a new session.

Setting Environment Variables

You can set and edit the environment variables that TotalView passes to processes. When TotalView creates a new process, it passes a list of environment variables to the process. You can add to this list using the Environment Page in the **Process > Startup Parameters** Dialog Box.

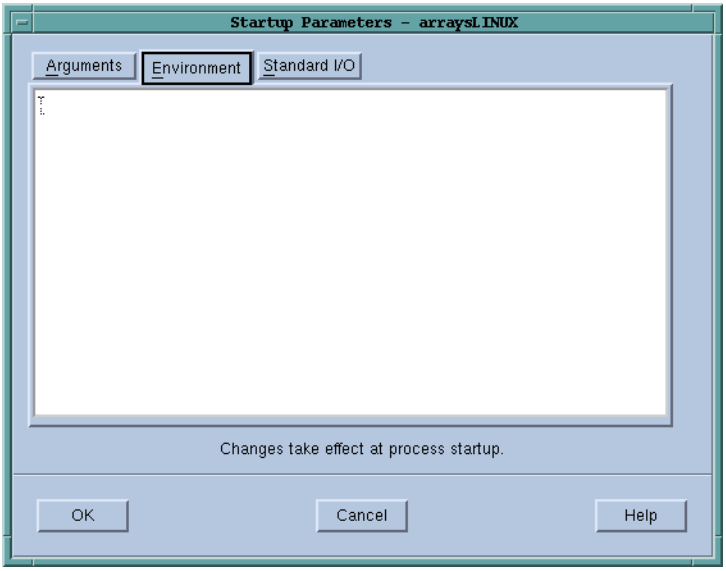
TotalView does not display the variables that were passed to it when you started your debugging session. Instead, this dialog box just displays the variables you added using this command.

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

```
DISPLAY=enterprise:0.0
```

To add, delete, or modify environment variables that you enter, select the Environment Tab from the dialog box displayed when you invoke the **Process > Startup Parameters** command. Here is the Environment Page from this dialog box

Figure 58: Process > Startup Parameters Dialog Box: Environment Page



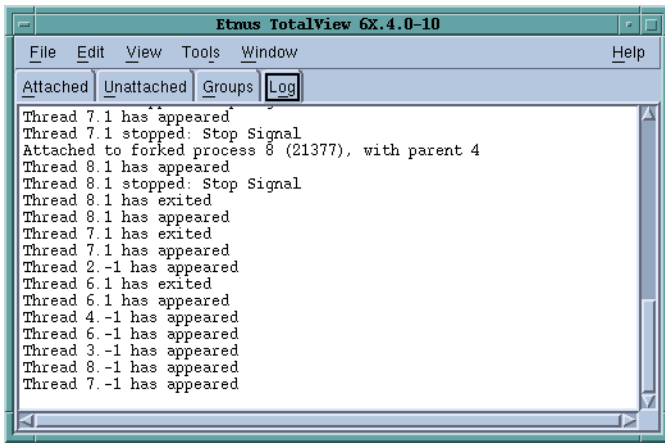
When you enter environment variables, place each one on a separate line. You can alter this information in the following ways:

- Change the name or value of an environment variable by editing a line.
- Add a new environment variable by inserting a new line and specifying a name and value.
- Delete an environment variable that you added by deleting its line.

Monitoring TotalView Sessions

TotalView logs all significant events that occur for all processes being debugged. To view the event log, select the Root Window Log Tab. This page displays a list of these events; for example:

Figure 59: Root Window Log Page



You can set the amount of information TotalView writes to this window using the CLI **dset** command to set the **VERBOSE** variable. If you always want it set to a value, you can set it in your **.tvdrc** file; for example:

```
dset VERBOSE WARNING
```

Setting Up Remote Debugging Sessions

4

This chapter explains how to set up TotalView remote debugging sessions.

This chapter contains the following sections:

- *"Setting Up and Starting the TotalView Debugger Server"* on page 63
- *"Debugging Over a Serial Line"* on page 74

Setting Up and Starting the TotalView Debugger Server

Debugging a remote process with TotalView is only slightly different than debugging a native process. The following are the primary differences:

- TotalView needs to work with a TotalView process that will be running on remote computers. This remote process, which TotalView usually launches, is called the TotalView Debugger Server (**tvdsvr**).
- TotalView performance depends on your network's performance. If the network is overloaded, debugging can be slow.

TotalView can automatically launch **tvdsvr** in one of the following ways:

- It can independently launch a **tvdsvr** on each remote host. This is called *single-process server launch*.
- It can launch all remote processes at the same time. This is called *bulk server launch*.

Because TotalView can automatically launch **tvdsvr**, you usually do not need to do anything special for programs that launch remote processes. When using TotalView, it doesn't matter whether a process is local or remote.

If the default single-process server launch procedure meets your needs and you're not experiencing any problems accessing remote processes from TotalView, you probably do not need the information in this chapter. If you do experience a problem launching the server, check that the **tvdsvr** *process is in your path.*

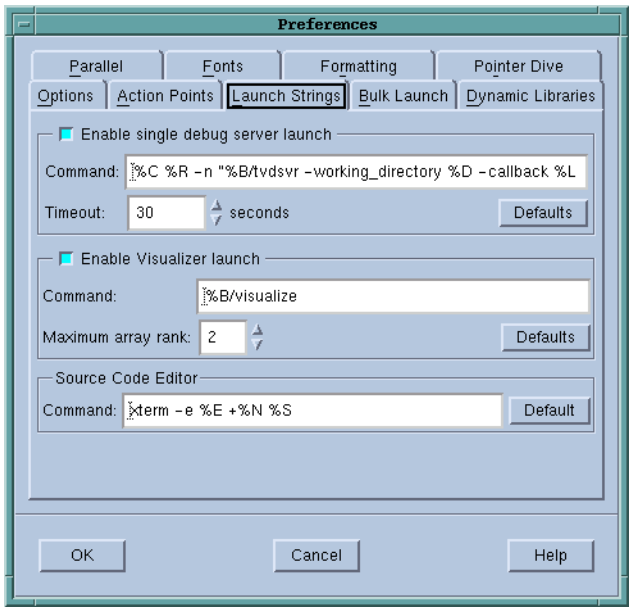
This section contains the following information:

- *"Setting Single-Process Server Launch Options"* on page 64
- *"Setting Bulk Launch Window Options"* on page 65
- *"Starting the Debugger Server Manually"* on page 67
- *"Using the Single-Process Server Launch Command"* on page 68
- *"Bulk Server Launch Setting on an* SGI MIPs *Computer"* on page 69
- *"Setting Bulk Server Launch on an* HP *Alpha Computer"* on page 70
- *"Setting Bulk Server Launch on an* IBM RS/6000 AIX *Computer"* on page 71
- *"Disabling Autolaunch"* on page 71
- *"Changing the Remote Shell Command"* on page 71
- *"Changing Arguments"* on page 72
- *"Autolaunching Sequence"* on page 72

Setting Single-Process Server Launch Options

The **Enable single debug server launch** check box in the Launch Strings Page of the **File** > **Preferences** Dialog Box lets you disable autolaunching, change the command that TotalView uses when it launches remote servers, and alter the amount of time TotalView waits when establishing connections to a **tvdsvr** process. (The **Enable Visualizer launch** and **Source Code Editor** areas are not used when setting launch options.)

Figure 60: File > Preferences: Launch Strings Page



Enable single debug server launch

If you select this check box, TotalView independently launches the TotalView Debugger Server (**tvdsvr**) on each remote system.

CLI: dset TV::server_launch_enabled

Even if you have enabled bulk server launch, you probably also want to enable this option. TotalView uses this launch string after you start TotalView and when you name a host in the File > New Program *Dialog Box or have used the* –remote *command-line option. You only want to disable single server launch when it can't work.*

Command

Enter the command that TotalView will use when it independently launches **tvdsvr**. For information on this command and its options, see "*Using the Single-Process Server Launch Command*" on page 68.

CLI: dset TV::server_launch_string

Timeout

After TotalView automatically launches the **tvdsvr** process, it waits 30 seconds for it to respond. If the connection isn't made in this time, TotalView times out. You can change the length of time by entering a value from 1 to 3600 seconds (1 hour).

CLI: dset TV::server_launch_timeout

If you notice that TotalView fails to launch **tvdsvr** (as shown in the **xterm** window from which you started TotalView) before the timeout expires, click **Yes** in the **Question** Dialog Box that appears.

Defaults

If you make a mistake or decide you want to go back to the TotalView default settings, click the **Defaults** button.

Clicking the **Defaults** button also discards all changes you made using a CLI variable. TotalView doesn't immediately change settings after you click the **Defaults** button; instead, it waits until you click the **OK** button.

Setting Bulk Launch Window Options

The fields in the **File** > **Preferences** Bulk Launch Page let you change the bulk launch command, disable bulk launch, and alter connection timeouts that TotalView uses when it launches **tvdsvr** programs. (The dialog box is shown on the next page.)

CLI: dset TV::bulk_launch_enabled

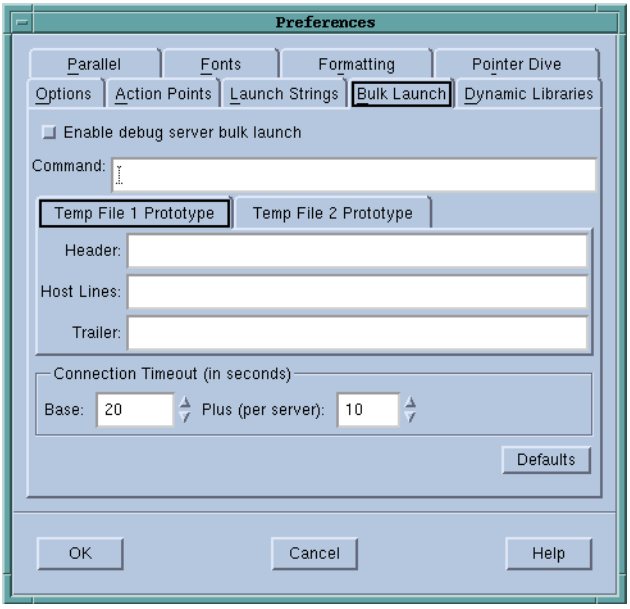
Enable debug server bulk launch

If you select this check box, TotalView uses its bulk launch procedure when launching the TotalView Debugger Server (**tvdsvr**). By default, bulk launch is disabled; that is, TotalView uses its single-server launch procedure.

Command

If you enable bulk launch, TotalView uses this command to launch **tvdsvr**. For information on this command and its options, see "*Bulk Server Launch Setting on*

Figure 61: File > Preferences: Bulk Launch Page



an SGI MIPs *Computer"* on page 69 and "*Setting Bulk Server Launch on an* IBM RS/6000 AIX *Computer"* on page 71.

CLI: dset TV::bulk_launch_string

Temp File 1 Prototype
Temp File 2 Prototype

Both of these tab pages have three fields. These fields let you specify the contents of temporary files that the bulk launch operation uses. For information on these fields, see "*TotalView Debugger Server* (*tvdsvr*) *Command Syntax*" in the *TotalView Reference Guide*.

CLI:
```
dset TV::bulk_launch_tmpfile1_header_line
dset TV::bulk_launch_tmpfile1_host_lines
dset TV::bulk_launch_tmpfile1_trailer_line
dset TV::bulk_launch_tmpfile2_header_line
dset TV::bulk_launch_tmpfile2_host_lines
dset TV::bulk_launch_tmpfile2_trailer_line
```

Connection Timeout (in seconds)

After TotalView launches **tvdsvr** processes, it waits 20 seconds for responses from the process (the **Base** time) plus 10 seconds for each server process being launched. If a connection is not made in this time, TotalView times out.

A **Base** timeout value can range from 1 to 3600 seconds (1 hour). The incremental **Plus** value is from 1 to 360

seconds (6 minutes). See the online Help for information on setting these values.

CLI:	dset TV::bulk_launch_base_timeout dset TV::bulk_launch_incr_timeout

If you notice that TotalView fails to launch **tvdsvr** (as shown in the **xterm** window from which you started the debugger) before the timeout expires, select **Yes** in the **Question** Dialog Box that appears.

Defaults

If you make a mistake or decide you want to go back to the debugger's default settings, click the **Defaults** button.

Clicking **Defaults** also throws away any changes you made using a CLI variable. TotalView doesn't immediately change settings after you click the **Defaults** button; instead, it waits until you click the **OK** button.

Starting the Debugger Server Manually

If TotalView can't automatically launch **tvdsvr**, you can start it manually.

If you use a "*hostname*:*portnumber*" qualifier when opening a remote process, TotalView does not launch a debugger server.

Here are the steps for manually starting **tvdsvr**:

1. Make sure that both bulk launch and single server launch are disabled. To disable bulk launch, click the Bulk Launch Tab in the **File > Preferences** Dialog Box. (You can select this command from the Root Window or the Process Window.) The dialog box shown on the next page appears.
2. Clear the **Enable debug server bulk launch** check box in the Bulk Launch Tab to disable autolaunching and then select **OK**.

CLI:	dset TV::bulk_launch_enabled

3. Select the Server Launch Tab and clear the **Enable single debug server launch** check box.

CLI:	dset TV::server_launch_enabled

4. Log in to the remote computer and start **tvdsvr**:

   tvdsvr –server

   If you don't (or can't) use the default port number (4142), you will need to use the **–port** or **–search_port** options. For details, see "*TotalView Debugger Server (tvdsvr) Command Syntax*" in the *TotalView Reference Guide*.

   After printing the port number and the assigned password, the server begins listening for connections. Be sure to remember the password—you need to enter it in step 5.

Using the –server *option is not secure, other users could connect to your* tvdsvr *process and use your* UNIX UID. *Consequently, this command-line option must be explicitly enabled.* (*Your system administrator usually does this.*) *For details, see* –server *in the "TotalView Command Syntax" chapter of the TotalView Reference Guide.*

5 From the Root Window, select the **File > New Program** command. Type the program's name in the **Executable** field and the *hostname:portnumber* in the **Remote Host** field and then select **OK**.

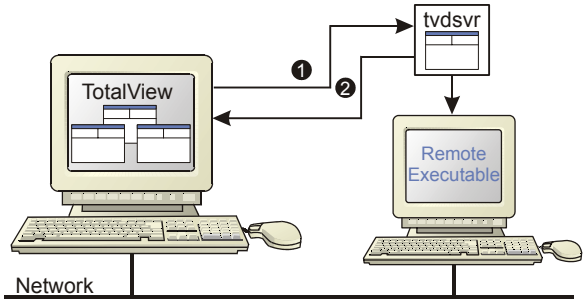
CLI: **dload** *executable* **–r** *hostname*

6 TotalView tries to connect to **tvdsvr**.

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

The following figure summarizes the steps for starting **tvdsvr** manually.

Figure 62: Manual Launching of Debugger Server



❶ Makes connection
❷ Listens

Using the Single-Process Server Launch Command

The following is the default command string that TotalView uses when it automatically launches the debugger server for a single process:

```
%C %R –n "%B/tvdsvr –working_directory %D –callback %L \
        –set_pw %P –verbosity %V %F"
```

where:

%C	Expands to the name of the server launch command being used. On most platforms, this is **rsh**. On HP computers, this command is **remsh**. If the **TVDSVRLAUNCHCMD** environment variable exists, TotalView uses its value instead of a platform-specific default value.
%R	Expands to the host name of the remote computer that you specified in the **File > New Program** or **dload** commands.
%B	Expands to the **bin** directory in which **tvdsvr** is installed.

–n Tells the remote shell to read standard input from **/dev/null**; that is, the process immediately receives an EOF (End-Of-File) signal.

–working_directory %D
Makes **%D** the directory to which TotalView connects. **%D** expands to the absolute path name of the directory.

When you use this option, the host computer and the target computer must mount identical file systems. That is, the path name of the directory to which TotalView connects must be identical on host and target computers.

After changing to this directory, the shell invokes the **tvdsvr** command.

You must make sure that the **tvdsvr** directory is in your path on the remote computer.

–callback %L Establishes a connection from **tvdsvr** to TotalView. **%L** expands to the host name and TCP/IP port number (*hostname*:*portnumber*) on which TotalView is listening for connections from **tvdsvr**.

–set_pw %P Sets a 64-bit password. TotalView must supply this password when **tvdsvr** establishes a connection with it. TotalView expands **%P** to the password that it automatically generates. For more information on this password, see "*TotalView Debugger Server* (*tvdsvr*) *Command Syntax*" in the *TotalView Reference Guide*.

–verbosity %V Sets the verbosity level of the TotalView Debugger Server. **%V** expands to the current TotalView verbosity setting. For information on verbosity, see the "*Variables*" chapter within the *TotalView Reference Guide*.

%F Contains the tracer configuration flags that need to be sent to **tvdsvr** processes. These are system-specific startup options that the **tvdsvr** process needs.

You can also use the **%H** option with this command. See "*Bulk Server Launch Setting on an* SGI MIPs *Computer*" on page 69 for more information.

For information on the complete syntax of the **tvdsvr** command, see "*TotalView Debugger Server* (*tvdsvr*) *Command Syntax*" in the *TotalView Reference Guide*.

Bulk Server Launch Setting on an SGI MIPs Computer

On an SGI computer, the bulk server launch string is as follows:

```
array tvdsvr –working_directory %D –callback_host %H \
        –callback_ports %L –set_pws %P –verbosity %V %F
```

where:

–working_directory %D
Makes **%D** the directory to which TotalView connects. TotalView expand **%D** to this directory's absolute path name.

When you use this option, the host computer and the target computer must mount identical file systems. That is, the path name of the directory to which TotalView connects must be identical on the host and target computers.

After performing this operation, **tvdsvr** starts executing.

–callback_host %H
Names the host upon which TotalView makes this callback. TotalView expands **%H** to the host name of the computer TotalView is running on.

–callback_ports %L
Names the ports on the host computers that TotalView uses for callbacks. TotalView expands **%L** to a comma-separated list of host names and TCP/IP port numbers (*hostname*:*portnumber*,*hostname*:*portnumbe*...) on which TotalView is listening for connections.

–set_pws %P Sets 64-bit passwords. TotalView must supply these passwords when **tvdsvr** establishes the connection with it. **%P** expands to a comma-separated list of 64-bit passwords that TotalView automatically generates. For more information, see "*TotalView Debugger Server* (*tvdsvr*) *Command Syntax*" in the *TotalView Reference Guide*.

–verbosity %V Sets the **tvdsvr** verbosity level. TotalView expands **%V** to the current TotalView verbosity setting.For information on verbosity, see the "*Variables*" chapter within the *TotalView Reference Guide*.

You must enable the use of the **array** command by **tvdsvr** by adding the following information to the **/usr/lib/array/arrayd.conf** file:

```
#
# Command that allow invocation of the TotalView
# Debugger server when performing a Bulk Server Launch.
#
command tvdsvr
   invoke /opt/totalview/bin/tvdsvr %ALLARGS
   user %USER
   group %GROUP
   project %PROJECT
```

If your code is not in **/opt/totalview/bin**, you will need to change this information. For information on the syntax of the **tvdsvr** command, see "*TotalView Debugger Server* (*tvdsvr*) *Command Syntax*" in the *TotalView Reference Guide*.

Setting Bulk Server Launch on an HP Alpha Computer

The following is the bulk launch string on an HP Alpha computer:

```
prun –T –1 tvdsvr –callback_host %H \
        –callback_ports %L –set_pws %P \
        –verbosity %V –working_directory %D %F
```

Information on the options and expansion symbols is in the "*TotalView Debugger Server* (*tvdsvr*) *Syntax*" chapter of the *TotalView Reference Guide*.

Setting Bulk Server Launch on an IBM RS/6000 AIX Computer

The following is the bulk server launch string on an IBM RS/6000 AIX computer:

```
%C %H –n "poe –pgmmodel mpmd –resd no –tasks_per_node 1\
        -procs %N –hostfile %t1 –cmdfile %t2 %F"
```

where the options unique to this command are:

%N The number of servers that TotalView launches.

%t1 A temporary file created by TotalView that contains a list of the hosts on which **tvdsvr** runs. This is the information you enter in the **Temp File 1 Prototype** field on the Bulk Launch Page.

TotalView generates this information by expanding the **%R** symbol. This is the information you enter in the **Temp File 2 Prototype** field on the Bulk Launch Page.

%t2 A file that contains the commands to start the **tvdsvr** processes on each computer. TotalView creates these lines by expanding the following template:

```
tvdsvr –working_directory %D \
   –callback %L –set_pw %P \
   –verbosity %V
```

Information on the options and expansion symbols is in the "*TotalView Debugger Server (tvdsvr) Syntax*" chapter of the *TotalView Reference Guide*.

Disabling Autolaunch

If after changing autolaunching options, TotalView still can't automatically start **tvdsvr**, you must disable autolaunching and start **tvdsvr** manually. Before trying to manually start the server, you must clear the **Enable single debug server launch** check box on the Launch Strings Page of the **File** > **Preferences** Dialog Box.

CLI: dset TV::server_launch_enabled

You can use the procedure described in "*Setting Up and Starting the TotalView Debugger Server*" on page 63 for get the program started. You will also need to enter a host name and port number in the bottom section of the **File** > **New Program** Dialog Box. This disables autolaunching for the current connection.

If you disable autolaunching, you must start **tvdsvr** *before you load a remote executable or attach to a remote process.*

Changing the Remote Shell Command

Some environments require you to create your own autolaunching command. You might do this, for example, if your remote shell command doesn't provide the security that your site requires.

If you create your own autolaunching command, you must use the **tvdsvr –callback** and **–set_pw** command-line options.

If you're not sure whether **rsh** (or **remsh** on HP computers) works at your site, try typing "**rsh** *hostname*" (or "**remsh** *hostname*") from an **xterm** window,

where *hostname* is the name of the host on which you want to invoke the remote process. If the process doesn't just run and instead this command prompts you for a password, you must add the host name of the host computer to your **.rhosts** file on the target computer.

For example, you can use the following combination of the **echo** and **telnet** commands:

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

After **telnet** establishes a connection to the remote host, you can use the **cd** and **tvdsvr** commands directly, using the values of **%D**, **%L**, **%P,** and **%V** that were displayed by the **echo** command; for example:

```
cd directory
tvdsvr -callback hostname:portnumber –set_pw password
```

If your computer doesn't have a command for invoking a remote process, TotalView can't autolaunch the **tvdsvr** and you must disable both single server and bulk server launches.

For information on the **rsh** and **remsh** commands, see the manual page supplied with your operating system.

Changing Arguments

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

For example, if the host computer doesn't mount the same file systems as your target computer, **tvdsvr** might need to use a different path to access the executable being debugged. If this is the case, you can change **%D** to the directory used on the target computer.

If the remote executable reads from standard input, you cannot use the **–n** option with your remote shell command because the remote executable receives an EOF immediately on standard input. If you omit the **–n** command-line option, the remote executable reads standard input from the **xterm** in which you started TotalView. This means that you should invoke **tvdsvr** from another **xterm** window if your remote program reads from standard input. The following is an example:

```
%C %R "xterm -display hostname:0 -e tvdsvr \
   -callback %L -working_directory %D -set_pw %P \
   -verbosity %V"
```

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

Autolaunching Sequence

This section describes the sequence of actions involved in autolaunching. You can skip this section if you aren't having any problems or if you aren't curious.

1. With the **File > New Program** or **dload** commands, you specify the host name of the computer on which you want to debug a remote process, as described in "*Setting Up and Starting the TotalView Debugger Server*" on page 63.
2. TotalView begins listening for incoming connections.

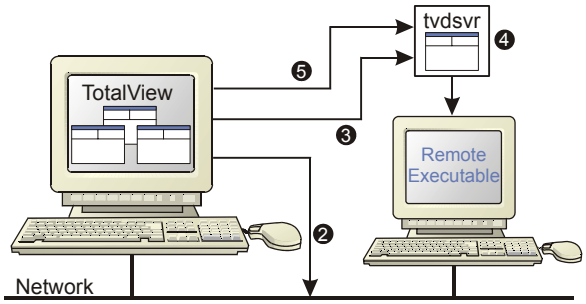
3 TotalView launches the **tvdsvr** process with the server launch command. (See "*Using the Single-Process Server Launch Command*" on page 68 for more information.)

4 The **tvdsvr** process starts on the remote computer.

5 The **tvdsvr** process establishes a connection with TotalView.

The following figure summarizes these actions if your program is launching one server:

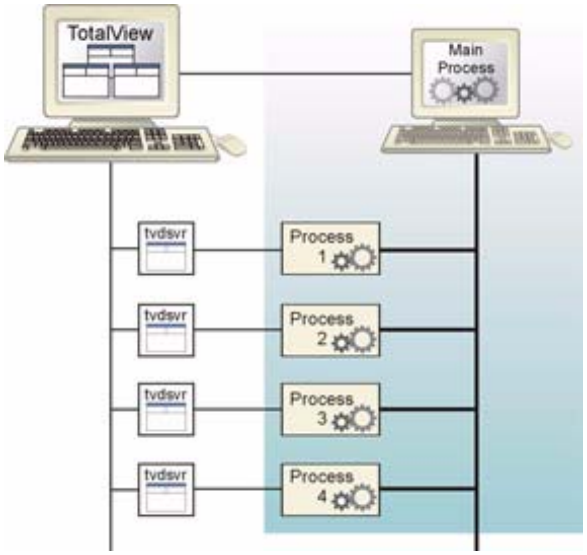
Figure 63: Launching tvdsvr



❷ Listens
❸ Invokes commands
❹ tvdsvr starts
❺ Makes connection

If you have more than one server process, the following figure shows what your environment might look like:

Figure 64: Multiple tvdsvr Processes



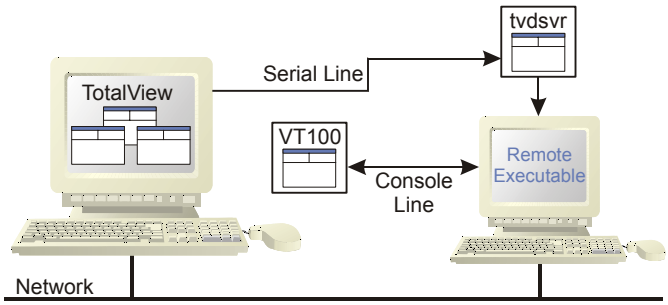
Debugging Over a Serial Line

TotalView lets you debug programs over a serial line as well as TCP/IP sockets. However, if a network connection exists, you probably want to use it to improve performance.

You need two connections to the target computer: one for the console and the other for TotalView. Do not try to use one serial line because TotalView cannot share a serial line with the console.

The following figure illustrates a TotalView debugging session using a serial line. In this example, TotalView is communicating over a dedicated serial line with a TotalView Debugger Server running on the target host. A VT100 terminal is connected to the target host's console line, which lets you type commands on the target host.

Figure 65: TotalView Debugging Session Over a Serial Line



This section contains the following topics:

- *"Starting the TotalView Debugger Server"* on page 74
- *"Starting TotalView on a Serial Line"* on page 75
- *"Using the New Program Window"* on page 75

Starting 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 (**tvdsvr**).

Using the console connected to the target computer, start **tvdsvr** and enter the name of the serial port device on the target computer. Use the following syntax:

tvdsvr –serial *device*[**:baud=**_num_]

where:

device	The name of the serial line device.
num	The serial line's baud rate. If you omit the baud rate, TotalView uses a default value of **38400**.

For example:

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

After it starts, **tvdsvr** waits for TotalView to establish a connection.

Starting TotalView on a Serial Line

Start TotalView on the host computer and include the name of the serial line device. The syntax of this command is as follows:

totalview –serial *device*[**:baud=***num*] *filename*

or

totalviewcli –serial *device*[**:baud=***num*] *filename*

where:

device	The name of the serial line device on the host computer.
num	The serial line's baud rate. If you omit the baud rate, TotalView uses a default value of **38400**.
filename	The name of the executable file.

For example:

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

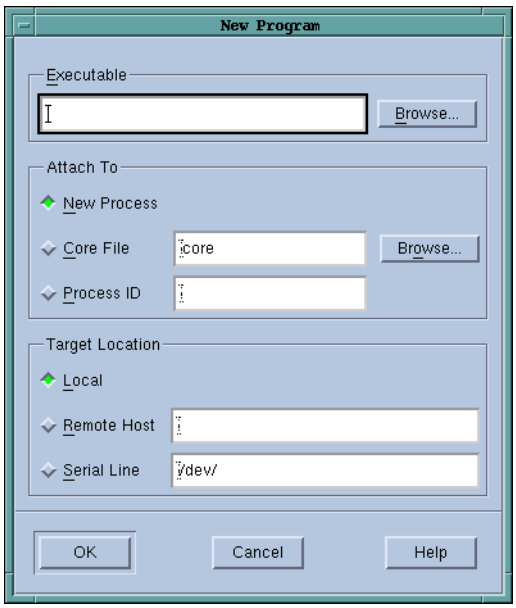
Using the New Program Window

GUI

The following procedure starts a TotalView debugging session over a serial line when you're already in TotalView:

1. Start the TotalView Debugger Server (see "*Starting the TotalView Debugger Server*" on page 74).
2. Select the **File > New Program** command.
   TotalView displays the dialog box shown in the following figure.

Figure 66: File > New Program Dialog Box



3. Type the name of the executable file in the **Executable** field.
   Type the name of the serial line device in the **Serial Line** field.
   Select **OK**.

Setting Up Parallel Debugging Sessions

5

This chapter explains how to set up TotalView parallel debugging sessions for applications that use the parallel execution models that TotalView supports.

This chapter contains the following topics:

- "*Debugging* MPICH *Applications*" on page 78
- "*Debugging* HP *Tru*64 *Alpha* MPI *Applications*" on page 81
- "*Debugging* HP MPI *Applications*" on page 82
- "*Debugging* IBM MPI *Parallel Environment* (PE) *Applications*" on page 83
- "*Debugging* LAM/MPI *Applications*" on page 86
- "*Debugging* QSW RMS *Applications*" on page 87
- "*Debugging* SGI MPI *Applications*" on page 88
- "*Debugging Sun* MPI *Applications*" on page 89
- "*Debugging Open*MP *Applications*" on page 95
- "*Debugging Global Arrays Applications*" on page 100
- "*Debugging* PVM (*Parallel Virtual Machine*) *and* DPVM *Applications*" on page 103
- "*Debugging Shared Memory* (SHMEM) *Code*" on page 109
- "*Debugging* UPC *Programs*" on page 110
- "*Debugging Parallel Applications Tips*" on page 113

This chapter also describes TotalView features that you can use with most parallel models:

- TotalView lets you decide which process you want it to attach to. See "*Attaching to Processes*" on page 113.
- If you're using a messaging system, TotalView displays this information visually as a message queue graph and textually in a message queue window. For more information, see "*Displaying the Message Queue Graph Window*" on page 90.
- See "*Debugging Parallel Applications Tips*" on page 113 for hints on how to approach debugging parallel programs.

Debugging MPICH Applications

To debug Message Passing Interface/Chameleon Standard (MPICH) applications, you must use MPICH version 1.2.3 or later on a homogenous collection of computers. If you need a copy of MPICH, you can obtain it at no cost from Argonne National Laboratory at **www.mcs.anl.gov/mpi**. (We strongly urge that you use a later version of MPICH. The *TotalView Platforms* document has information on versions that work with TotalView.)

The MPICH library should use the **ch_p4**, **ch_p4mpd**, **ch_shmem**, **ch_lfshmem**, or **ch_mpl** devices.

- For networks of workstations, the default MPICH library is **ch_p4**.
- For shared-memory SMP computers, use **ch_shmem**.
- On an IBM SP computer, use the **ch_mpl** device.

The MPICH source distribution includes all of these devices. Choose the device that best fits your environment when you configure and build MPICH.

When configuring MPICH, *you must ensure that the* MPICH *library maintains all of the information that TotalView requires. This means that you must use the* **–enable-debug** *option with the* MPICH **configure** *command.* (*Versions earlier than* 1.2 *used the* **--debug** *option.*) *In addition, the TotalView Release Notes contains information on patching your* MPICH *version* 1.2.3 *distribution.*

For more information, see:

- "*Starting TotalView on an* MPICH *Job*" on page 78
- "*Attaching to an* MPICH *Job*" on page 80
- "*Using* MPICH P4 *procgroup Files*" on page 81

Starting TotalView on an MPICH Job

Before you can bring an MPICH job under the debugger's control, both TotalView and the TotalView server must be in your path. You can do this in a login or shell startup script.

For version 1.1.2, the following command-line syntax starts a job under TotalView control:

mpirun [MPICH-*arguments*] **–tv** *program* [*program-arguments*]

For example:

```
mpirun -np 4 -tv sendrecv
```

The MPICH **mpirun** command obtains information from the **TOTALVIEW** environment variable and then uses this information when it starts the first process in the parallel job.

For Version 1.2.4, the syntax changes to the following:

mpirun –dbg=totalview [*other_mpich-args*] *program* [*program-args*]

For example:

```
mpirun -dbg=totalview -np 4 sendrecv
```

In this case, **mpirun** obtains the information it needs from the **–dbg** command-line option.

In other contexts, setting this environment variable means that you can use different versions of TotalView or pass command-line options to TotalView.

For example, the following is the C shell command that sets the **TOTALVIEW** environment variable so that **mpirun** passes the **–no_stop_all** option to TotalView:

```
setenv TOTALVIEW "totalview -no_stop_all"
```

TotalView begins by starting the first process of your job, the master process, under its control. You can then set breakpoints and begin debugging your code.

On the IBM SP computer with the **ch_mpl** device, the **mpirun** command uses the **poe** command to start an MPI job. While you still must use the MPICH **mpirun** (and its **–tv** option) command to start an MPICH job, the way you start MPICH differs. For details on using TotalView with **poe**, see *"Starting TotalView on a PE Program"* on page 84.

Starting TotalView using the **ch_p4mpd** device is similar to starting TotalView using **poe** on an IBM computer or other methods you might use on Sun and HP platforms. In general, you start TotalView using the **totalview** command, with the following syntax;

totalview mpirun [*totalview_args*] **–a** [*mpich-args*] *program* [*program-args*]

CLI: **totalviewcli mpirun** [*totalview_args*] \
–a [*mpich-args*] *program* [*program-args*]

As your program executes, TotalView automatically acquires the processes that are part of your parallel job as your program creates them. Before TotalView begins to acquire them, it asks if you want to stop the spawned processes. If you click **Yes**, you can stop processes as they are initialized. This lets you check their states or set breakpoints that are unique to the process TotalView automatically copies breakpoints from the master process to the slave processes as it acquires them. Consequently, you don't have to stop them just to set these breakpoints.

If you're using the GUI, TotalView updates the Root Window Attached Page to show these newly acquired processes. For more information, see *"Attaching to Processes"* on page 113.

Attaching to an MPICH Job

TotalView lets you to attach to an MPICH application even if it was not started under the debugger's control.

To attach to an MPICH application:

1 Start TotalView.

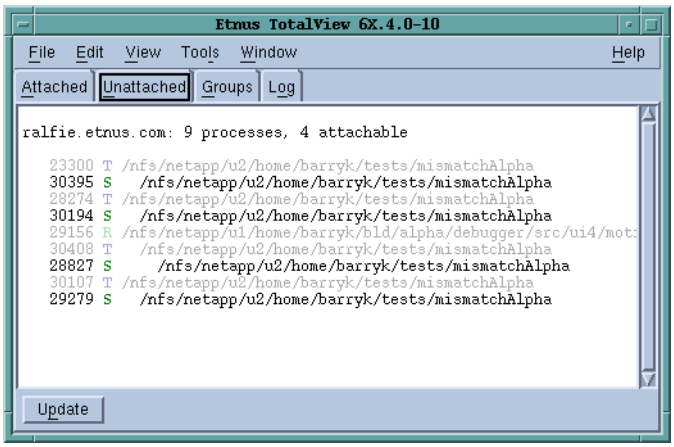
The Root Window Unattached Page displays the processes that are not yet owned.

2 Attach to the first MPICH process in your workstation cluster by diving into it.

CLI: **dattach** *executable pid*

3 On an IBM SP with the **ch_mpi** device, attach to the **poe** process that started your job. For details, see "*Starting TotalView on a* PE *Program*" on page 84. The following figure shows the Unattached Page.

Figure 67: Root Window: Unattached Page



Normally, the first MPICH process is the highest process with the correct program name in the process list. Other instances of the same executable can be:

- The **p4** listener processes if MPICH was configured with **ch_p4**.
- Additional slave processes if MPICH was configured with **ch_shmem** or **ch_lfshmem**.
- Additional slave processes if MPICH was configured with **ch_p4** and has a file that places multiple processes on the same computer.

4 After you attach to your program's processes, TotalView asks if you also want to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPICH processes.

As an alternative, you can use the **Group** > **Attach Subset** command to predefine what TotalView should do. For more information, see "*Attaching to Processes*" on page 113.

In some situations, the processes you expect to see might not exist (for example, they may crash or exit). TotalView acquires all the processes it can

and then warns you if it can not attach to some of them. If you attempt to dive into a process that no longer exists (for example, using a message queue display), TotalView tells you that the process no longer exists.

Using MPICH P4 procgroup Files

If you're using MPICH with a P4 **procgroup** file (by using the **–p4pg** option), you must use the *same* absolute path name in your **procgroup** file and on the **mpirun** command line. For example, if your **procgroup** file contains a different path name than that used in the **mpirun** command, even though this name resolves to the same executable, TotalView assumes that it is a different executable, which causes debugging problems.

The following example uses the same absolute path name on the TotalView command line and in the **procgroup** file:

```
% cat p4group
local 1 /users/smith/mympichexe
bigiron 2 /users/smith/mympichexe
% mpirun -p4pg p4group -tv /users/smith/mympichexe
```

In this example, TotalView does the following:

1. Reads the symbols from **mympichexe** only once.
2. Places MPICH processes in the same TotalView share group.
3. Names the processes **mypichexe.0**, **mympichexe.1**, **mympichexe.2**, and **mympichexe.3**.

If TotalView assigns names such as **mympichexe<mympichexe>.0**, a problem occurred and you need to compare the contents of your **procgroup** file and **mpirun** command line.

Debugging HP Tru64 Alpha MPI Applications

To use TotalView with HP Tru64 Alpha MPI applications, you must use HP Tru64 Alpha MPI version 1.7 or later.

Starting TotalView on an HP Alpha MPI Job

In most cases, you start an HP Alpha MPI program by using the **dmpirun** command. The command for starting an MPI program under the debugger's control is similar; it uses the following syntax:

{ totalview | totalviewcli } dmpirun –a *dmpirun-command-line*

This command invokes TotalView and tells it to show you the code for the main program in **dmpirun**. Since you're not usually interested in debugging this code, use the **Process > Go** command to let the program run.

CLI: **dfocus p dgo**

The **dmpirun** command runs and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them.

Problems can occur if you rerun HP *Alpha* MPI *programs that are under TotalView control because resource allocation issues exist within* HP *Alpha* MPI. *The* HP *Alpha* MPI *documentation contains information on using* **mpiclean** *to clean up the* MPI *system state*.

Attaching to an HP Alpha MPI Job

To attach to a running HP Alpha MPI job, attach to the **dmpirun** process that started the job. The procedure for attaching to a **dmpirun** process is the same as the procedure for attaching to other processes. For details, see *"Attaching to Processes"* on page 42. You can also use the **Group > Attach Subset** command which is discussed in *"Attaching to Processes"* on page 113.

After you attach to the **dmpirun** process, TotalView asks if you also want to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPICH processes.

Debugging HP MPI Applications

You can debug HP MPI applications on a PA-RISC 1.1 or 2.0 processor. To use TotalView with HP MPI applications, you must use HP MPI versions 1.6 or 1.7.

Starting TotalView on an HP MPI Job

TotalView lets you start an MPI program in one of the following ways:

{ totalview | totalviewcli } *program* **–a** *mpi-arguments*
This command tells TotalView to start the MPI process. TotalView then shows you the machine code for the HP MPI **mpirun** executable.

CLI:	dfocus p dgo

mpirun *mpi-arguments* **–tv –f** *startup_file*
This command tells MPI to start TotalView and then start the MPI processes as they are defined in the *startup_file* script. This file names the processes that MPI starts. Typically, this file has contents that are similar to:

```
-h localhost –np 1 sendrecv
-h localhost –np 1 sendrecva
```

In this example, **sendrecv** and **sendrecva** are two different executable programs.

Your HP MPI documentation describes the contents of this startup file.

mpirun *mpi-arguments* **–tv** *program*
This command tells MPI to start TotalView.

Just before **mpirun** starts your MPI processes, TotalView acquires them and asks if you want to stop the processes before they start executing. If you

answer **yes**, TotalView halts them before they enter the **main()** routine. You can then create breakpoints.

Attaching to an HP MPI Job

To attach to a running HP MPI job, attach to the HP MPI **mpirun** process that started the job. The procedure for attaching to an **mpirun** process is the same as the procedure for attaching to any other process. For details, see *"Attaching to Processes"* on page 42.

After TotalView attaches to the HP MPI **mpirun** process, it displays the same dialog box as it does with MPICH. (See step 4 on page 80 of *"Attaching to an* MPICH *Job"* on page 80.)

Debugging IBM MPI Parallel Environment (PE) Applications

You can debug IBM MPI Parallel Environment (PE) applications on the IBM RS/6000 and SP platforms.

To take advantage of the debugger's ability to automatically acquire processes, you must be using release 3,1 or later of the Parallel Environment for AIX.

Topics in this section are:

- *"Preparing to Debug a* PE *Application"* on page 83
- *"Starting TotalView on a* PE *Program"* on page 84
- *"Setting Breakpoints"* on page 84
- *"Starting Parallel Tasks"* on page 85
- *"Attaching to a* PE *Job"* on page 85

Preparing to Debug a PE Application

The following sections describe what you must do before TotalView can debug a PE application.

Using Switch-Based Communications

If you're using switch-based communications (either IP *over the switch* or *user space*) on an SP computer, you must configure your PE debugging session so that TotalView can use IP *over the switch* for communicating with the TotalView Debugger Server (**tvdsvr**). Do this by setting the **–adapter_use** option to **shared** and the **–cpu_use** option to **multiple**, as follows:

- If you're using a PE host file, add **shared multiple** after all host names or pool IDs in the host file.
- Always use the following arguments on the **poe** command line:
  `-adapter_use shared -cpu_use multiple`

If you don't want to set these arguments on the **poe** command line, set the following environment variables before starting **poe**:

```
setenv MP_ADAPTER_USE shared
setenv MP_CPU_USE multiple
```

When using IP *over the switch*, the default is usually **shared adapter use** and **multiple cpu use**; we recommend that you set them explicitly using one of these techniques. You must run TotalView on an SP or SP2 node. Since TotalView will be using IP *over the switch* in this case, you cannot run TotalView on an RS/6000 workstation.

Performing a Remote Login

You must be able to perform a remote login using the **rsh** command. You also need to enable remote logins by adding the host name of the remote node to the **/etc/hosts.equiv** file or to your **.rhosts** file.

When the program is using switch-based communications, TotalView tries to start the TotalView Debugger Server by using the **rsh** command with the switch host name of the node.

Setting Timeouts

If you receive communications timeouts, you can set the value of the MP_TIMEOUT environment variable; for example:

```
setenv MP_TIMEOUT 1200
```

If this variable isn't set, TotalView uses a **timeout** value of 600 seconds.

Starting TotalView on a PE Program

The following is the syntax for running Parallel Environment (PE) programs from the command line:

program [*arguments*] [*pe_arguments*]

You can also use the **poe** command to run programs as follows:

poe *program* [*arguments*] [*pe_arguments*]

If, however, you start TotalView on a PE application, you must start **poe** as the debugger's target using the following syntax:

{ totalview | totalviewcli } poe –a *program* [*arguments*] [PE_*arguments*]

For example:

```
totalview poe -a sendrecv 500 -rmpool 1
```

Setting Breakpoints

After TotalView is running, start the **poe** process using the **Process > Go** command.

CLI:	dfocus p dgo

TotalView responds by displaying a dialog box—in the CLI, it prints a question—that asks if you want to stop the parallel tasks.

If you want to set breakpoints in your code before they begin executing, answer **Yes**. TotalView initially stops the parallel tasks, which also allows you to set breakpoints. You can now set breakpoints and control parallel tasks in the same way as any process controlled by TotalView.

If you have already set and saved breakpoints with the **Action Point** > **Save All** command, and you want to reload the file, answer **No**. After TotalView loads these saved breakpoints, the parallel tasks begin executing.

CLI: **dactions –save** *filename*
dactions –load *filename*

Starting Parallel Tasks

After you set breakpoints, you can start all of the parallel tasks with the Process Window **Group** > **Go** command.

CLI: **dfocus G dgo**
Abbreviation: G

No parallel tasks reach the first line of code in your main routine until all parallel tasks start.

Be very cautious in placing breakpoints at or before a line that calls **MPI_Init()** or **MPL_Init()** because timeouts can occur while your program is being initialized. After you allow the parallel processes to proceed into the **MPI_Init()** or **MPL_Init()** call, allow all of the parallel processes to proceed through it within a short time. For more information on this, see *"Avoid unwanted timeouts"* on page 118.

Attaching to a PE Job

To take full advantage of the debugger's **poe**-specific automation, you need to attach to **poe** itself, and let TotalView automatically acquire the **poe** processes on all of its nodes. In this way, TotalView acquires the processes you want to debug.

Attaching from a Node Running poe

To attach TotalView to **poe** from the node running **poe**:

1. Start TotalView in the directory of the debug target.
   If you can't start TotalView in the debug target directory, you can start TotalView by editing the TotalView Debugger Server (**tvdsvr**) command line before attaching to **poe**. See *"Using the Single-Process Server Launch Command"* on page 68.
2. In the Root Window Unattached Page, find the **poe** process list, and attach to it by diving into it. When necessary, TotalView launches TotalView Debugger Servers. TotalView also updates the Root Window Attached Page and opens a Process Window for the **poe** process.

   CLI: **dattach poe** *pid*

3. Locate the process you want to debug and dive on it. TotalView responds by opening a Process Window for it.

If your source code files are not displayed in the Source Pane, you might not have told TotalView where these files reside. You can fix this by invoking the **File** > **Search Path** command to add directories to your search path.

Attaching from a Node Not Running poe

The procedure for attaching TotalView to **poe** from a node that is not running **poe** is essentially the same as the procedure for attaching from a node that is running **poe**. Since you did not run TotalView from the node running **poe** (the startup node), you won't be able to see **poe** on the process list in your Root Window Attached Page and you won't be able to start it by diving into it.

To place **poe** in this list:

1. Connect TotalView to the startup node. For details, see "*Setting Up and Starting the TotalView Debugger Server*" on page 63 and "*Attaching to Processes*" on page 42.
2. Select the Root Window Unattached Page, and then invoke the **Window** > **Update** command.
3. Look for the process named **poe** and continue as if attaching from a node that is running **poe**.

CLI: **dattach -r** *hostname* **poe** *poe-pid*

Debugging LAM/MPI Applications

The following is a description of the LAM/MPI implementation of the MPI standard. Here are the first two paragraphs of Chapter 2 of the "LAM/MPI *User's Guide*," version 7.0. The URL for this document is: **http://www.lam-mpi.org/download/files/7.0.5-user.pdf**.

> "LAM/MPI is a high-performance, freely available, open source implementation of the MPI standard that is researched, developed, and maintained at the Open Systems Lab at Indiana University. LAM/MPI supports all of the MPI-1 Standard and much of the MPI-2 standard. More information about LAM/MPI, including all the source code and documentation, is available from the main LAM/MPI web site.
>
> "LAM/MPI is not only a library that implements the mandated MPI API, but also the LAM run-time environment: a user-level, daemon-based run-time environment that provides many of the services required by MPI programs. Both major components of the LAM/MPI package are designed as component frameworks—extensible with small modules that are selectable (and configurable) at run-time. ...

You debug a LAM/MPI program in a similar way to how you debug most MPI programs. Use the following syntax if TotalView is in your path:

mpirun tv mpirun *args prog prog_args*

As an alternative, you can invoke TotalView on **mpirun**:

totalview mpirun –a *prog prog_args*

The LAN/MPI *User's Guide* discusses how to use TotalView to debug LAM/MPI programs.

Debugging QSW RMS Applications

TotalView supports automatic process acquisition on AlphaServer SC systems and 32-bit Red Hat Linux systems that use Quadrics RMS resource management system with the QSW switch technology.

Message queue display is only supported if you are running version 1, patch 2 or later, of AlphaServer SC.

Starting TotalView on an RMS Job

To start a parallel job under the debugger's control, use TotalView as if you were debugging **prun**:

{ **totalview** | **totalviewcli** } **prun** –**a** *prun-command-line*

TotalView starts and shows you the machine code for RMS **prun**. Since you're not usually interested in debugging this code, use the **Process** > **Go** command to let the program run.

CLI: **dfocus p dgo**

The RMS **prun** command executes and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at startup. If you answer **yes**, TotalView halts them before they enter the main program. You can then create breakpoints.

Attaching to an RMS Job

To attach to a running RMS job, attach to the RMS **prun** process that started the job.

You attach to the **prun** process the same way you attach to other processes. For details on attaching to processes, see "*Attaching to Processes*" on page 42.

After you attach to the RMS **prun** process, TotalView asks if you also want to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPI processes.

As an alternative, you can use the **Group** > **Attach Subset** command to predefine what TotalView should do. For more information, see "*Attaching to Processes*" on page 113.

Debugging SGI MPI Applications

TotalView can acquire processes started by SGI MPI applications. This MPI is part of the Message Passing Toolkit (MPT) 1.3 and 1.4 packages. TotalView can display the Message Queue Graph Window for these releases. See "*Displaying the Message Queue Graph Window*" on page 90 for message queue display.

Starting TotalView on an SGI MPI Job

You normally start SGI MPI programs by using the **mpirun** command. You use a similar command to start an MPI program under the TotalView control, as follows:

{ **totalview** | **totalviewcli** } **mpirun** –**a** *mpirun-command-line*

This invokes TotalView and tells it to show you the machine code for **mpirun**. Since you're not usually interested in debugging this code, use the **Process** > **Go** command to let the program run.

CLI: **dfocus p dgo**

The SGI MPI **mpirun** command runs and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at startup. If you answer **Yes**, TotalView halts them before they enter the main program. You can then create breakpoints.

If you set a verbosity level that allows informational messages, TotalView also prints a message that shows the name of the array and the value of the array services handle (**ash**) to which it is attaching.

Attaching to an SGI MPI Job

To attach to a running SGI MPI program, attach to the SGI MPI **mpirun** process that started the program. The procedure for attaching to an **mpirun** process is the same as the procedure for attaching to any other process. For details, see "*Attaching to Processes*" on page 42.

After you attach to the **mpirun** process, TotalView asks if you also want to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPICH processes.

As an alternative, you can use the **Group** > **Attach Subset** command to predefine what TotalView will do. For more information, see "*Attaching to Processes*" on page 113.

Debugging Sun MPI Applications

TotalView can debug a Sun MPI program and can display Sun MPI message queues. This section describes how to perform *job startup* and *job attach* operations.

To start a Sun MPI application, use the following procedure.

1. Type the following command:

   totalview mprun [*totalview_args*] **–a** [*mpi_args*]

   For example:

   ```
   totalview mprun –g blue –a –np 4 /usr/bin/mpi/conn.x
   ```

   CLI: **totalviewcli mprun** [*totalview_args*] **–a** [*mpi_args*]

   When the TotalView Process Window appears, select the **Go** button.

   CLI: **dfocus p dgo**

   TotalView may display a dialog box with the following text:

   ```
   Process mprun is a parallel job. Do you want to stop
   the job now?
   ```

2. If you compiled using the **–g** option, click **Yes** to tell TotalView to open a Process Window that shows your source. All processes are halted.

Attaching to a Sun MPI Job

To attach to an already running **mprun** job:

1. Find the host name and process identifier (PID) of the **mprun** job by typing **mpps –b**. For more information, see the **mpps(1M)** manual page.

   The following is sample output from this command:

   ```
   JOBNAME      MPRUN_PID    MPRUN_HOST
   cre.99       12345        hpc-u2-9
   cre.100      12601        hpc-u2-8
   ```

2. After selecting **File** > **New Program**, type **mprun** in the **Executable** field and type the PID in the **Process ID** field.

   CLI: **dattach** *mprun mprun-pid*

   For example:

   dattach mprun 12601

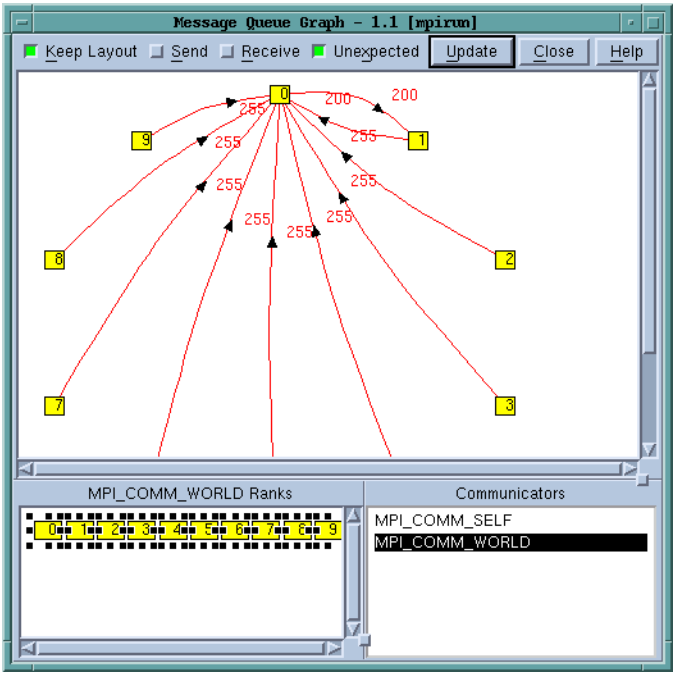
3. If TotalView is running on a different node than the **mprun** job, enter the host name in the **Remote Host** field.

   CLI: **dattach –r** *host-name* **mprun** *mprun-pid*

Displaying the Message Queue Graph Window

TotalView can graphically display your MPI program's message queue state. If you select the Process Window **Tools > Message Queue Graph** command, TotalView displays a window with a large empty area. After you select the ranks you want to monitor, the type of messages, and message states, TotalView updates this window to show the current queue state.

Figure 68: Tools > Message Queue Graph Window



The numbers in the boxes indicate the MPI message tag number. Diving on a box tells TotalView to open a Process Window for that process.

The numbers next to the arrows indicate the number of messages that existed when TotalView created the graph. Diving on an arrow tells TotalView to display its **Tools > Message Queue** Window, which has detailed information about the messages. If TotalView has not attached to a process, it displays this information in a grey box.

The colors used to draw the lines and arrows have the following meaning:

- Green: sent messages
- Blue: received messages
- Red: unexpected messages

The message queue graph shows your program's state at a particular instant. Selecting the **Update** button tells TotalView to fetch new information and redraw the graph.

You can use the Message Queue Graph Window in many ways, including the following:

- Pending messages often indicate that a process can't keep up with the amount of work it is expected to perform. These messages indicate places where you may be able to improve your program's efficiency.
- Unexpected messages can indicate that something is wrong with your program because the receiving process doesn't know how to process the message. The red lines indicate unexpected messages.
- After a while, the shape of the graph tends to tell you something about how your program is executing. If something doesn't look right, you might want to determine why.
- You can change the shape of the graph by dragging nodes or arrows. This is often useful when you're comparing sets of nodes and their messages with one another. TotalView doesn't remember the places to which you have dragged the nodes and arrows. This means that if you select the **Display** button after you arrange the graph, your changes are lost.

Displaying the Message Queue

The **Tools > Message Queue** Window displays your MPI program's message queue state textually. This can be useful when you need to find out why a deadlock occurred.

To use the message queue display feature, you must be using one of the following MPI versions:

- MPICH version 12.3 or later.
- HP Alpha MPI (DMPI) version 1.8, 1.9, or 1.96.
- HP HP-UX version 1.6 or 1.7.
- IBM MPI Parallel Environment (PE) version 3.1 or 3.2, but only programs using the threaded IBM MPI libraries. MQD is not available with earlier releases, or with the non-thread-safe version of the IBM MPI library. Therefore, to use TotalView MQD with IBM MPI applications, you must use the **mpcc_r**, **mpxlf_r**, or **mpxlf90_r** compilers to compile and link your code.
- Sun MPI. Sun MPI does not implement message queue support for blocking MPI send and receive operations.
- SGI MPI Message Passing Toolkit (MPT) release 1.3 or 1.4 if you want to see the message queue display.

For more information, see:

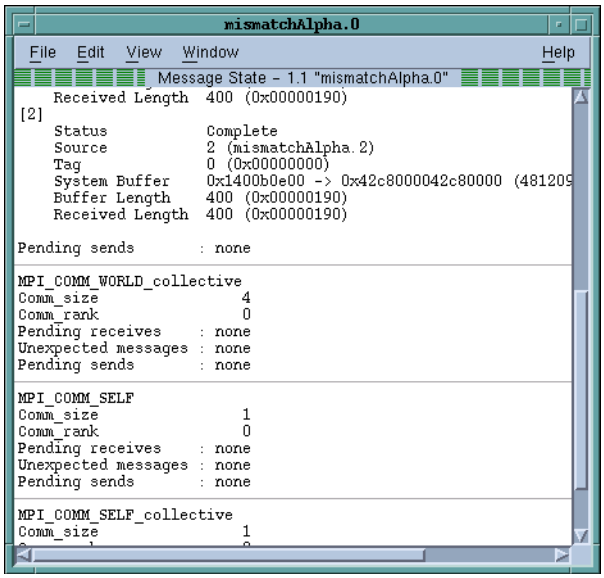
- *"About the Message Queue Display"* on page 91
- *"Using Message Operations"* on page 92

About the Message Queue Display

After an MPI process returns from the call to **MPI_Init()**, you can display the internal state of the MPI library by selecting the **Tools > Message Queue** command. (This window is shown on the next page.)

This window displays the state of the process's MPI communicators. If user-visible communicators are implemented as two internal communica-

Figure 69: Message Queue Window



tor structures, TotalView displays both of them. One is used for point-to-point operations and the other is used for collective operations.

You cannot edit any of the fields in the Message Queue Window.

The contents of the Message Queue Window are only valid when a process is stopped.

Using Message Operations

GUI

For each communicator, TotalView displays a list of pending receive operations, pending unexpected messages, and pending send operations. Each operation has an index value displayed in brackets ([*n*]). The online Help for this window contains a description of the fields that you can display.

For more information, see:

- "*Diving on* MPI *Processes*" on page 92
- "*Diving on* MPI *Buffers*" on page 93
- "*About Pending Receive Operations*" on page 93
- "*About Unexpected Messages*" on page 93
- "*About Pending Send Operations*" on page 94

Diving on MPI Processes

To display more detail, you can dive into fields in the Message Queue Window. When you dive into a process field, TotalView does one of the following:

- Raises its Process Window if it exists.
- Sets the focus to an existing Process Window on the requested process.
- Creates a new Process Window for the process if a Process Window doesn't exist.

Diving on MPI Buffers

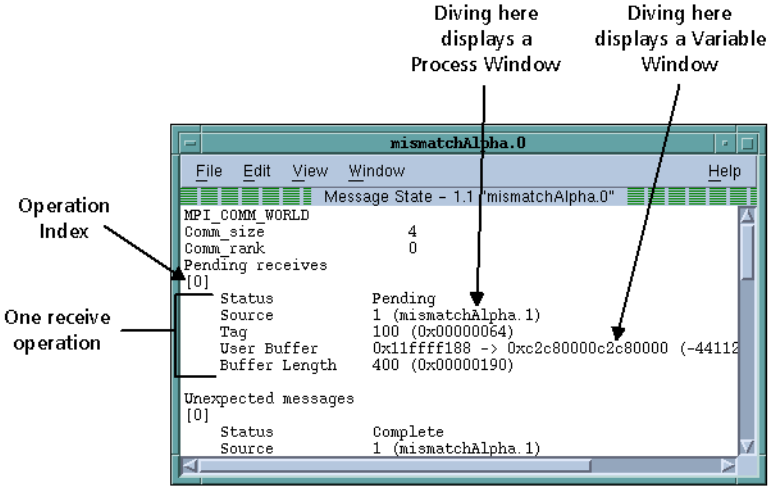
When you dive into the buffer fields, TotalView opens a Variable Window. It also guesses what the correct format for the data should be based on the buffer length and the data alignment. You can edit the **Type** field within the Variable Window, if necessary.

TotalView doesn't use the MPI *data type to set the buffer type.*

About Pending Receive Operations

TotalView displays each pending receive operation in the **Pending receives** list. The following figure shows an example of an MPICH pending receive operation.

Figure 70: Message Queue Window Showing Pending Receive Operation



TotalView displays all receive operations maintained by the IBM MPI *library. Set the environment variable* MP_EUIDEVELOP *to the value* DEBUG *if you want blocking operations to be visible; otherwise, the library only maintains nonblocking operations. For more details on the* MP_EUIDEVELOP *environment variable, see the* IBM *Parallel Environment Operations and Use manual.*

About Unexpected Messages

The **Unexpected messages** portion of the **Message Queue** Window shows information for retrieved and enqueued messages that are not yet matched with a receive operation.

Some MPI libraries, such as MPICH, only retrieve messages that have already been received as a side effect of calls to functions such as **MPI_Recv()** or **MPI_Iprobe()**. (In other words, while some versions of MPI may know about the message, the message may not yet be in a queue.) This means that TotalView can't list a message until after the destination process makes a call that retrieves it.

About Pending Send Operations

TotalView displays each pending send operation in the **Pending sends** list.

MPICH does not normally keep information about pending send operations. If you want to see them, start your program under the debugger's control and use the **mpirun –ksq** or **–KeepSendQueue** command.

Depending on the device for which MPICH was configured, blocking send operations may or may not be visible. However, if TotalView doesn't display them, you can see that these operations occurred because the call is in the stack backtrace.

If you attach to an MPI program that isn't maintaining send queue information, TotalView displays the following message:

```
Pending sends : no information available
```

Troubleshooting MPI Applications

If you can't successfully start TotalView on MPI programs, check the following:

- Can you successfully start MPICH programs without TotalView?

  The MPICH code contains some useful scripts that let you verify that you can start remote processes on all of the computers in your computers file. (See **tstmachines** in **mpich/util**.)
- You won't get a message queue display if you get the following warning:

  ```
  The symbols and types in the MPICH library used by
  TotalView to extract the message queues are not as
  expected in the image <your image name>. This is probably
  an MPICH version or configuration problem.
  ```

  You need to check that you are using MPICH Version 1.1.0 or later and that you have configured it with the **–debug** option. (You can check this by looking in the **config.status** file at the root of the MPICH directory tree.)
- Does the TotalView Debugger Server (**tvdsvr**) fail to start?

  tvdsvr must be in your **PATH** when you log in. Remember that TotalView uses **rsh** to start the server, and that this command doesn't pass your current environment to remotely started processes.
- Make sure you have the correct MPI version and have applied all required patches. See the *TotalView Release Notes* for up-to-date information.
- Under some circumstances, MPICH kills TotalView with the **SIGINT** signal. You can see this behavior when you use the **Group** > **Delete** command as the first step in restarting an MPICH job.

CLI:	dfocus g ddelete

  If TotalView exits and terminates abnormally with a **Killed** message, try setting the **TV::ignore_control_c** variable to true.

Debugging OpenMP Applications

TotalView supports many OpenMP C and Fortran compilers. Supported compilers and architectures are listed in the *TotalView Platforms* document, which is on our Web site.

The following are some of the features that TotalView supports:

- Source-level debugging of the original OpenMP code.
- The ability to plant breakpoints throughout the OpenMP code, including lines that are executed in parallel.
- Visibility of OpenMP worker threads.
- Access to **SHARED** and **PRIVATE** variables in OpenMP PARALLEL code.
- A stack-back link token in worker threads' stacks so that you can find their master stack.
- Access to OMP THREADPRIVATE data in code compiled by the IBM and Guide, SGI IRIX, and HP Alpha compilers.

The code examples used in this section are included in the TotalView distribution in the **examples/omp_simplef** file.

On the SGI IRIX *platform, you must use the* MIPS*pro* 7.3 *compiler or later to debug* *Open*MP.

Topics in this section are:

- "*Debugging Open*MP *Programs*" on page 95
- "*Viewing Open*MP *Private and Shared Variables*" on page 97
- "*Viewing Open*MP THREADPRIVATE *Common Blocks*" on page 98
- "*Viewing the Open*MP *Stack Parent Token Line*" on page 100

Debugging OpenMP Programs

The way in which you debug OpenMP code is similar to the way you debug multithreaded code. The major differences are related to the way the OpenMP compiler alters your code. These alterations include:

- The most visible transformation is *outlining*. The compiler pulls the body of a parallel region out of the original routine and places it in an *outlined routine*. In some cases, the compiler generates multiple outlined routines from a single parallel region. This allows multiple threads to execute the parallel region.

  The outlined routine's name is based on the original routine's name. In most cases, the compiler adds a numeric suffix.
- The compiler inserts calls to the OpenMP runtime library.
- The compiler splits variables between the original routine and the outlined routine. Normally, shared variables reside in the master thread's original routine, and private variables reside in the outlined routine.
- The master thread creates threads to share the workload. As the master thread begins to execute a parallel region in the OpenMP code, it creates the worker threads, dispatches them to the outlined routine, and then calls the outlined routine itself.

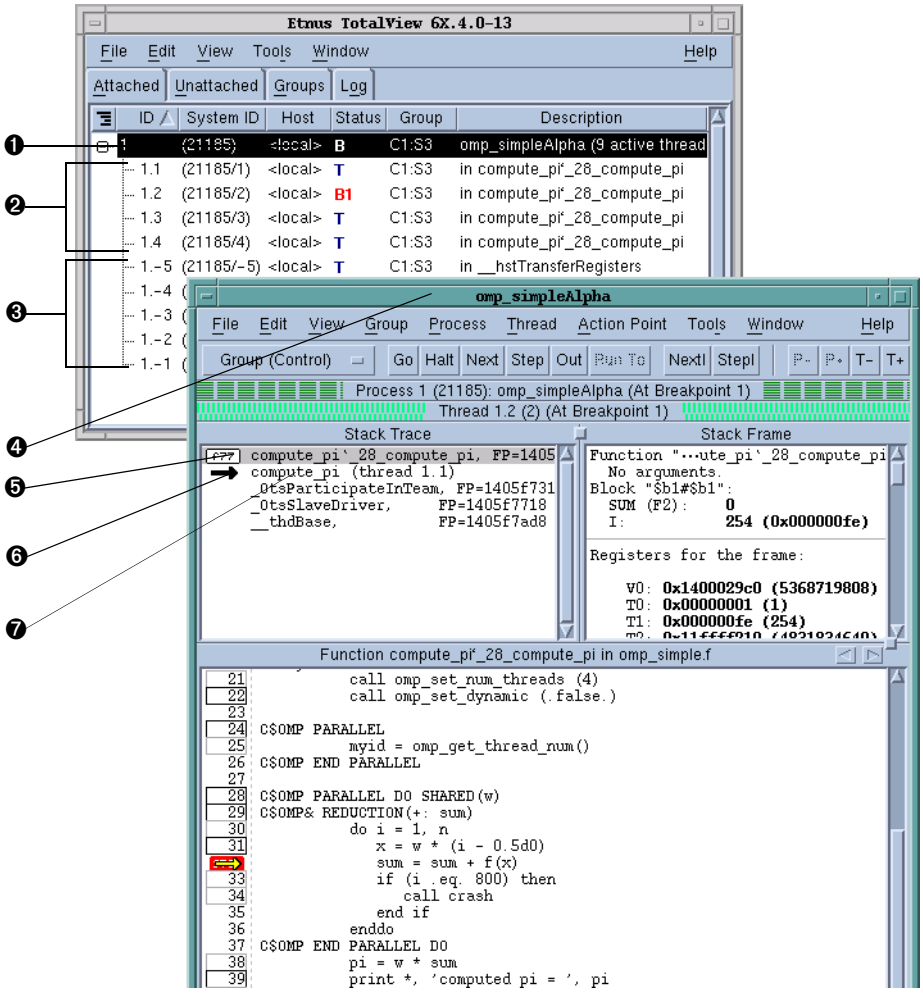
About TotalView OpenMP Features

TotalView interprets the changes that the OpenMP compiler makes to your code so that it can display your program in a coherent way. Here are some things you should know:

- The compiler can generate multiple outlined routines from a single parallel region. This means that a single line of source code can generate multiple blocks of machine code inside different functions.
- You can't single step into or out of a parallel region. Instead, set a breakpoint inside the parallel region and let the process run to it. After execution reaches the parallel region, you can single step in it.
- OpenMP programs are multithreaded programs, so the rules for debugging multithreaded programs apply.

The following figure shows a sample OpenMP debugging session.

Figure 71: Sample OpenMP Debugging Session



❶ OpenMP master thread
❷ OpenMP worker threads
❸ Manager threads (don't touch these threads)
❹ Slave Thread Window
❺ Original routine name
❻ Stack parent token (select or dive to view master)
❼ Outlined routine name

About OpenMP Platform Differences

In general, TotalView smooths out the differences that occur when you execute OpenMP platforms on different platforms. The following list discusses these differences:

- The OpenMP master thread has logical thread ID number 1. The OpenMP worker threads have a logical thread ID number greater than 1.
- When you stop the OpenMP master thread in a **PARALLEL DO** outlined routine, the stack backtrace shows the following call sequence:
  - ➤ The outlined routine called from.
  - ➤ The OpenMP runtime library called from.
  - ➤ The original routine (containing the parallel region).
- When you stop the OpenMP worker threads in a **PARALLEL DO** outlined routine, the stack backtrace shows the following call sequence:
  - ➤ Outlined routine called from the special stack parent token line.
  - ➤ The OpenMP runtime library called from.
- Select or dive on the stack parent token line to view the original routine's stack frame in the OpenMP master thread.
- On HP Alpha Tru64 UNIX, the system manager threads have a negative thread ID; since these threads are created by OpenMP, they are not part of the code you wrote. Consequently, you should never manipulate them.
- On HP Alpha Tru64 UNIX and on the Guide compilers, the OpenMP threads are implemented by the compiler as **pthreads**. On SGI IRIX, they are implemented as **sprocs**. TotalView shows the threads' logical and/or system thread ID, not the OpenMP thread number.
- SGI OpenMP uses the **SIGTERM** signal to terminate threads. Because TotalView stops a process when the process receives a **SIGTERM**, the OpenMP process doesn't terminate. If you want the OpenMP process to terminate instead of stop, set the default action for the **SIGTERM** signal to *Resend*.

Viewing OpenMP Private and Shared Variables

TotalView lets you view both OpenMP private and shared variables.

The compiler maintains OpenMP private variables in the outlined routine, and treats them like local variables. See "*Displaying Local Variables and Registers*" on page 244. In contrast, the compiler maintains OpenMP shared variables in the master thread's original routine stack frame. However, Guide compilers pass shared variables to the outlined routine as parameter references.

TotalView lets you display shared variables through a Process Window focused on the OpenMP master thread, or through one of the OpenMP worker threads.

To see these variables, you must:

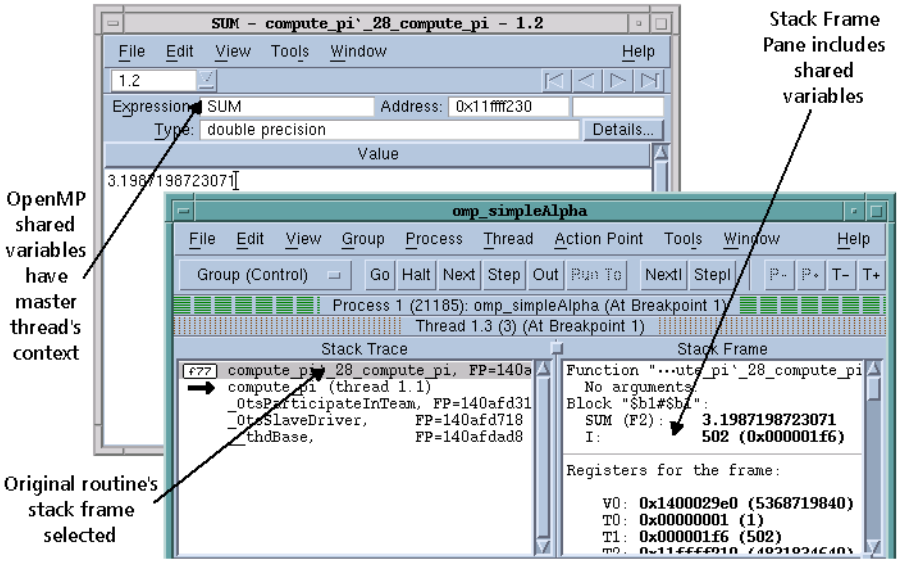
1. Select the outlined routine in the Stack Trace Pane, or select the original routine stack frame in the OpenMP master thread.

2 Dive on the variable name, or select the **View > Lookup Variable** command. When prompted, enter the variable name.

> CLI: dprint
>
> You will need to set your focus to the OpenMP master thread first.

TotalView opens a Variable Window that displays the value of the OpenMP shared variable, as shown in the following figure.

Figure 72: OpenMP Shared Variable



Shared variables reside in the OpenMP master thread's stack. When displaying shared variables in OpenMP worker threads, TotalView uses the stack context of the OpenMP master thread to find the shared variable. TotalView uses the OpenMP master thread's context when displaying the shared variable in a Variable Window.

GUI

You can also view OpenMP shared variables in the Stack Frame Pane by selecting either of the following:

- Original routine stack frame in the OpenMP master thread.
- Stack parent token line in the Stack Trace Pane of OpenMP worker threads, as shown in the following figure.

Viewing OpenMP THREADPRIVATE Common Blocks

The HP Alpha Tru64 UNIX OpenMP and SGI IRIX compilers implement OpenMP THREADPRIVATE common blocks by using the thread local storage system facility. This facility stores a variable declared in OpenMP THREADPRIVATE common blocks at different memory locations in each thread in an OpenMP process. This allows the variable to have different values in each thread. In contrast, the IBM and Guide compilers use the pthread key facility.

When you use SGI compilers, the compiler maps **THREADPRIVATE** variables to the same virtual address. However, they have different physical addresses.

To view a variable in an OpenMP **THREADPRIVATE** common block, or the OpenMP **THREADPRIVATE** common block:

1. In the Threads Pane of the Process Window, select the thread that contains the private copy of the variable or common block you want to view.
2. In the Stack Trace Pane of the Process Window, select the stack frame that lets you access the OpenMP **THREADPRIVATE** common block variable. You can select either the outlined routine or the original routine for an OpenMP master thread. You must, however, select the outlined routine for an OpenMP worker thread.
3. From the Process Window, dive on the variable name or common block name, or select the **View > Lookup Variable** command. When prompted, enter the name of the variable or common block. You may need to append an underscore character (_) after the common block name.

   CLI: **dprint**

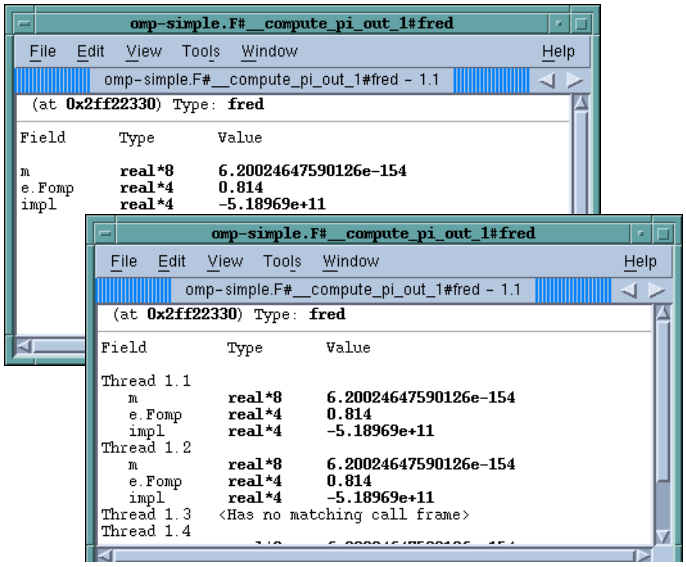
   TotalView opens a Variable Window that displays the value of the variable or common block for the selected thread.

   See "*Displaying Variables*" on page 238 for more information on displaying variables.

4. To view OpenMP **THREADPRIVATE** common blocks or variables across all threads, use the Variable Window's **View > Laminate Thread** command. See "*Displaying a Variable in all Processes or Threads*" on page 290.

The following figure shows Variable Windows displaying OpenMP **THREADPRIVATE** common blocks. Because the Variable Window has the same thread context as the Process Window from which it was created, the title bar patterns for the same thread match. TotalView displays the values of the common block across all threads in laminated views.

Figure 73: OpenMP THREADPRIVATE Common Block Variables

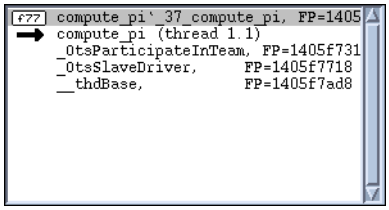


Viewing the OpenMP Stack Parent Token Line

TotalView inserts a special stack parent token line in the Stack Trace Pane of OpenMP worker threads when they are stopped in an outlined routine.

When you select or dive on the stack parent token line, the Process Window switches to the OpenMP master thread, allowing you to see the stack context of the OpenMP worker thread's routine.

Figure 74: OpenMP Stack Parent Token Line



This stack context includes the OpenMP shared variables.

Debugging Global Arrays Applications

The following paragraphs, which are copies from the Global Arrays home site (**http://www.emsl.pnl.gov/docs/global/ga.html**), describe the global arrays environment:

> The Global Arrays (GA) toolkit provides a shared memory style programming environment in the context of distributed array data structures (called "global arrays"). From the user perspective, a global array can be used as if it was stored in shared memory. All details of the data distribution, addressing, and data access are encapsulated in the global array objects. Information about the actual data distribution and locality can be easily obtained and taken advantage of whenever data locality is important. The primary target architectures for which GA was developed are massively-parallel distributed-memory and scalable shared-memory systems.
>
> GA divides logically shared data structures into "local" and "remote" portions. It recognizes variable data transfer costs required to access the data depending on the proximity attributes. A local portion of the shared memory is assumed to be faster to access and the remainder (remote portion) is considered slower to access. These differences do not hinder the ease-of-use since the library provides uniform access mechanisms for all the shared data regardless where the referenced data is located. In addition, any processes can access a local portion of the shared data directly/in-place like any other data in process local memory. Access to other portions of the shared data must be done through the GA library calls.
>
> GA was designed to complement rather than substitute for the message-passing model, and it allows the user to combine shared-memory and message-passing styles of programming in the same program. GA inherits an execution environment from a message-passing library (w.r.t. processes, file descriptors etc.) that started the parallel program.

TotalView supports Global Arrays on the Intel IA-64 platform. You debug a Global Arrays program in basically the same way that you debug any other multiprocess program. The one difference is that you will use the **Tools > Global Arrays** command to display information about your global data.

The global arrays environment has a few unique attributes. Using TotalView, you can:

- Display a list of a program's global arrays.
- Dive from this list of global variables to see the contents of a global array in C or Fortran format.
- Cast the data so that TotalView interprets data as a global array handle. This means that TotalView displays the information as a global array. Specifically, casting to **<GA>** forces the Fortran interpretation; casting to **<ga>** forces the C interpretation; and casting to **<Ga>** tells TotalView to use the language in the current context.

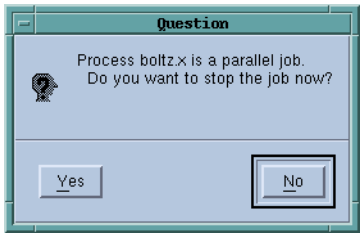
Within a Variable Window, the commands that operate on a local array, such as slicing, filtering, obtaining statistics, and visualization, also operate on global arrays.

The command you use to start TotalView depends on your operating system. For example, the following command starts TotalView on a program that is invoked using **prun** and which uses three processes:

```
totalview prun -a -N 3 boltz.x
```

Before your program starts parallel execution, TotalView asks if you want to stop the job.

Figure 75: Question Window for Global Arrays Program



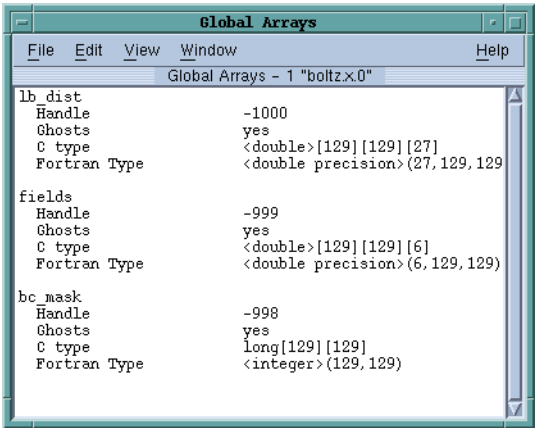
Choose **Yes** if you want to set breakpoints or inspect the program before it begins execution.

After your program hits a breakpoint, use the **Tools > Global Arrays** command to begin inspecting your program's global arrays.TotalView displays the window shown on the next page.

CLI:	dga

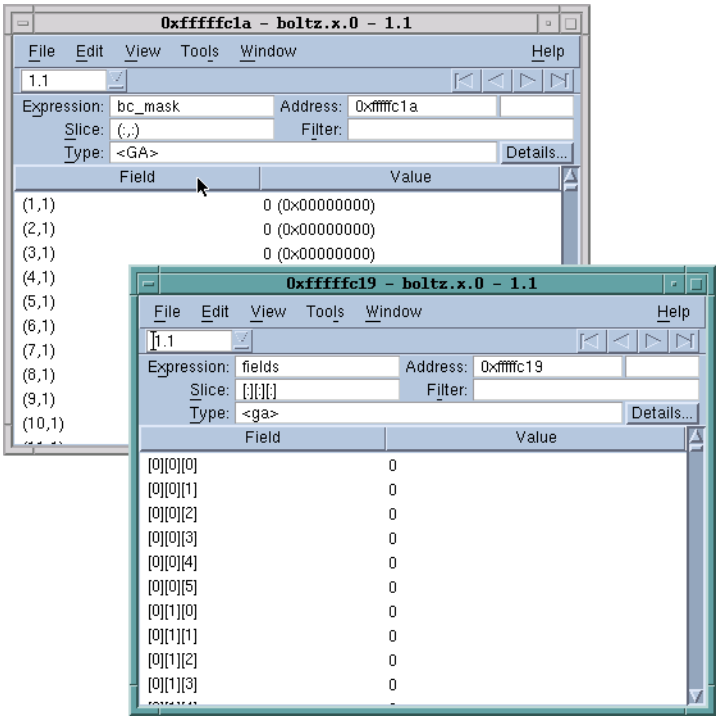
The arrays named in this window are displayed using their C and Fortran type names. Diving on the line that contains the type definition tells

Figure 76: Tools > Global Arrays Window



TotalView to display Variable Windows that contains information about that array.

Figure 77: Fortran and C Variable Windows

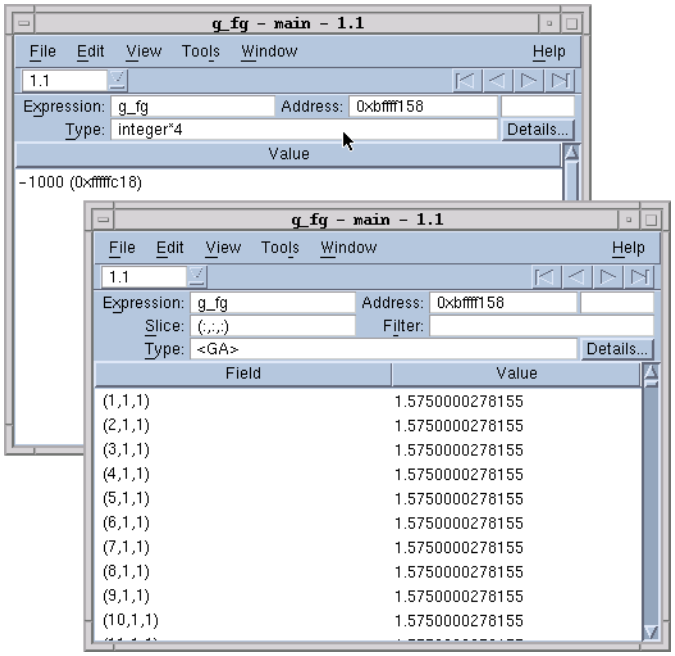


After TotalView displays this information, you can use other standard commands and operations on the array. For example, you can use the slice and filter operations and the commands that visualize, obtain statistics, and show the nodes from which the data was obtained.

If you inadvertently dive on a global array variable from the Process Window, TotalView does not know that it is a component of a global array. If, however, you do dive on the variable, you can cast the variable into a global array using either `<ga>` for a C Language cast or `<GA>` for a Fortran

cast. The following figure shows a Variable Window before and after the data was cast.

Figure 78: Casting in Fortran



Debugging PVM (Parallel Virtual Machine) and DPVM Applications

You can debug applications that use the Parallel Virtual Machine (PVM) library or the HP Alpha Tru64 UNIX Parallel Virtual Machine (DPVM) library with TotalView on some platforms. TotalView supports ORNL PVM Version 3.4.4 on all platforms and DPVM Version 1.9 or later on the HP Alpha platform.

See the TotalView Platforms document for the most up-to-date information regarding your PVM or DPVM software.

For tips on debugging parallel applications, see "*Debugging Parallel Applications Tips*" on page 113.

Topics in this section are:

- "*Supporting Multiple Sessions*" on page 104
- "*Setting Up ORNL PVM Debugging*" on page 104
- "*Starting an ORNL PVM Session*" on page 104
- "*Starting a DPVM Session*" on page 105
- "*Automatically Acquiring PVM/DPVM Processes*" on page 106
- "*Attaching to PVM/DPVM Tasks*" on page 107

Supporting Multiple Sessions

When you debug a PVM or DPVM application, TotalView becomes a PVM tasker. This lets it establish a debugging context for your session. You can do the following:

- You can run TotalView PVM or DPVM debugging session for a user and for an architecture; that is, different users can't interfere with each other on the same computer or same computer architecture.

  One user can start TotalView to debug the same PVM or DPVM application on different computer architectures. However, a single user can't have multiple instances of TotalView debugging the same PVM or DPVM session on a single computer architecture.

  For example, if you start a PVM session on Sun 5 and HP Alpha computers. You must start two TotalView sessions: one on the Sun 5 computer to debug the Sun 5 portion of the PVM session, and one on the HP Alpha computer to debug the HP Alpha portion of the PVM session. These two TotalView sessions are separate and don't interfere with one another.
- In one TotalView session, you can run either a PVM application or a DPVM application, but not both. However, if you run TotalView on an HP Alpha, you can have two TotalView sessions: one debugging PVM and one debugging DPVM.

Setting Up ORNL PVM Debugging

To enable PVM, create a symbolic link from the PVM **bin** directory (which is usually **$HOME/pvm3/bin/$PVM_ARCH/tvdsvr**) to the TotalView Debugger Server (**tvdsvr**). With this link in place, TotalView invokes **pvm_spawn()** to spawn the **tvdsvr** tasks.

For example, if **tvdsvr** is installed in the **/opt/totalview/bin** directory, enter the following command:

```
ln -s /opt/totalview/bin/tvdsvr \
            $HOME/pvm3/bin/$PVM_ARCH/tvdsvr
```

If the symbolic link doesn't exist, TotalView can't spawn **tvdsvr**. If TotalView can't spawn **tvdsvr**, it displays the following error:

```
Error spawning TotalView Debugger Server: No such file
```

Starting an ORNL PVM Session

Start the ORNL PVM daemon process before you start TotalView. See the ORNL PVM documentation for information about the PVM daemon process and console program. The procedure for starting an ORNL PVM application is as follows:

1. Use the **pvm** command to start a PVM console session—this command starts the PVM daemon.

   If PVM isn't running when you start TotalView (with PVM support enabled), TotalView exits with the following message:

   ```
   Fatal error: Error enrolling as PVM task:
   pvm error
   ```
2. If your application uses groups, start the **pvmgs** process before starting TotalView.

   PVM groups are unrelated to TotalView process groups. For information about TotalView process groups, see "*Examining Groups*" on page 188.

3 You can use the **–pvm** command-line option to the **totalview** command. As an alternative, you can set the **TV::pvm** variable in a startup file.

The command-line options override the CLI variable. For more information, see "*TotalView Command Syntax*" in the *TotalView Reference Guide*.

4 Set the TotalView directory search path to include the PVM directories.

This directory list must include those needed to find both executable and source files. The directories you use can vary, but should always contain the current directory and your home directory.

You can set the directory search path using either the **EXECUTABLE_PATH** variable or the **File > Search Path** command. See "*Setting Search Paths*" on page 50 for more information.

For example, to debug the PVM examples, you can place the following directories in your search path:

```
.
$HOME
$PVM_ROOT/xep
$PVM_ROOT/xep/$PVM_ARCH
$PVM_ROOT/src
$PVM_ROOT/src/$PVM_ARCH
$PVM_ROOT/bin/$PVM_ARCH
$PVM_ROOT/examples
$PVM_ROOT/examples/$PVM_ARCH
$PVM_ROOT/gexamples
$PVM_ROOT/gexamples/$PVM_ARCH
```

5 Verify that the action taken by TotalView for the **SIGTERM** signal is appropriate. (You can examine the current action by using the Process Window **File > Signals** command. See "*Handling Signals*" on page 48 for more information.)

PVM uses the **SIGTERM** signal to terminate processes. Because TotalView stops a process when the process receives a **SIGTERM**, the process is not terminated. If you want the PVM process to terminate, set the action for the **SIGTERM** signal to **Resend**.

TotalView will automatically acquire your application's PVM processes. For more information, see "*Automatically Acquiring* PVM/DPVM *Processes*" on page 106.

Starting a DPVM Session

Starting a DPVM debugging session is similar to starting any other TotalView debugging session. The only additional requirement is that you must start the DPVM daemon before you start TotalView. See the DPVM documentation for information about the DPVM daemon and its console program. The procedure for starting an DPVM application is as follows

1 Use the **dpvm** command to start a DPVM console session; starting the session also starts the DPVM daemon.

If DPVM isn't running when you start TotalView (with DPVM support enabled), TotalView displays the following error message before it exits:

```
Fatal error: Error enrolling as DPVM task: dpvm error
```

2 Enable DPVM support either by using the **TV::dpvm** CLI variable or by using the **–dpvm** command-line option to the **totalview** command.

The command-line options override the **TV:dpvm** command variable. For more information on the **totalview** command, see "*TotalView Command Syntax*" in the *TotalView Reference Guide*.

3 Verify that the default action taken by TotalView for the **SIGTERM** signal is appropriate. (You can examine the default actions with the Process Window **File** > **Signals** command in TotalView. See "*Handling Signals*" on page 48 for more information.)

DPVM uses the **SIGTERM** signal to terminate processes. Because TotalView stops a process when the process receives a **SIGTERM**, the process is not terminated. If you want the DPVM process to terminate, set the action for the **SIGTERM** signal to **Resend**.

If you enable PVM support using the **TV::pvm** variable and you need to use DPVM, you must use both **–no_pvm** and **–dpvm** command-line options when you start TotalView. Similarly, when enabling DPVM support us the **TV::dpvm** variable, you must use the **–no_dpvm** and **–pvm** command-line options.

You cannot use CLI *variables to start both* PVM *and* DPVM.

Automatically Acquiring PVM/ DPVM Processes

When you start TotalView as part of a PVM or DPVM debugging session, it takes the following actions:

- TotalView makes sure that no other PVM or DPVM taskers are running. If TotalView finds a tasker on a host that it is debugging, it displays the following message and then exits:

  ```
  Fatal error: A PVM tasker is already running
  on host 'host'
  ```

- TotalView finds all the hosts in the PVM or DPVM configuration. Using the **pvm_spawn()** call, TotalView starts a TotalView Debugger Server (**tvdsvr**) on each remote host that has the same architecture type as the host TotalView is running on. It tells you it has started a debugger server by displaying the following message:

  ```
  Spawning TotalView Debugger Server onto PVM
  host 'host'
  ```

If you add a host with a compatible computer architecture to your PVM or DPVM debugging session after you start TotalView, TotalView automatically starts a debugger server on that host.

After all debugger servers are running, TotalView intercepts every PVM or DPVM task created with the **pvm_spawn()** call on hosts that are part of the debugging session. If a PVM or DPVM task is created on a host with a different computer architecture, TotalView ignores that task.

When TotalView receives a PVM or DPVM tasker event, the following actions occur:

1. TotalView reads the symbol table of the spawned executable.
2. If a saved breakpoint file for the executable exists and you have enabled automatic loading of breakpoints, TotalView loads breakpoints for the process.
3. TotalView asks if you want to stop the process before it enters the **main()** routine.

   If you answer **Yes**, TotalView stops the process before it enters **main()** (that is, before it executes any user code). This allows you to set breakpoints in the spawned process before any user code executes. On most computers, TotalView stops a process in the **start()** routine of the **crt0.o** module if it is statically linked. If the process is dynamically linked, TotalView stops it just after it finishes running the dynamic linker. Because the Process Window displays assembler instructions, you need to use the **View > Lookup Function** command to display the source code for **main()**.

   CLI: **dlist** *function-name*

   For more information on this command, see "*Finding the Source Code for Functions*" on page 181.

Attaching to PVM/ DPVM Tasks

You can attach to a PVM or DPVM task if the following are true:

- The computer architecture on which the task is running is the same as the computer architecture upon which TotalView is running.
- The task must be created. (This is indicated when flag 4 is set in the PVM Tasks and Configuration Window.)
- The task must not be a PVM tasker. If flag 400 is clear in the PVM Tasks and Configuration Window, the process is a tasker.
- The executable name must be known. If the executable name is listed as a dash (**–**), TotalView cannot determine the name of the executable. (This can occur if a task was not created with the **pvm_spawn()** call.)

To attach to a PVM or DPVM task:

1. Select the **Tools > PVM Tasks** command from the Root Window.

   TotalView responds the PVM Tasks Window. (This is shown on the next page.)

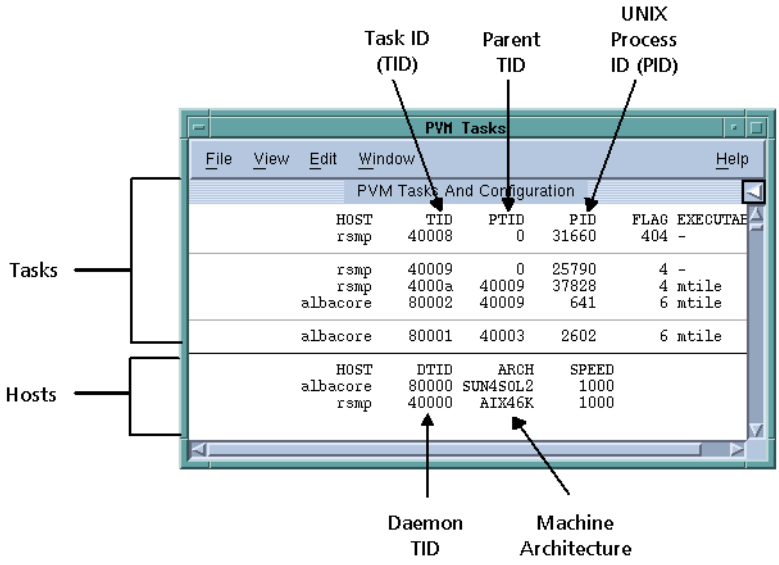
   This window displays current information about PVM tasks and hosts—TotalView automatically updates this information as it receives events from PVM.

   Since PVM doesn't always generate an event that allows TotalView to update this window, use the **Window > Update** command to ensure that you are seeing the most current information.

   For example, you can attach to the tasks named **xep** and **mtile** in the preceding figure because flag 4 is set. In contrast, you can't attach to the **–** (dash) executables and **tvdsvr**, because flag 400 is set.
2. Dive on a task entry that meets the criteria for attaching to tasks.

   TotalView attaches to the task.

Figure 79: PVM Tasks and Configuration Window



3 If the task to which you attached has related tasks that can be debugged, TotalView asks if you want to attach to these related tasks. If you answer **Yes**, TotalView attaches to them. If you answer **No**, it only attaches to the task you dove on.

After attaching to a task, TotalView looks for attached tasks that are related to this task; if there are related tasks, TotalView places them in the same control group. If TotalView is already attached to a task you dove on, it simply opens and raises the Process Window for the task.

About Reserved Message Tags

TotalView uses PVM message tags in the range 0xDEB0 through 0xDEBF to communicate with PVM daemons and the TotalView Debugger Server. Avoid sending messages that use these reserved tags.

Cleaning Up Processes

The **pvmgs** process registers its task ID in the PVM database. If the **pvmgs** process terminates, the **pvm_joingroup()** routine hangs because PVM won't clean up the database. If this happens, you must manually terminate the program and then restart the PVM daemon.

TotalView attempts to clean up the **tvdsvr** processes that also act as taskers. If some of these processes do not terminate, you must manually terminate them.

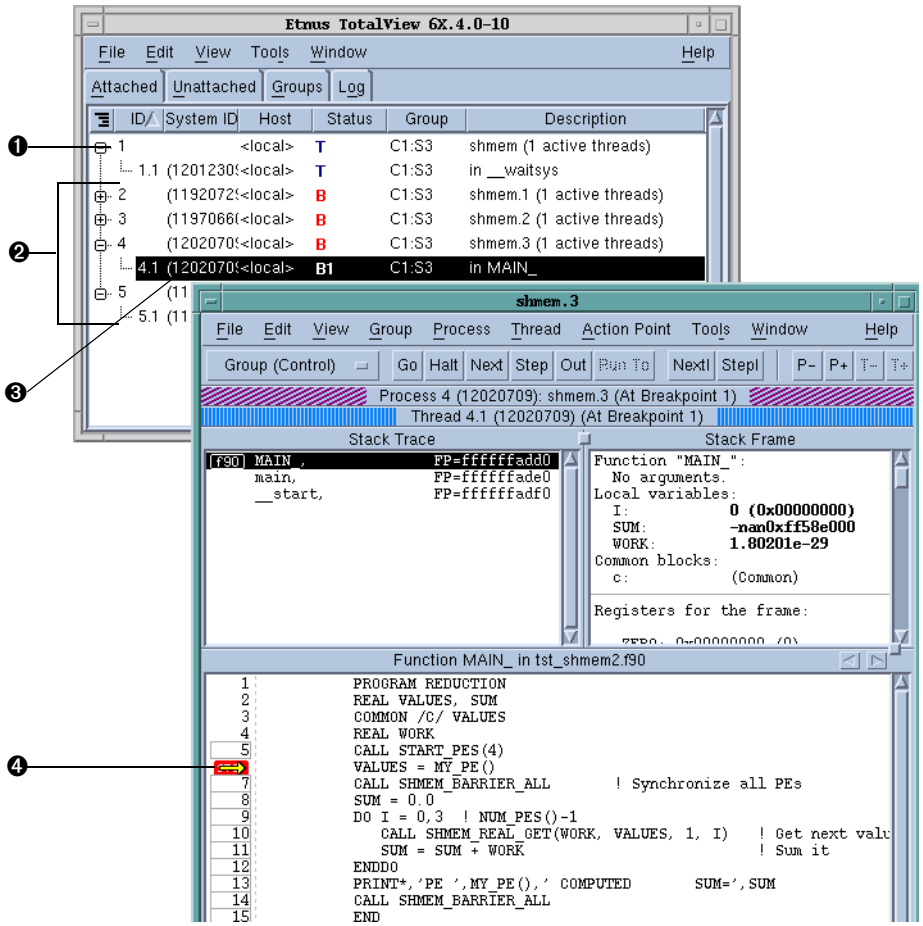
Debugging Shared Memory (SHMEM) Code

TotalView supports the SGI IRIX logically shared, distributed memory access (SHMEM) library.

To debug a SHMEM program:

1. Link the SHMEM program with the dbfork library. See "*Linking with the dbfork Library*" in the "*Compilers and Platforms*" chapter of the *TotalView Reference Guide*.
2. Start TotalView on your program. (See Chapter 3, "*Setting Up a Debugging Session*," on page 35.)
3. Set at least one breakpoint after the call to the **start_pes()** SHMEM routine.

Figure 80: SHMEM Sample Session



❶ SHMEM starter process
❷ SHMEM worker processes
❸ Select a worker process in the Root Window
❹ Set a breakpoint *after* the call to start_pes()

You cannot single-step over the call to **start_pes()**.

The call to **start_pes()** creates new worker processes that return from the **start_pes()** call and execute the remainder of your program. The original process never returns from **start_pes()**, but instead stays in that routine, waiting for the worker processes it created to terminate.

Debugging UPC Programs

TotalView lets you debug UPC programs that were compiled using the HP Compaq Alpha UPC 2.0 and the Intrepid (SGI gcc UPC) compilers. This section only discusses the UPC-specific features of TotalView. It is not an introduction to the UPC Language. For an introduction to the UPC language, go to **http://www.gwu.edu/~upc**.

When debugging UPC *code, TotalView requires help from a* UPC *assistant library that your compiler vendor provides. You need to include the location of this library in your* **LD_LIBRARY_PATH** *environment variable. Etnus also provides assistants that you can use. You can find these assistants at* **http://www.etnus.com/Products/TotalView/developers/index.html**.

Topics in this section are:

- *"Invoking TotalView"* on page 110
- *"Viewing Shared Objects"* on page 110
- *"Displaying Pointer to Shared Variables"* on page 112

Invoking TotalView

The way in which you invoke TotalView on a UPC program is straight-forward. However, this procedure depends on the computer upon which the program is executing:

- When running on an SGI system using the gcc UPC compiler, invoke TotalView on your UPC program in the same way as you would invoke it on most other programs; for example:
  `totalview prog_upc -a prog_upc_args`
- When running on HP Compaq SC computers, debug UPC code in the same way that you would debug other kinds of parallel code; for example:
  `totalview prun -a -n node_count prog_upc prog_upc_args`

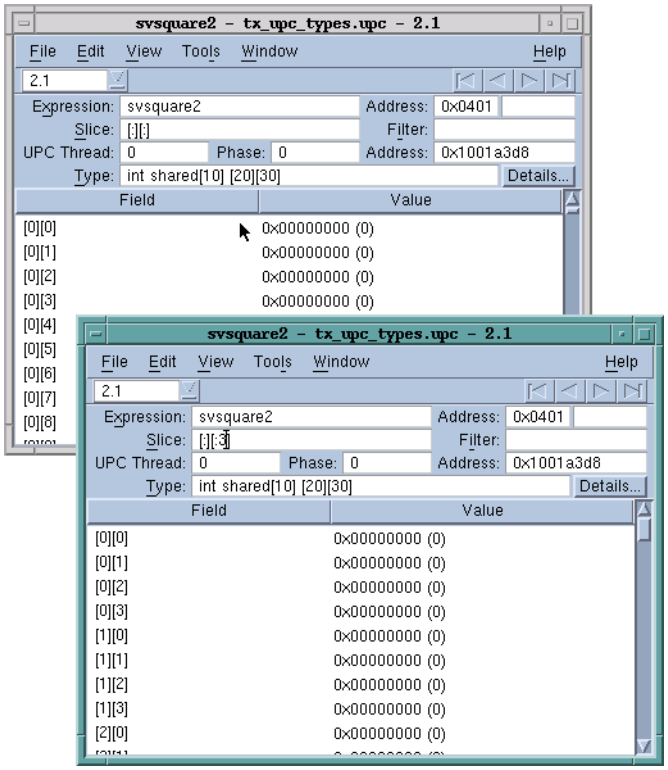
Viewing Shared Objects

Totalview displays UPC shared objects, and fetches data from the UPC thread with which it has an affinity. For example, TotalView always fetches shared scalar variables from thread 0.

The upper-left screen in the following figure displays elements of a large shared array. You can manipulate and examine shared arrays the same as any other array. For example, you can slice, filter, obtain statistical information, and so on. (For more information on displaying array data, see Chap-

ter 13, "*Examining Arrays*," on page 279.) The lower-right screen shows a ten-element slice of this array.

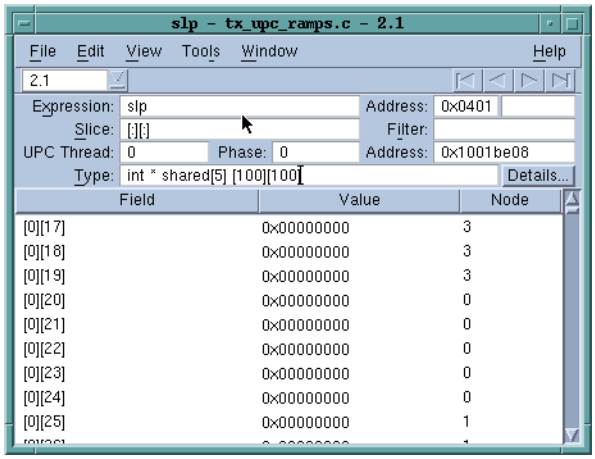
Figure 81: A *Sliced* UPC *Array*



In this figure, TotalView displays the value of a pointer-to-shared variable whose target is the array in the **Shared Address** area. As usual, the address in the process appears in the top left of the display.

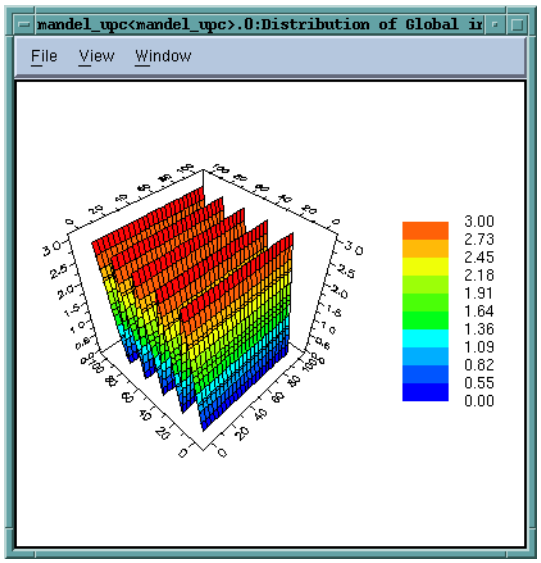
Since the array is shared, it has an additional property: the element's affinity. You can display this information if you right-click your mouse on the header and tell TotalView to display Nodes..

Figure 82: UPC *Variable Window Showing Nodes*



You can also use the **Tools > Visualize Distribution** command to visualize this array. For more information on visualization, see "*Visualizing Array Data*" on page 139.

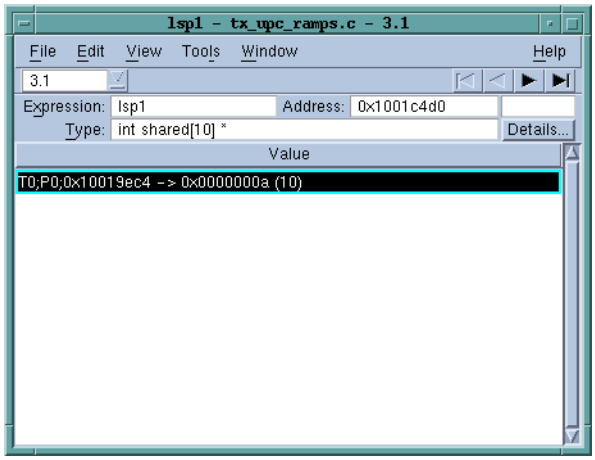
Figure 83: Visualizing a UPC *Variable*



Displaying Pointer to Shared Variables

TotalView understands pointer-to-shared data and displays the components of the data, as well as the target of the pointer to shared variables. For example, the following figure shows this data being displayed:

Figure 84: A Pointer to a Shared Variable



In this figure, notice the following:

- Because the **Type** field shows the full type name, TotalView is telling you that this is a pointer to a shared **int** with a block size of 10.
- In this figure, TotalView also displays the **upc_threadof ("T0")**, the **upc_phaseof ("P0")**, and the **upc_addrfield (0x0x10010ec4)** components of this variable.

In the same way that TotalView normally shows the target of a pointer variable, it also shows the target of a UPC pointer variable. When dereferencing a UPC pointer, TotalView fetches the target of the pointer from the UPC thread with which the pointer has affinity.

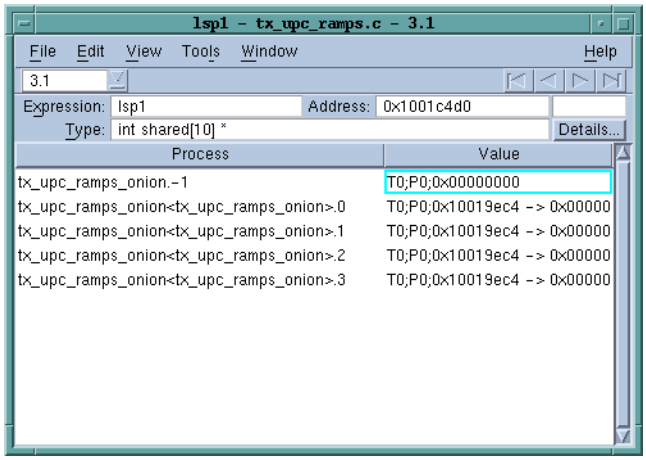
You can update the pointer by selecting the pointer value and editing the thread, phase, or address values. If the phase is corrupt, you'll see something like the following in the **Value** field:

```
Value: T0;P6;0x3ffc0003b00 <Bad phase [max 4]> ->
                    0xc0003c80 (-1073726336)
```

In this example, the pointer is invalid because the phase is outside the legal range. TotalView displays a similar message if the thread is invalid.

Since the pointer itself is not shared, you can use the **Tools** > **Laminate** commands to display the value from each of the UPC threads.

Figure 85: Pointer to a Shared Variable



Debugging Parallel Applications Tips

This section contains information about debugging parallel programs:

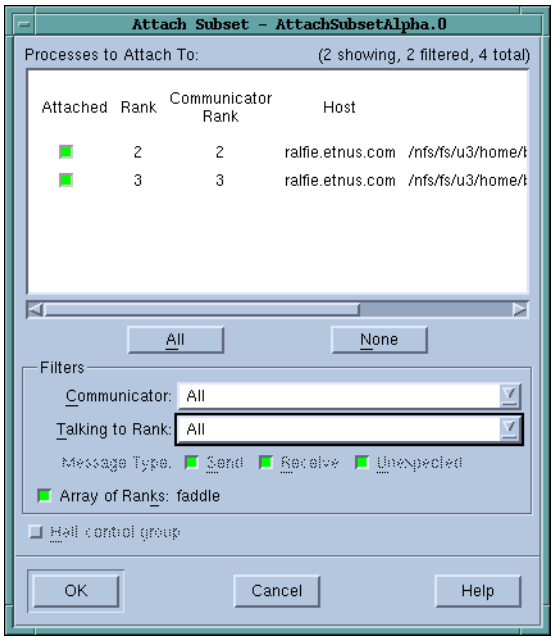
- "*Attaching to Processes*" on page 113
- "*Parallel Debugging Tips*" on page 116
- "MPICH *Debugging Tips*" on page 118
- "IBM PE *Debugging Tips*" on page 118

Attaching to Processes

In a typical multiprocess job, you're interested in what's occurring in some of your processes and not as much interested in others. By default, TotalView tries to attach to all of the processes that your program starts. If there are a lot of processes, there can be considerable overhead involved in opening and communicating with the jobs.

You can minimize this overhead by using the **Group** > **Attach Subset** command, which displays the dialog box shown in the figure on the next page.

Figure 86: Group > Attach Subset Dialog Box



Selecting boxes on the left side of the list tells TotalView which processes it should attach to. Although your program will launch all of these processes, TotalView only attaches to the processes that you have selected.

The controls under the **All** and the **None** buttons let you limit which processes TotalView automatically attaches to, as follows:

- The **Communicator** control specifies that the processes must be involved with the communicators that you select. For example, if something goes wrong that involves a communicator, selecting it from the list tells TotalView to only attach to the processes that use that communicator.
- The **Talking to Rank** control further limits the processes to those that you name here. Most of the entries in this list are just the process numbers. In most cases, you would select **All** or **MPI_ANY_SOURCE**.
- The three checkboxes in the **Message Type** area add yet another qualifier. Checking a box tells TotalView to only display communicators that are involved with a **Send**, **Receive**, or **Unexpected** message.

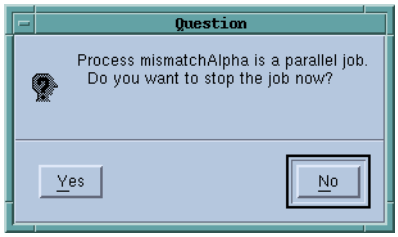
After you find the problem, you can detach from these nodes by selecting **None**. In most cases, use the **All** button to set all the check boxes, then clear the ones that you're not interested in.

Many applications place values that indicate the rank in a variable so that the program can refer to them as they are needed. If you do this, you can display the variable in a Variable Window and then select the **Tools > Attach Subset (Array of Ranks)** command to display this dialog box.

You can use the Group > Attach Subset command at any time, but you would probably use it immediately before TotalView launches processes. Unless you have set preferences otherwise, TotalView stops and asks if you want it to stop your processes. When selected, the **Halt control group**

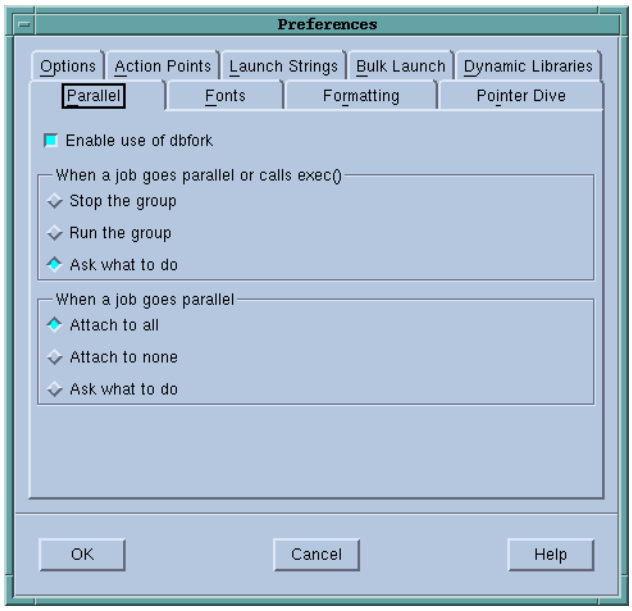
check box also tells TotalView to stop a process just before it begins executing.

Figure 87: Stop Before Going Parallel Question Box



The commands on the Parallel Page in the **File > Preferences** Dialog Box let you control what TotalView does when your program goes parallel.

Figure 88: File > Preferences: Parallel Page



The radio button in the **When a job goes parallel or calls exec()** area let TotalView:

- **Stop the group**: Stop the control group immediately after the processes are created.
- **Run the group**: Allow all newly created processes in the control group to run freely.
- **Ask what to do**: Ask what should occur. If you select this option, TotalView asks if it should start the created processes.

CLI: dset TV::parallel_stop

The radio buttons in the **When a job goes parallel** area let TotalView:

- **Attach to all**: Automatically attach to all processes when they begin executing.
- **Attach to none**: Does not attach to any created process when it begins executing.
- **Ask what to do**: Asks what should occur. If you select this option, TotalView opens the same dialog box that is displayed when you select **Group > Attach Subset**. TotalView then attaches to the processes that you have selected. This dialog box isn't displayed when you set the preference. Instead, it controls what happenes when your program creates parallel processes.

CLI: dset TV::parallel_attach

Parallel Debugging Tips

The following tips are useful for debugging most parallel programs:

- **Setting Breakpoint behavior**

  When you're debugging message-passing and other multiprocess programs, it is usually easier to understand the program's behavior if you change the default stopping action of breakpoints and barrier breakpoints. By default, when one process in a multiprocess program hits a breakpoint, TotalView stops all the other processes.

  To change the default stopping action of breakpoints and barrier breakpoints, you can set TotalView preferences. The online Help contains information on these preference. These preferences tell TotalView whether to continue to run when a process or thread hits the breakpoint.

  These options only affect the default behavior. You can choose a behavior for a breakpoint by setting the breakpoint properties in the **File > Preferences** Action Points Page. See "*Setting Breakpoints for Multiple Processes*" on page 300.

- **Synchronizing Processes**

  TotalView has two features that make it easier to get all of the processes in a multiprocess program synchronized and executing a line of code. Process barrier breakpoints and the process hold/release features work together to help you control the execution of your processes. See "*Setting Barrier Points*" on page 303.

  The Process Window **Group > Run To** command is a special stepping command. It lets you run a group of processes to a selected source line or instruction. See "*Stepping (Part I)*" on page 207.

- **Using group commands**

  Group commands are often more useful than process commands.

  It is often more useful to use the **Group > Go** command to restart the whole application instead of the **Process > Go** command.

  CLI: dfocus g dgo
  Abbreviation: G

You would then use the **Group** > **Halt** command instead of **Process** > **Halt** to stop execution.

CLI:	dfocus g dhalt Abbreviation: H

The group-level single-stepping commands such as **Group** > **Step** and **Group** > **Next** let you single-step a group of processes in a parallel. See "*Stepping* (*Part* I)" on page 207.

CLI:	dfocus g dstep Abbreviation: S dfocus g dnext Abbreviation: N

- **Stepping at Process-level**

  If you use a process-level single-stepping command in a multiprocess program, TotalView may appear to hang (it continuously displays the watch cursor). If you single-step a process over a statement that can't complete without allowing another process to run, and that process is stopped, the stepping process appears to hang. This can occur, for example, when you try to single-step a process over a communication operation that cannot complete without the participation of another process. When this happens, you can abort the single-step operation by selecting **Cancel** in the **Waiting for Command to Complete** Window that TotalView displays. As an alternative, consider using a group-level single-step command.

CLI:	Type Ctrl+C

Etnus receives many bug reports about processes being hung. In almost all cases, the reason is that one process is waiting for another. Using the **Group** *debugging commands almost always solves this problem.*

- **Determining which processes and threads are executing**

  The TotalView Root Window helps you determine where various processes and threads are executing. When you select a line of code in the Process Window, the Root Window Attached Page updates to show which processes and threads are executing that line.

- **Viewing variable values**

  You can view (laminate) the value of a variable that is replicated across multiple processes or multiple threads in a single Variable Window. See "*Displaying a Variable in all Processes or Threads*" on page 290.

- **Restarting from within TotalView**

  You can restart a parallel program at any time. If your program runs past the point you want to examine, you can kill the program by selecting the **Group** > **Delete** command. This command kills the master process and all the slave processes. Restarting the master process (for example, **mpirun** or **poe**) recreates all of the slave processes. Start up is faster when you do

this because TotalView doesn't need to reread the symbol tables or restart its **tvdsvr** processes, since they are already running.

CLI: dfocus g dkill

MPICH Debugging Tips

The following debugging tips apply only to MPICH:

- **Passing options to mpirun**

  You can pass options to TotalView using the MPICH **mpirun** command.

  To pass options to TotalView when running **mpirun**, you can use the **TOTALVIEW** environment variable. For example, you can cause **mpirun** to invoke TotalView with the **–no_stop_all** option, as in the following C shell example:

  ```
  setenv TOTALVIEW "totalview -no_stop_all"
  ```

- **Using ch_p4**

  If you start remote processes with MPICH/**ch_p4**, you may need to change the way TotalView starts its servers.

  By default, TotalView uses **rsh** to start its remote server processes. This is the same behavior as **ch_p4** uses. If you configure **ch_p4** to use a different start-up mechanism from another process, you probably also need to change the way that TotalView starts the servers.

  For more information about **tvdsvr** and **rsh**, see "*Setting Single-Process Server Launch Options*" on page 64. For more information about **rsh**, see "*Using the Single-Process Server Launch Command*" on page 68.

IBM PE Debugging Tips

The following debugging tips apply only to IBM MPI (PE):

- **Avoid unwanted timeouts**

  Timeouts can occur if you place breakpoints that stop other processes too soon after calling **MPI_Init()** or **MPL_Init()**. If you create "stop all" breakpoints, the first process that gets to the breakpoint stops all the other parallel processes that have not yet arrived at the breakpoint. This can cause a timeout.

  To turn the option off, select the Process Window **Action Point > Properties** command while the line with the stop symbol is selected. After the **Properties** Dialog Box appears, select the **Process** button in the **When Hit, Stop** area, and also select the **Plant in share group** button.

  CLI: dbarrier *location* –stop_when_hit process

- **Control the poe process**

  Even though the **poe** process continues under TotalView control, do not attempt to start, stop, or otherwise interact with it. Your parallel tasks require that **poe** continues to run. For this reason, if **poe** is stopped, TotalView automatically continues it when you continue any parallel task.

- **Avoid slow processes due to node saturation**

  If you try to debug a PE program in which more than three parallel tasks run on a single node, the parallel tasks on each node can run noticeably slower than they would run if you were not debugging them.

In general, the number of processes running on a node should be the same as the number of processors in the node.

This becomes more noticeable as the number of tasks increases, and, in some cases, the parallel tasks does not progress. This is because PE uses the SIGALRM signal to implement communications operations, and AIX requires that debuggers must intercept all signals. As the number of parallel tasks on a node increases, TotalView becomes saturated and can't keep up with the SIGALRM signals being sent, thus slowing the tasks.

Part III: Using the GUI

The two chapters in this part of the users guide contains information about using the TotalView GUI.

Chapter 6: Using TotalView Windows
: Describes using the mouse and the fundamental TotalView windows.

Chapter 7: Visualizing Programs and Data
: Some of the debugger's commands and tools are only useful if you're using the GUI. Here you will find information on the Call Tree and Visualizer.

Using TotalView Windows

6

This chapter introduces you to the most important TotalView windows and the mechanics of using the GUI. The topics in this chapter are as follows:

- "*Using the Mouse Buttons*" on page 123
- "*Using the Root Window*" on page 124
- "*Using the Process Window*" on page 129
- "*Viewing the Assembler Version of Your Code*" on page 131
- "*Diving into Objects*" on page 132
- "*Resizing and Positioning Windows and Dialog Boxes*" on page 134
- "*Editing Text*" on page 135
- "*Saving the Contents of Windows*" on page 136

Using the Mouse Buttons

TotalView uses the buttons on your three-button mouse as follows:

Button	Action	Purpose	How to Use It
Left	Select	Selects or edits object. Scrolls in windows and panes.	Move the cursor over the object and click the button.
Middle	Paste	Writes information previously copied or cut into the clipboard.	Move the cursor to where you will be inserting the information and click the button. Not all windows support pasting.
	Dive	Displays more information or replaces window contents.	Move the cursor over an object, then click the middle-mouse button.
Right	Context menu	Displays a menu with commonly used commands.	Move the cursor over an object and click the button. Most windows and panes have context menus; dialog boxes do not have context menus.

In most cases, a single-click selects what's under the cursor and a double-click dives on the object. However, if the field is editable, TotalView goes into its edit mode, in which you can alter the selected item's value.

In some places such as the Stack Trace Pane, selecting a line tells TotalView to perform an action. In this pane, TotalView dives on the selected routine. (In this case, *diving* means that TotalView finds the selected routine and shows it in the Source Pane.)

In the line number area of the Source Pane, a left mouse click sets a breakpoint at that line. TotalView shows you that it has set a breakpoint by displaying a STOP icon instead of a line number.

Selecting the STOP icon a second time deletes the breakpoint. If you change any of the breakpoint's properties or if you've created an eval point (indicated by an EVAL icon), selecting the icon disables it. For more information on breakpoints and eval points, see Chapter 14, "*Setting Action Points*," on page 293.

Using the Root Window

The Root Window appears when you start TotalView. If you do not enter a program name when you start TotalView, it is the only window that appears. If you type a program name immediately after the **totalview** command, TotalView also opens a Process Window that contains the program's source code.

The *Root Window* contains the following four tabbed pages, which are described in the following sections:

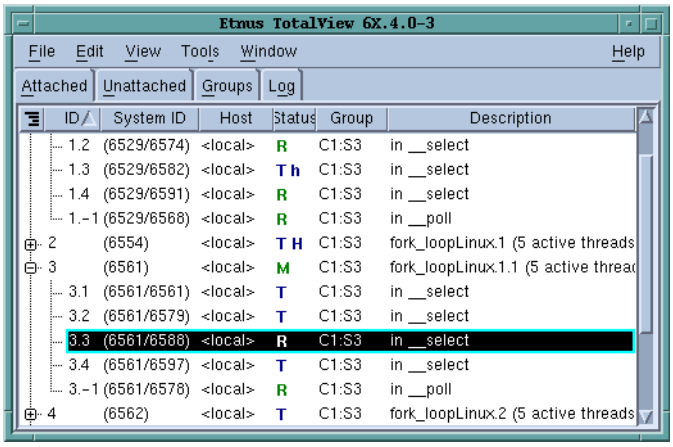
- "*Using the Attached Page*" on page 124
- "*Using the Unattached Page*" on page 127
- "*Using the Groups Page*" on page 128
- "*Using the Log Page*" on page 128

Using the Attached Page

The Attached Page displays a list of all the processes and threads being debugged. Initially—that is, before your program begins executing—the Root Window just contains the name of the program being debugged. As your program creates processes and threads, TotalView adds them to this list. Associated with each is a name, location (if a remote process), process ID, status, and a list of executing threads for each process. It also shows the thread ID, status, and the routine being executed in each thread.

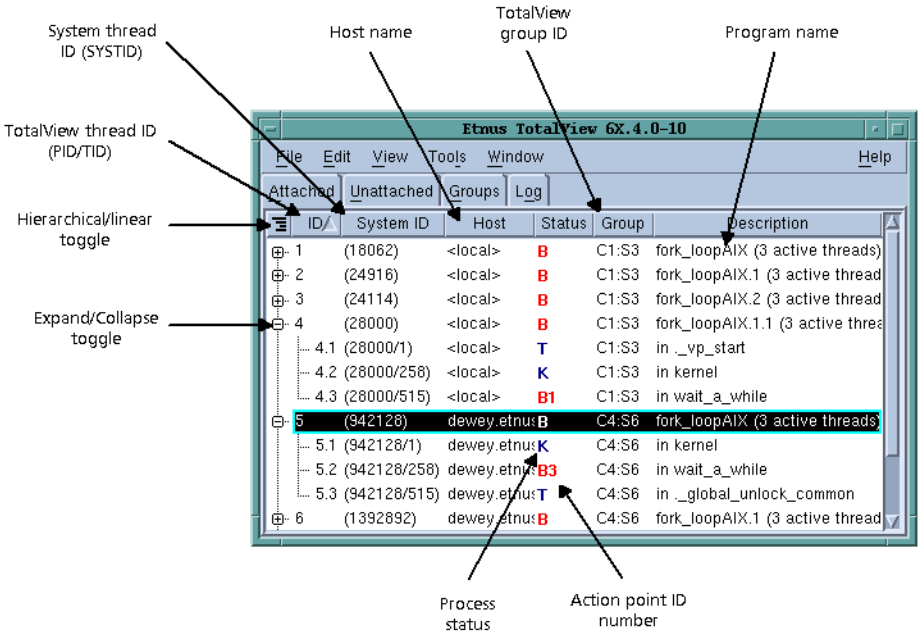
The following figure shows the Attached Page for an executing multi-threaded multiprocess program.

Figure 89: Root Window Attached Page



When debugging a remote process, TotalView displays an abbreviated version of the host name on which the process is running in brackets ([ ]). The full host name appears in brackets in the title bar of the Process Window. In the following figure, the process is running on the computer **dewey.etnus.com**. This name is abbreviated in the Root Window. This figure also describes the contents of the columns in this window.

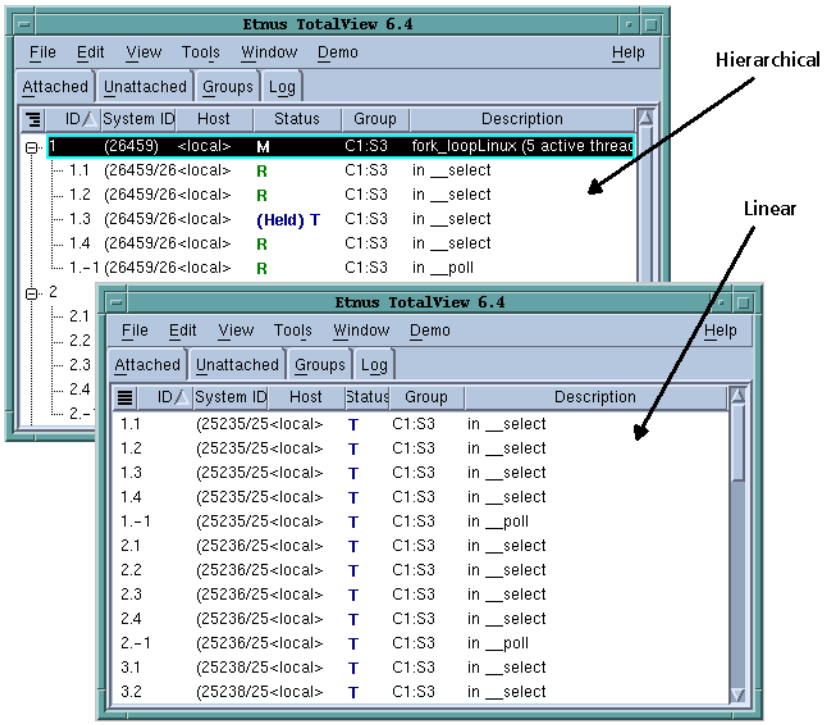
Figure 90: Root Window Showing Two Host Computers



When you dive on a line in this window, TotalView displays the source for that process or thread in a Process Window.

TotalView can display process and thread data linearly and hierarchically.

Figure 91: Two Views of the Root Window



Selecting the hierarchy toggle button ( ) changes the view from linear to hierarchical view. When data is being displayed hierarchically, you can perform the following additional operations:

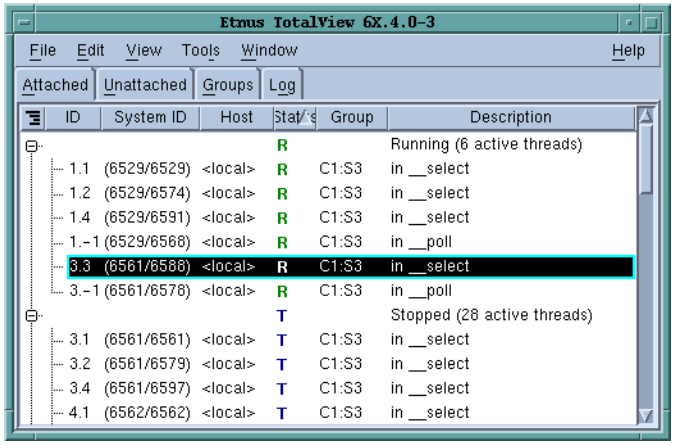
- Selectively display information using the + or – indicators. The **View > Expand All** and **View > Compress All** commands let you open and close all of this window's hierarchies.
- Sort a column by clicking on a column header.

The hierarchical view lets you group similar information. For example, if you sort the information by clicking the **Status** header, TotalView groups all attached processes by their status. This lets you see, for example, which threads are held, at a breakpoint, and so on. When information is aggregated (that is grouped) like this, you can also display information selectively. This is shown in the figure on the next page.

TotalView displays all of your program's processes and threads. You can change this using the following commands:

- **View > Display Manager Threads**: When multiprocess and multithreaded programs run, the operating system often creates threads whose sole function is to manage your program's processes. Usually, you are not interested in these threads. This command lets you remove these threads from the display. Most users don't want to see these threads.
- **View > Display Exited Threads**: Tracking when processes stop and start executing in a multiprocess, multithreaded enviroment can be challenging. Selecting this command tells TotalView to display threads after

Figure 92: Sorted and Aggregated Root Window

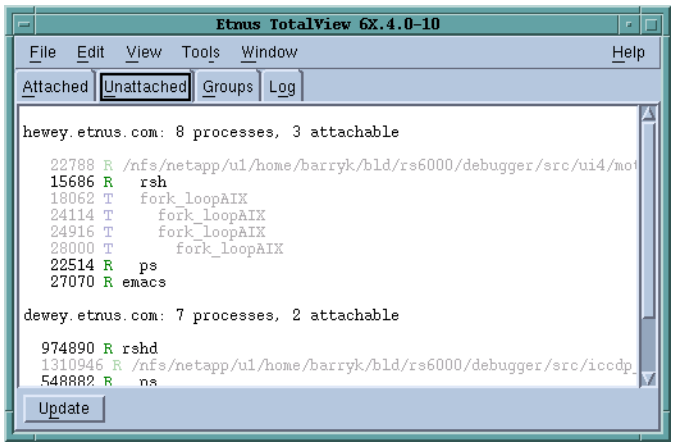


they've exited. While this clutters your display with information about threads that are no longer executing, it can sometimes be helpful in trying to track down some problems. You probably don't want to see these threads in the listing. However, you can tell TotalView to show them at anytime. That is, TotalView remembers them so that toggling this command shows this information.

Using the Unattached Page

The Unattached Page displays processes over which you have control. If you can't attach to one of these processes, TotalView displays it in grey. The following figure shows the Unattached Page.

Figure 93: Root Window Unattached Page

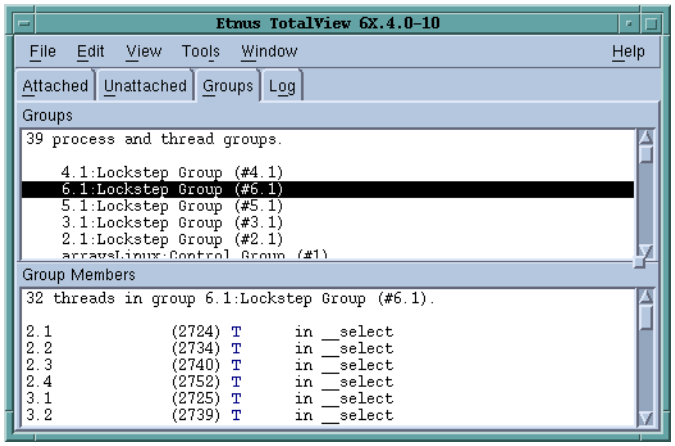


Diving on processes in this pane tells TotalView to attach to them.

Using the Groups Page

The Groups Page lists the groups used by your program. The top pane in the following figure lists all of your program's groups. This list includes all the groups that TotalView creates and all the groups that you create. When you select a group in the top pane, TotalView displays the group's members in the bottom pane.

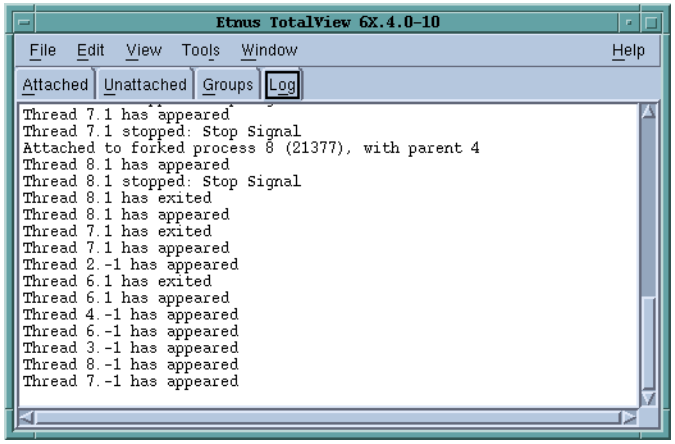
Figure 94: Root Window Groups Page



Using the Log Page

The Log Page contains a journal of the debugging actions. This information is sometimes useful when analyzing the behavior of misbehaving multiprocess, multithreaded programs.

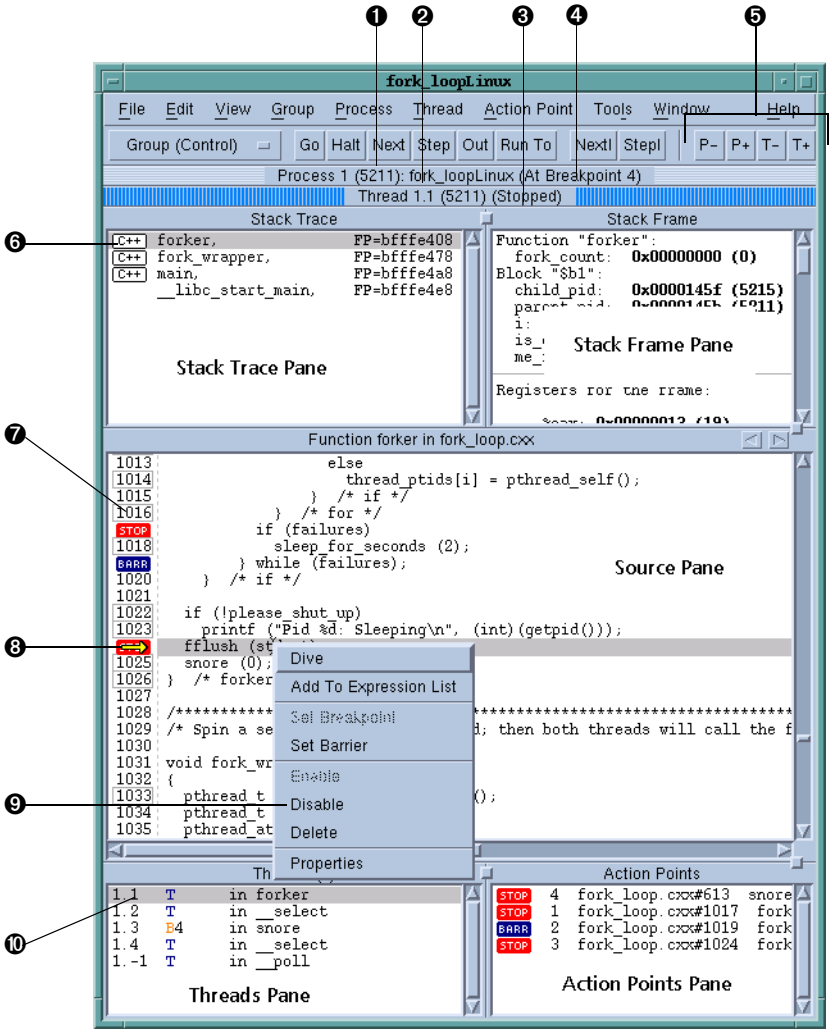
Figure 95: Root Window Log Page



Using the Process Window

The *Process Window* contains the code for the process or thread that you're debugging, as well as other related information. This window contains five *panes* of information. The large scrolling list in the middle of the Process Window is the Source Pane. (The contents of these panes are discussed later in this section.)

Figure 96: A Process Window



❶ Process ID (PID)
❷ Thread ID (TID)
❸ Thread status
❹ Process status
❺ Navigation controls
❻ Language of routine
❼ Line number area
❽ Current program counter
❾ Context menu
❿ Selected thread

As you examine the Process Window, notice the following:

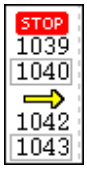
- The thread ID shown in the Root Window and in the process's Threads Pane is the logical thread ID (TID) assigned by TotalView and the system-assigned thread ID (SYSTID). On systems such as HP Alpha Tru64 UNIX, where the TID and SYSTID values are the same, TotalView displays only the TID value.

  In other windows, TotalView uses the value *pid.tid* to identify a process's threads.

  The Threads Pane shows the list of threads that currently exist in a process. When you select a different thread in this list, TotalView updates the Stack Trace Pane, Stack Frame Pane, and Source Pane to show the information for that thread. When you dive on a different thread in the thread list, TotalView finds or opens a new window that displays information for that thread.
- The Stack Trace Pane shows the call stack of routines that the selected thread is executing. You can move up and down the call stack by clicking on the routine's name (stack frame). When you select a different stack frame, TotalView updates the Stack Frame and Source Panes to show the information about the routine you just selected.
- The Stack Frame Pane displays all of a routine's parameters, its local variables, and the 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. This means that the information that they display is not up-to-date while the thread is running.
- The left margin of the *Source Pane* displays line numbers and action point icons. You can place a breakpoint at any line whose line number is contained within a box. The box indicates that executable code was created by the source code.

  When you place a breakpoint on a line, TotalView places a STOP icon over the line number. An arrow over the line number shows the current location of the program counter (PC) in the selected stack frame.

Figure 97: Line Numbers with Stop Icon and PC Arrow



Each thread has its own unique program counter (PC). When you stop a multiprocess or multithreaded program, the routine displayed in the Stack Trace Pane for a thread depends on the thread's PC. Because threads execute asynchronously, threads are stopped at different places. (When your thread hits a breakpoint, the TotalView default is to stop all the other threads in the process as well.)

- The *Action Points Pane* shows the list of breakpoints, eval points, and watchpoints for the process.

Viewing the Assembler Version of Your Code

You can display your program in source or assembler. You can use the following commands:

Source code (Default)
Select the **View** > **Source As** > **Source** command.

Assembler code Select the **View** > **Source As** > **Assembler** command.

Both Source and assembler
Select the **View** > **Source As** > **Both** command.

The Source Pane divides into two parts. The left pane contains the program's source code and the right pane contains the assembler version of this code. You can set breakpoints in either of these panes. Setting an action point at the first instruction after a source statement is the same as setting it at that source statement.

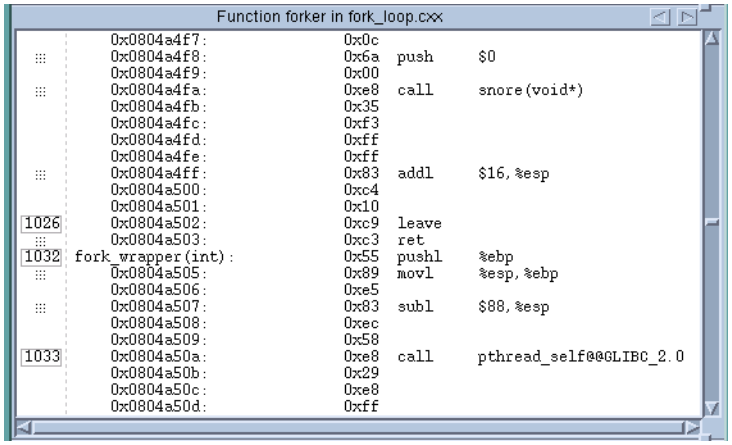
The commands in the following table tell TotalView to display your assembler code by using symbolic or absolute addresses:

Command	Display
View > Assembler > By Address	Absolute addresses for locations and references (default)
View > Assembler > Symbolically	Symbolic addresses (function names and offsets) for locations and references

You can also display assembler instructions in a Variable Window. For more information, see "Displaying Machine Instructions" on page 248.

The following three figures illustrate the different ways TotalView can display assembler code. In the following figure, the second column (the one to the right of the line numbers) shows the absolute address location. The fourth column shows references using absolute addresses.

Figure 98: Address Only (Absolute Addresses)



The following figure shows information symbolically. The second column shows locations using functions and offsets.

Figure 99: Assembly Only (Symbolic Addresses)

```
Function forker in fork_loop.cxx
      forker(long)+0x5b3:     0x0c
      forker(long)+0x5b4:     0x6a  push    $0
      forker(long)+0x5b5:     0x00
      forker(long)+0x5b6:     0xe8  call    snore(void*)
      forker(long)+0x5b7:     0x35
      forker(long)+0x5b8:     0xf3
      forker(long)+0x5b9:     0xff
      forker(long)+0x5ba:     0xff
      forker(long)+0x5bb:     0x83  addl    $16,%esp
      forker(long)+0x5bc:     0xc4
      forker(long)+0x5bd:     0x10
 1026 forker(long)+0x5be:     0xc9  leave
      forker(long)+0x5bf:     0xc3  ret
 1032 fork_wrapper(int):      0x55  pushl   %ebp
      fork_wrapper(int)+0x01: 0x89  movl    %esp,%ebp
      fork_wrapper(int)+0x02: 0xe5
      fork_wrapper(int)+0x03: 0x83  subl    $88,%esp
      fork_wrapper(int)+0x04: 0xec
      fork_wrapper(int)+0x05: 0x58
 1033 fork_wrapper(int)+0x06: 0xe8  call    pthread_self@@GLIBC_2.0
      fork_wrapper(int)+0x07: 0x29
      fork_wrapper(int)+0x08: 0xe8
      fork_wrapper(int)+0x09: 0xff
```

The final assembler figure shows the split Source Pane, with one side showing the program's source code and the other showing the assembler version. In this example, the assembler is shown symbolically. How it is shown depends on whether you've selected **View > Assembler > By Address** or **View > Assembler > Symbolically**.

Figure 100: Both Source and Assembler (Symbolic Addresses)

```
Function forker in fork_loop.cxx
1015                 }  /* if *
1016               }  /* for */
STOP            if (failures)
1018              sleep_for_seco
BRKP          } while (failures)
1020      }  /* if */
1021
1022    if (!please_shut_up)
1023      printf ("Pid %d: Sleep
        fflush (stdout);
1025    snore (0);
1026  }  /* forker */
1027
1028  /*************************
1029  /* Spin a second thread, i
1030
1031  void fork_wrapper (int for
1032  {
1033    pthread_t my_ptid = pthr
1034    pthread_t new_tid;
1035    pthread_attr_t attr;
1036    int whoops;
1037    int local_fork_count;

      forker(long)+0x595:
      forker(long)+0x596:
      forker(long)+0x597:
      forker(long)+0x598:  call    prir
      forker(long)+0x599:
      forker(long)+0x59a:
      forker(long)+0x59b:
      forker(long)+0x59c:
      forker(long)+0x59d:  addl    $16,
      forker(long)+0x59e:
      forker(long)+0x59f:
      forker(long)+0x5a0:  subl    $12,
      forker(long)+0x5a1:
      forker(long)+0x5a2:
      forker(long)+0x5a3:  pushl   stdc
      forker(long)+0x5a4:
      forker(long)+0x5a5:
      forker(long)+0x5a6:
      forker(long)+0x5a7:
      forker(long)+0x5a8:
      forker(long)+0x5a9:  call    fflu
      forker(long)+0x5aa:
      forker(long)+0x5ab:
```

Diving into Objects

Diving, which is clicking your middle mouse button on something in a TotalView window, is one of the debugger's more distinguishing features.

In some cases, single-clicking preforms a dive. For example, single-clicking on a function name in the Stack Trace Pane tells TotalView to dive into the function. In other cases, double-clicking does the same thing.

Diving on processes and threads in the Root Window is the quickest way to display a Process Window that contains information about what you're diving on. The procedure is simple: dive on a process or thread and TotalView takes care of the rest. Another example is diving on variables in the Process Window, which tells TotalView to display information about the variable in a Variable Window.

The following table describes typical diving operations:

Items you dive on:	Information Displayed:
Process or thread	When you dive on a thread in the Root Window, TotalView finds or opens a Process Window for that process. If it doesn't find a matching window, TotalView replaces the contents of an existing window and shows you the selected process.
Variable	The contents of the variable appear in a separate Variable Window.
Routine in the Stack Trace Pane	The stack frame and source code for the routine appear in a Process 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 *nested* dive.
Pointer	TotalView dereferences the pointer and shows the result in a separate Variable Window. Given the nature of pointers, you may need to cast the result into the logical data type.
Subroutine	The source code for the routine replaces the current contents of the Source Pane. When this occurs TotalView places a right angle bracket (>) in the process's title. Every time it dives, it adds another angle bracket. See the figure that follows this table A routine must be compiled with source-line information (usually, with the **–g** option) for you to dive into it and see source code. If the subroutine wasn't compiled with this information, TotalView displays the routine's assembler code.
Variable Window	TotalView replaces the contents of the Variable Window with information about the variable or element you're diving on.
Expression List Window	TotalView displays information about the variable in a separate Variable Window.

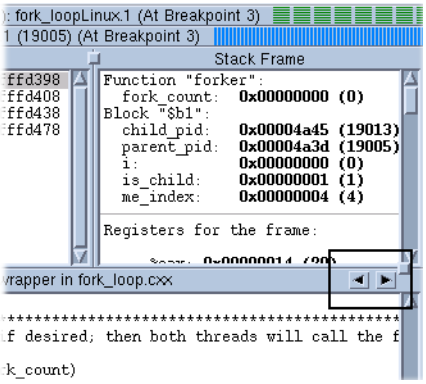
Figure 101: Nested Dive

TotalView tries to reuse windows whenever possible. For example, if you dive on a variable and that variable is already being displayed in a window, TotalView pops the window to the top of the display. If you want the information to appear in a separate window, use the **View > Dive in New Window** command.

Diving on a process or a thread might not create a new window if TotalView determines that it can reuse a Process Window. If you really want to see information in two windows, use the Process Window **Window > Duplicate** *command.*

When you dive into functions in the Process Window, or when you are chasing pointers or following structure elements in the Variable Window, you can move back and forth between your selections by using the *forward* and *backward* buttons. The boxed area of the following figure shows the location of these two controls.

Figure 102: Backward and Forward Buttons



For additional information about displaying variable contents, see "*Diving in Variable Windows*" on page 249.

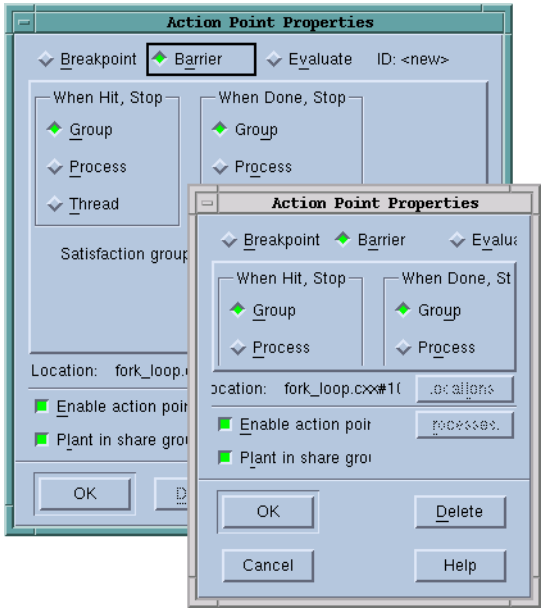
You can also use the following additional windowing commands:

- **Window > Duplicate**: (Variable and Expression List Windows) Creates a duplicate copy of the current Variable Window.
- **File > Close**: Closes an open window.
- **File > Close Relatives**: Closes windows that are related to the current window. The current window isn't closed.
- **File > Close Similar**: Closes the current window and all windows similar to it.

Resizing and Positioning Windows and Dialog Boxes

You can resize most TotalView windows and dialog boxes. While TotalView tries to do the right thing, you can push things to the point where shrinking doesn't work very well. The figure on the next page shows a before-and-after look in which a dialog box was made too small.

Figure 103: Resizing (and Its Consequences)



Many programmers like to have their windows always appear in the same position in each session. The following two commands can help:

- **Window > Memorize**: Tells TotalView to remember the position of the current window. The next time you bring up this window, it'll be in this position.
- **Window > Memorize All**: Tells TotalView to remember the positions of most windows. The next time you bring up any of the windows displayed when you used this command, it will be in the same position.

Most modern window managers such as KDE or Gnome do an excellent job managing window position. If you are using an older window manager such as **twm** or **mwm**, you may want to select the **Force window positions (disables window manager placement modes)** check box option located on the Options Page of the **File > Preferences** Dialog Box. This tells TotalView to manage a window's position and size. If it isn't selected, TotalView only manages a window's size.

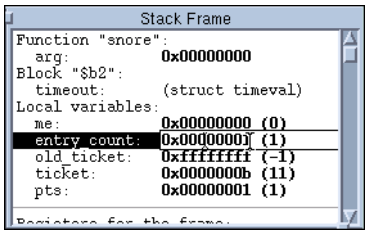
Editing Text

The TotalView field editor lets you change the values of fields in windows or change text fields in dialog boxes.

To edit text:

1. Click the left mouse button to select the text you want to change. If you can edit the selected text, TotalView will display an editing cursor. (This is shown in the figure on the next page.)
2. Edit the text and press Return.

Figure 104: Editing Cursor



Like other Motif-based applications, you can use your mouse to copy and paste text in TotalView and to other X Windows applications by using your mouse buttons.

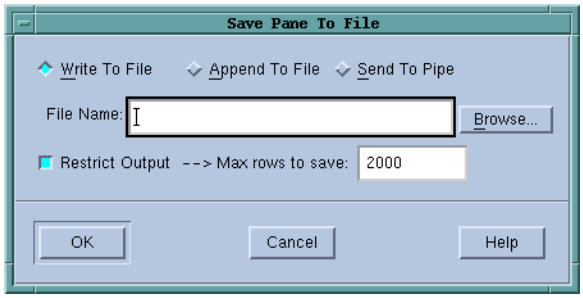
You can also manipulate text by using **Edit > Copy**, **Edit > Cut**, **Edit > Paste**, and **Edit > Delete**. If you haven't yet pressed the Return key to confirm your change, you can use the **Edit > Undo** command to restore information.

Usually TotalView dives when you click your middle-mouse button on something. However, if TotalView is displaying an editing cursor, clicking your middle-mouse button pastes text.

Saving the Contents of Windows

You can write an ASCII equivalent to most pages and panes by using the **File > Save Pane** command. This command also lets you pipe data to UNIX shell commands.

Figure 105: File > Save Pane Dialog Box



If the window or pane contains a lot of data, you can use the **Restrict Output** option to limit how much information TotalView writes or sends. For example, you might not want to write a 100 x 100 x 10,000 array to disk. If this option is checked (the default), TotalView only sends the indicated number of lines. You can, of course, change the amount indicated here.

When piping information, TotalView sends what you've typed to **/bin/sh**. This means that you can enter a series of shell commands. For example, the following is a command that ignores the top five lines of output, compares the current ASCII text to an existing file, and writes the differences to another file:

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

Visualizing Programs and Data

7

TotalView provides a set of tools that let you visualize how your program is performing and its variables. This chapter describes:

- "*Displaying Your Program's Call Tree*" on page 137
- "*Visualizing Array Data*" on page 139

Other visualization tools are described in the following sections:

- "*Using the* P/T *Set Browser*" on page 229
- "*Displaying the Message Queue Graph Window*" on page 90

If you need to connect another graphic system to TotalView, see "Adapting a Third Party Visualizer" on our web site at **http://www.etnus.com/Support/docs/rel6/html/User_Guide/AdaptingaThirdPartyVisualizer**.

Displaying Your Program's Call Tree

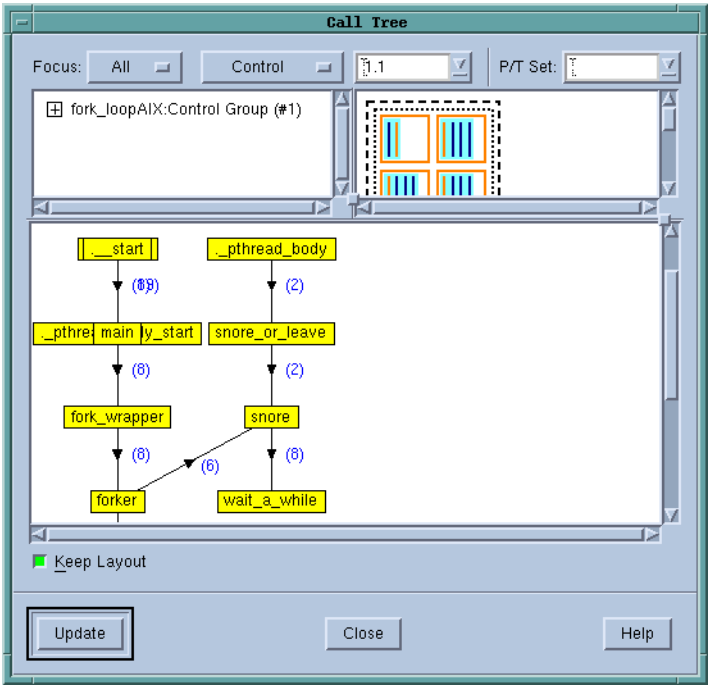
Debugging is an art, not a science. Debugging often means having the intuition to make guesses about what a program is doing and where to look for what is causing the problem. Locating a problem is often 90% or more of the effort. The call tree can help you understand what your program is doing so that you can begin to understand how your program is executing.

Choose the **Tools > Call Tree** command in the Process Window to tell TotalView to display a Call Tree Window. (A sample call tree is shown on the next page.)

The call tree is a diagram that shows all the currently active routines. These routines are linked by arrows indicating that one routine is called by another. The call tree is a *dynamic* call tree in that it displays the call tree at the time when TotalView creates it. The **Update** button tells TotalView to recreate this display.

Information on using the P/T *Set Controls in the top portion of this window is in Chapter* 11, "*Using Groups, Processes, and Threads,*" *on page* 205.

Figure 106: Tools > Call Tree Dialog Box



You can tell TotalView to display a call tree for the processes and threads specified with the controls at the top of this window. If you don't touch these controls, TotalView displays a call tree for the group defined in the toolbar of your Process Window. If TotalView is displaying the call tree for a multiprocess or multithreaded program, numbers next to the arrows indicate how many times a routine is on the call stack.

As you begin to understand your program, you will see that it has a rhythm and a dynamic that is reflected in this diagram. As you examine and understand this structure, you will sometimes see things that don't look right—which is a subjective response to how your program is operating. These places are often where you want to begin looking for problems.

Looking at the call tree can also tell you where bottlenecks are occurring. For example, if one routine is used by many other routines, and that routine controls a shared resource, this thread might be negatively affecting performance. For example, in the preceding figure, the **snore** routine might be a bottleneck. Creating routine names that say what the routine does helps. For example, if you see many links pointing to a routine named **snore**, you've probably designed things so that your routines will wait there. In this case; seeing lots of links wouldn't represent a problem.

Visualizing Array Data

The TotalView Visualizer creates graphic images of your program's array data.

The Visualizer isn't available on Linux Alpha and 32-bit SGI Irix. It's available on all other platforms.

Topics in this section are:

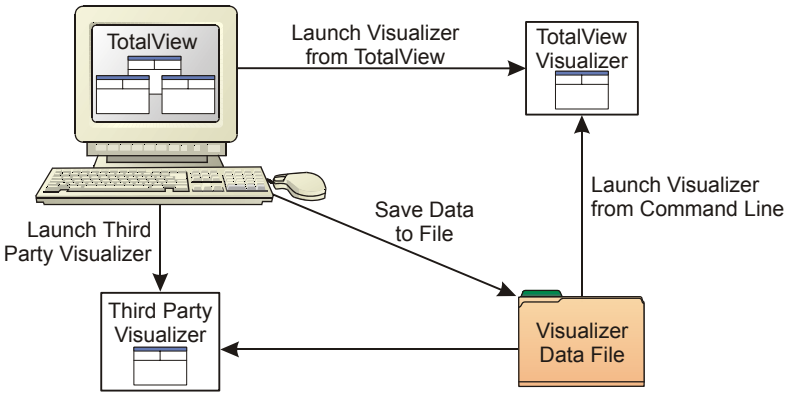
- "*How the Visualizer Works*" on page 139
- "*Configuring TotalView to Launch the Visualizer*" on page 140
- "*Visualizing Data Manually*" on page 143
- "*Visualizing Data Programmatically*" on page 143
- "*Using the Visualizer*" on page 145
- "*Using the Graph Window*" on page 147
- "*Using the Surface Window*" on page 149
- "*Launching the Visualizer from the Command Line*" on page 152

How the Visualizer Works

The Visualizer is a stand-alone program that is integrated with TotalView. This relationship gives considerable flexibility; for example:

- If you launch the Visualizer from within TotalView, you can visualize your program's data as you are debugging your program.
- You can save the data that would be sent to the Visualizer, and then invoke the Visualizer from the command line and have it read this previously written data.

Figure 107: TotalView Visualizer Relationships



- Because TotalView is sending a data stream to the Visualizer, you can even replace our Visualizer with any tool that can read this data.

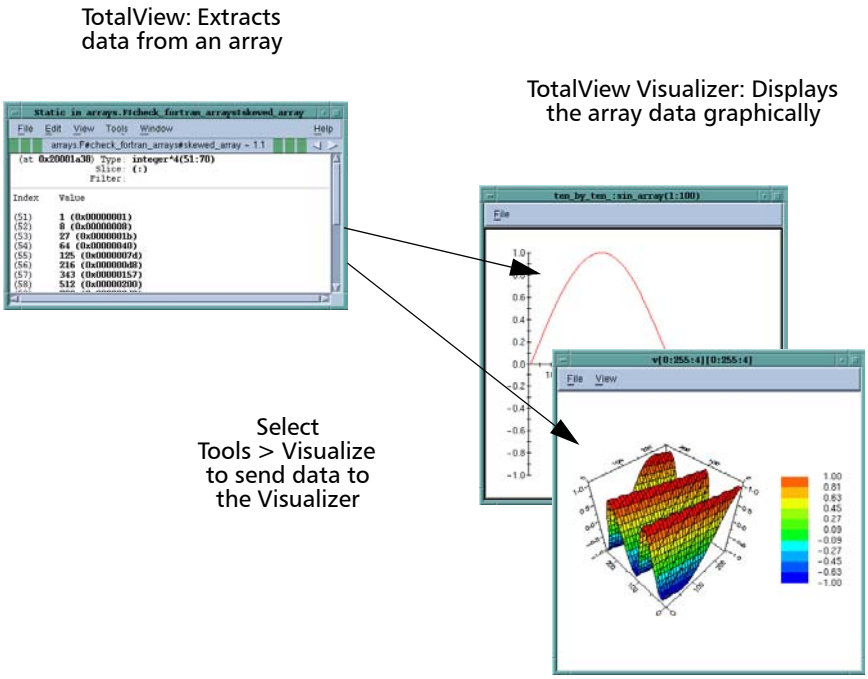
The online Help contains information on adapting a third-party visualizer so that it can be used with TotalView. This is also on our web site at **http://www.etnus.com/Support/docs/rel6/html/User_Guide/AdaptingaThirdPartyVisualizer.**

Visualizing your program's data is a two step process, as follows:

1 You select the data that you want visualized.
2 You tell the Visualizer how to display this data.

TotalView marshals the program's data and pipes it to the Visualizer. The Visualizer reads this data and displays it for analysis.

Figure 108: TotalView Visualizer Connection



Configuring TotalView to Launch the Visualizer

TotalView launches the Visualizer when you select the **Tools > Visualize** command from the Variable Window. It also launches it if or when you use a **$visualize** function in an eval point and the **Tools > Evaluate** Dialog Box.

TotalView lets you set a preference that disables visualization. This lets you turn off visualization when your program executes code that contains eval points, without having to individually disable all the eval points.

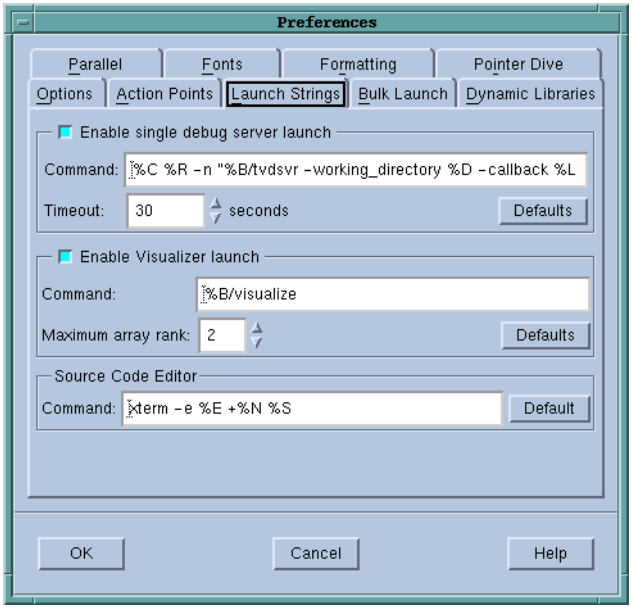
To change the Visualizer launch options interactively, select **File > Preferences**, and then select the Launch Strings Tab. (These options are shown in the figure on the next page.)

Using the commands on this page, you can do the following:

- Customize the command that TotalView uses to start a visualizer by entering the visualizer's start up command in the **Command** edit box.
- Change the autolaunching option. If you want to disable visualization, clear the **Enable Visualizer launch** check box.
- Change the maximum permissible rank. Edit the value in the **Maximum array rank** field to save the data exported from the debugger or display it in a different visualizer. A rank's value can range from **1** to **16**.

  Setting the maximum permissible rank to either 1 or 2 (the default is 2) ensures that the TotalView Visualizer can use your data—the Visualizer

Figure 109: File > Preferences Launch Strings Page



displays only two dimensions of data. This limit doesn't apply to data saved in files or to third-party visualizers that can display more than two dimensions of data.

- Clicking the **Defaults** button returns all values to their default values. This reverts options to their default values even if you have used X resources to change them.

If you disable visualization while the Visualizer is running, TotalView closes its connection to the Visualizer. If you reenable visualization, TotalView launches a new Visualizer process the next time you visualize something.

Setting the Visualizer Launch Command

You can change the shell command that TotalView uses to launch the Visualizer by editing the Visualizer launch command. (In most cases, the only reason you'd do this is if you're having path problems or you're running a different visualizer.) You can also change what's entered here so that you can view this information at another time; for example:

```
cat > your_file
```

Later, you can visualize this information using either of the following commands:

```
visualize -persist < your_file
visualize -file your_file
```

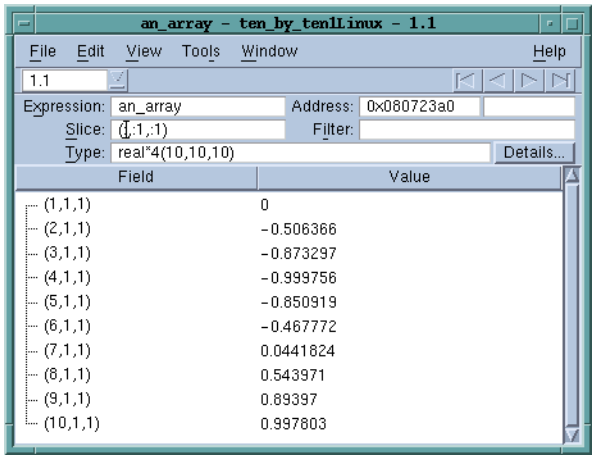
You can preset the Visualizer launch options by setting X resources. These resources are described on our web site. For more information, go to **http://www.etnus.com/Support/docs/**.

Viewing Data Types in the Visualizer

The data selected for visualization is called a *dataset*. TotalView tags each dataset with a numeric identifier This identifier lets the Visualizer know whether it is seeing a new dataset or an update to an existing dataset. TotalView treats stack variables at different recursion levels or call paths as different datasets.

TotalView can visualize one- and two-dimensional arrays of character, integer, or floating-point data. If an array has more than two dimensions, you can visualize part of it using an array slice that creates a subarray that has fewer dimensions. The following figure shows a three-dimensional variable sliced into two dimensions by selecting a single index in the middle dimension.

Figure 110: A Three-Dimensional Array Sliced into Two Dimensions



Viewing Data

Different datasets can require different views to display their data. For example, a graph is more suitable for displaying one-dimensional datasets or two-dimensional datasets if one of the dimensions has a small extent. However, a surface view is better for displaying a two-dimensional dataset.

When TotalView launches the Visualizer, one of the following actions occurs:

- If a Data Window is currently displaying the dataset, the Visualizer raises it to the top of the desktop. If you had minimized the window, the Visualizer restores it.
- If you haven't visualized the dataset in this session, the Visualizer chooses a method based on how well your dataset matches what is best shown for each kind of visualization method. You can enable and disable this feature from the Options menu in the Visualizer Directory Window.
- If you previously visualized a dataset but you've killed its window, the Visualizer creates a new Data Window by using the most recent visualization method.

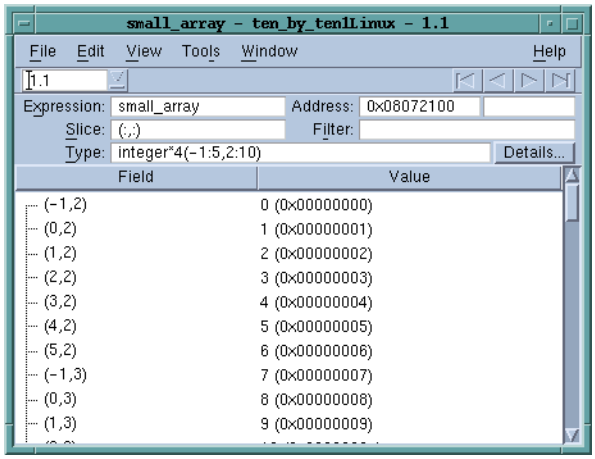
Visualizing Data Manually

Before you can visualize an array, you must do the following:

- Open a Variable Window for the array's data.
- Stop program execution when the array's values are set to what you want them to be when they are visualized.

The next figure shows a Variable Window that contains an array.

Figure 111: A Variable Window



You can restrict the data being visualized by editing the **Type** and **Slice** fields. For example, editing the Slice field limits the amount of data being visualized. (See "*Displaying Array Slices*" on page 279.) Limiting the amount of data increases the speed of the Visualizer.

After selecting the Variable Window **Tools** > **Visualize** command, the Visualizer begins executing and then creates its window. The data sent to the Visualizer isn't automatically updated as you step through your program. Instead, you must explicitly update the display by selecting the **Tools** > **Visualize** again.

TotalView can visualize laminated variables. (See "*Visualizing a Laminated Variable Window*" on page 292.) When you visualize a laminated variable, the Visualizer use the process or thread index as one dimension. This means that you can only visualize scalar or vector information. If you do not want the process or thread index to be a dimension, use a nonlaminated display.

Visualizing Data Programmatically

The **$visualize** function lets you add visualization expressions in evaluation action points or with expressions entered in the **Tools** > **Evaluate** Window. If you enter this function in an expression, TotalView interprets rather than compiles the expression, which can greatly decrease performance. See "*Defining Eval Points and Conditional Breakpoints*" on page 306 for information about compiled and interpreted expressions. Adding this function also lets you visualize several different variables from a single expression or eval point.

Using the **$visualize** function in an eval point lets you animate the changes that occur in your data, because the Visualizer updates the array's display

every time TotalView reaches the eval point. The following is the syntax for the **$visualize** function:

$visualize (*array* [, *slice_string*])

The *array* argument names the dataset being visualized. The optional *slice_string* argument is a quoted string that defines a constant slice expression that modifies the *array* parameter's dataset. In Fortran, you must use a single quotation mark ('). You can use either a single- or double-quotation mark if your language is C or C++.

The following examples show how you can use this function. Notice that the array's dimension ordering differs between C and in Fortran.

C	`$visualize(my_array);`
	`$visualize (my_array,"[::2][10:15]");`
	`$visualize (my_array,"[12][:]");`
Fortran	`$visualize (my_array)`
	`$visualize (my_*array,'(11:16,::2)')`
	`$visualize (my_array,'(:,13)')`

The first example in each programming language group visualizes the entire array. The second example selects every second element in the array's major dimension; it also clips the minor dimension to all elements in the range. The third example reduces the dataset to a single dimension by selecting one subarray.

You may need to cast your data so that TotalView knows what the array's dimensions are. In the following example, the C function declaration passes a two-dimensional array parameter. It does not specify the major dimension's extent.

```
void my_procedure (double my_array[][32])
{ /* procedure body */ }
```

The following example casts an array so that TotalView can visualize it:

```
$visualize (*(double[32][32]*)my_array);
```

Sometimes, it's hard to know what to specify. You can quickly refine array and slice arguments, for example, by entering the **$visualize** function into the **Tools** > **Evaluate** Dialog Box. When you select the Evaluate button, you quickly see the result. You can even use this technique to display several arrays simultaneously.

Using the Visualizer

The Visualizer uses two types of windows:

- **A Using Directory Window Commands**

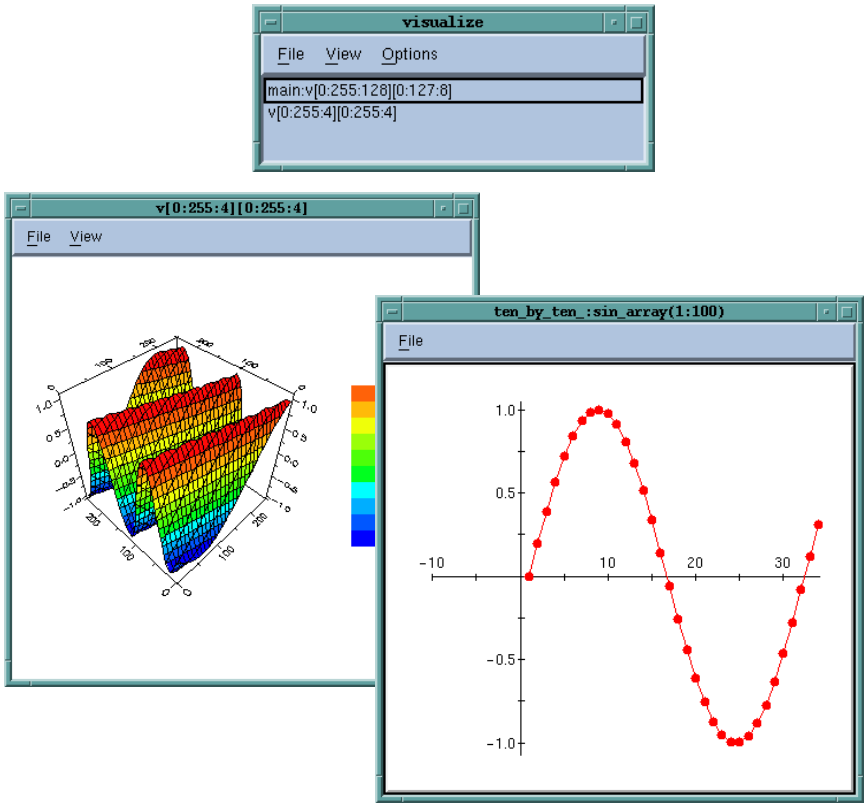
  This window lists the datasets that you can visualize. Use this window to set global options and to create views of your datasets. Commands in this window let you obtain different views of the same data by opening more than one Data Window.

- **Using Data Windows Commands**

  These are the windows that display your data. The commands in a Data Window let you set viewing options and change the way the Visualizer displays your data.

The top window in is a Directory Window. The two remaining windows show a surface and a graph view.

Figure 112: Sample Visualizer Windows



Using Directory Window Commands

The Directory Window contains a list of the datasets you can display. Double-clicking the dataset tells the Visualizer to display it.

The **View** menu lets you select **Graph** or **Surface** visualization. Whenever TotalView sends a new dataset to the Visualizer, the Visualizer updates its dataset list. To delete a dataset from the list, click on it, display the File menu, and then select Delete. (It's usually easier to just close the Visualizer.)

The following commands are in the Directory Window menu bar:

File > Delete Deletes the currently selected dataset. It removes the dataset from the dataset list and *destroys* the Data Windows that displays it.

File > Exit Closes all windows and exits the Visualizer.

View > Graph Creates a new Graph Window; see "*Using the Graph Window*" on page 147.

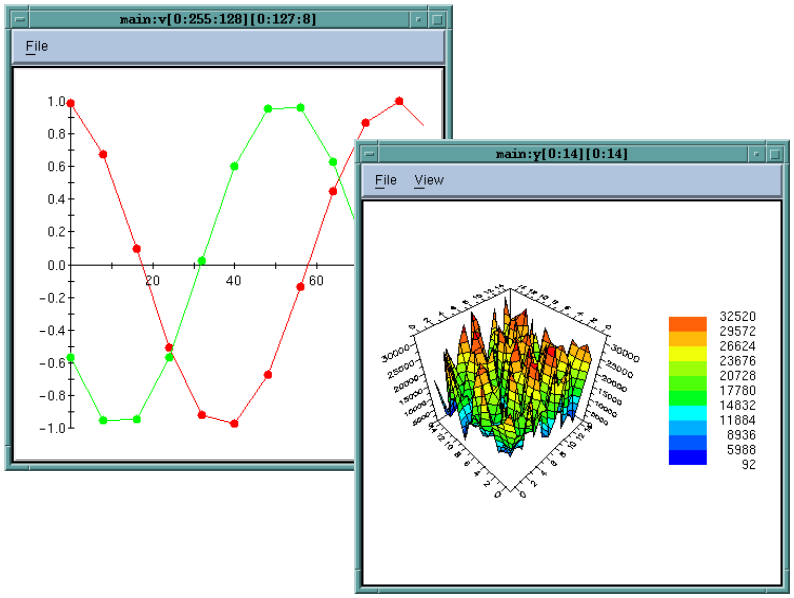
View > Surface Creates a new Surface Window; see "*Using the Surface Window*" on page 149.

Options > Auto Visualize
This item is a toggle; when enabled, the Visualizer automatically visualizes new datasets as they are read. Typically, this option is left on. If, however, you have large datasets and need to configure how the Visualizer displays the graph, you should disable this option.

Using Data Windows Commands

Data Windows display graphic images of your data. The following figure shows a graph view and a surface view. Every Data Window contains a menu bar and a drawing area. The Data Window title is its dataset identification.

Figure 113: A 2D and a 3D Visualizer Window



The Data Window menu commands are as follows:

File > Close Closes the Data Window.

File > Delete Deletes the Data Window dataset from the dataset list. This also destroys other Data Windows that view the dataset.

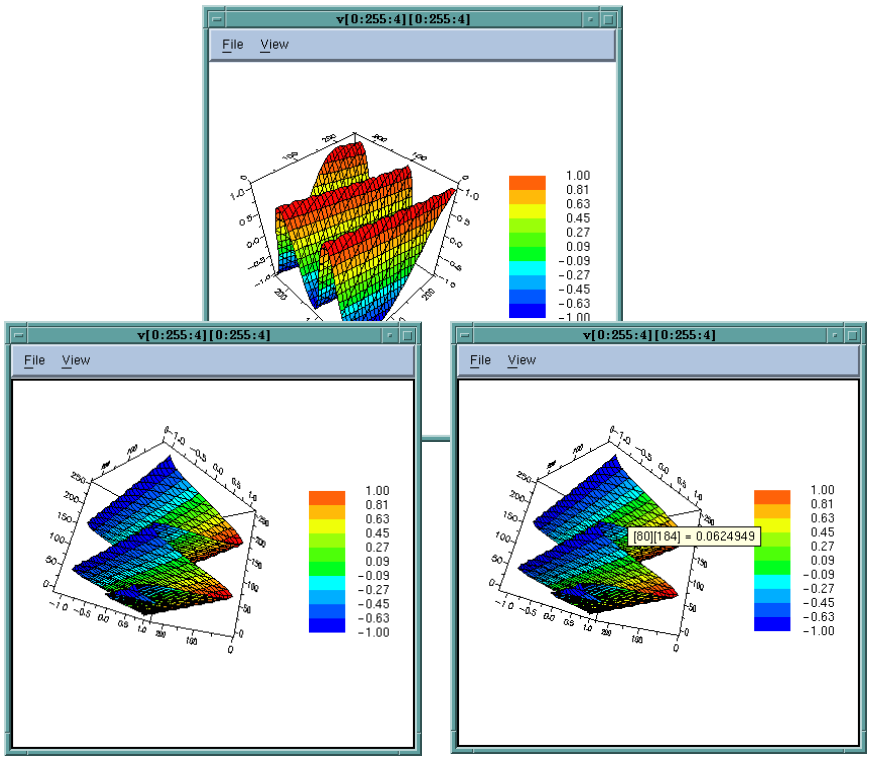
File > Directory Raises the Directory Window to the front of the desktop. If you minimized the Directory Window, the Visualizer restores it.

File > New Base Window
Creates a new Data Window that has the same visualization method and dataset as the current Data Window.

File > Options Pops up a window of viewing options.

The drawing area displays the image of your data. You can interact with the drawing area to alter the view of your data. For example, if the Visualizer is showing a surface, you can rotate the surface to view it from different angles. You can also get the value and indices of the dataset element nearest the cursor by clicking on it. A pop-up window displays the information.

Figure 114: Rotating and Querying



Using the Graph Window

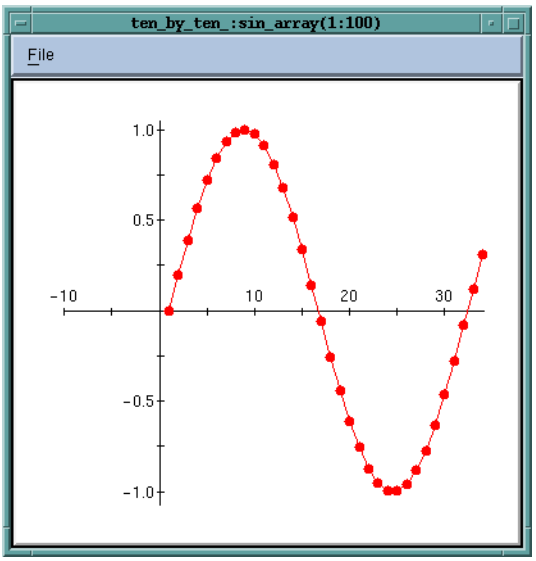
The Graph Window displays a two-dimensional graph of one- or two-dimensional datasets. If the dataset is two-dimensional, the Visualizer displays multiple graphs. When you first create a Graph Window on a two-dimensional dataset, the Visualizer uses the dimension with the larger number of elements for the X axis. It then draws a separate graph for each subarray that has the smaller number of elements. If you don't like this choice, you can transpose the data.

You probably don't want to use a graph to visualize two-dimensional datasets with large extents in both dimensions as the display can be very cluttered.

You can display graphs with markers for each element of the dataset, with lines connecting dataset elements, or with both lines and markers as shown in this figure. If the Visualizer is displaying more than one graph,

each is displayed in a different color. The X axis of the graph is annotated with the indices of the long dimension. The Y axis shows you the data value.

Figure 115: Visualizer Graph Data Window

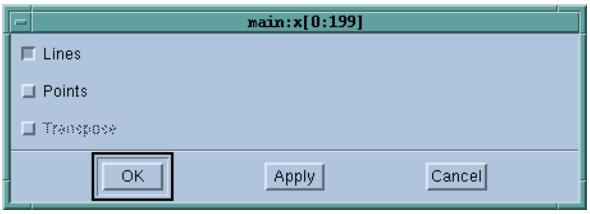


You can scale and translate the graph, or pop up a window that displays the indices and values for individual dataset elements.

Displaying Graphs

The **File** > **Options** Dialog Box lets you control how the Visualizer displays the graph.

Figure 116: Graph Options Dialog Box



The following describes the meanings of these check boxes:

Lines	If this is set, the Visualizer displays lines connecting dataset elements.
Points	If this is set, the Visualizer displays markers for dataset elements.
Transpose	If this is set, the Visualizer inverts the X and Y axes of the displayed graph.

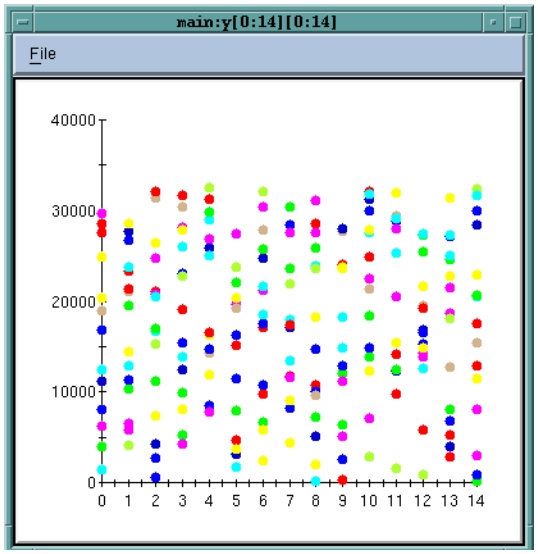
Manipulating Graphs

You can manipulate the way the Visualizer displays a graph by using the following actions:

Scale	Press the **Control key** and hold down the **middle-mouse** button. Move the mouse down to zoom in on the center of the drawing area, or up to zoom out.
Translate	Press the **Shift key** and hold down the **middle-mouse** button. Moving the mouse drags the graph.
Zoom	Press the **Control key** and hold down the **left-mouse** button. Drag the mouse to create a rectangle that encloses an area. The Visualizer scales the graph to fit the drawing area.
Reset View	Select **View** > **Reset** to reset the display to its initial state.
Query	Hold down the **left-mouse** button near a graph marker. A window pops up that displays the dataset element's indices and values.

The following figure shows a graph view of two-dimensional random data created by selecting **Points** and clearing lines in the Data Window **File** > **Options** Dialog Box.

Figure 117: A 2D Point Display

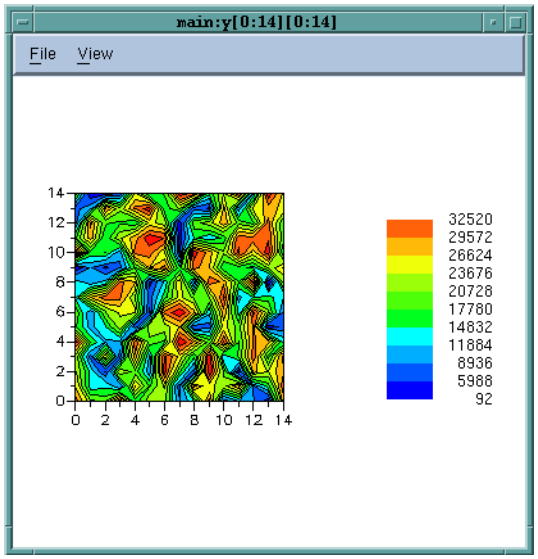


Using the Surface Window

The Surface Window displays two-dimensional datasets as a surface in two or three dimensions. The dataset's array indices map to the first two dimensions (**X** and **Y** axes) of the display. The following figure shows a two-dimensional map, where the dataset values are shown using only the **Zone** option. (This demarcates ranges of element values.) For a zone map with

contour lines, turn the **Zone** and **Contour** settings on, and turn the **Mesh** and **Shade** settings off.

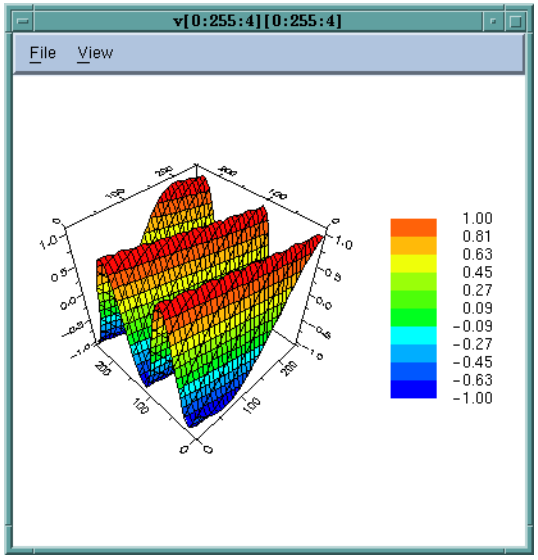
Figure 118: A 2D Surface Display



You can display random data by selecting only the **Zone** setting and turning the **Mesh, Shade**, and **Contour** settings off. The display shows where the data is located, and you can click the display to get the values of the data points.

The following figure shows a three-dimensional surface that maps element values to the height (**Z** axis).

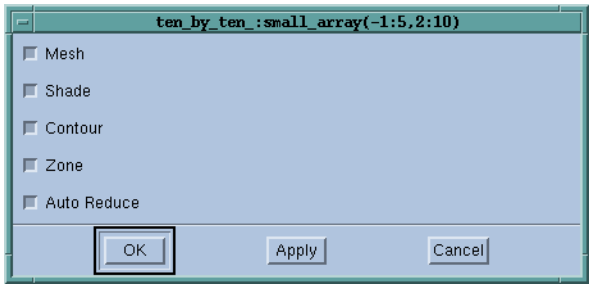
Figure 119: A 3D Surface Display



Displaying Surface Data

The Surface Window **File** > **Options** command lets you control how the Visualizer displays the graph.

Figure 120: Surface Options Dialog Box



This dialog box has the following choices:

Mesh — If this option is set, the Visualizer displays the surface as a three-dimensional mesh, with the **X-Y** grid projected onto the surface. If you don't set this or the **Shade** option, the Visualizer displays the surface in two dimensions.

Shade — If this option is set, the Visualizer displays the surface in three dimensions and shaded either in a flat color to differentiate the top and bottom sides of the surface, or in colors that correspond to the value if the **Zone** option is also set. When neither this nor the **Mesh** option is set, the Visualizer displays the surface in two dimensions.

Contour — If this option is set, the Visualizer displays contour lines that indicate ranges of element values.

Zone — If this option is set, the Visualizer displays the surface in colors that show ranges of element values.

Auto Reduce — If this option is set, the Visualizer derives the displayed surface by averaging over neighboring elements in the original dataset. This speeds up visualization by reducing the resolution of the surface. Clear this option if you want to accurately visualize all dataset elements.

The **Auto Reduce** option lets you choose between viewing all your data points—which takes longer to appear in the display—or viewing the averaging of data over a number of nearby points.

You can reset the viewing parameters to those used when you first invoked the Visualizer by selecting the **View** > **Reset** command, which restores all translation, rotation, and scaling to its initial state, and enlarges the display area slightly.

Manipulating Surface Data

The following commands change the display or give you information about it:

Query	Hold down the **left-mouse** button near the surface. A window pops up that displays the nearest dataset element's indices and value.
Rotate	Hold down the **middle-mouse** button and drag the mouse to freely rotate the surface. You can also press the **X**, **Y**, or **Z** keys to select a single axis of rotation. The Visualizer lets you rotate the surface in two dimensions simultaneously. (Rotating is shown in the next figure.) While you're rotating the surface, the Visualizer displays a wire-frame bounding box of the surface and moves it as your mouse moves.
Scale	Press the **Control key** and hold down the **middle-mouse** button. Move the mouse down to zoom in on the center of the drawing area, or up to zoom out.
Translate	Press the **Shift key** and hold down the **middle-mouse** button. Moving the mouse drags the surface.
Zoom	Press the **Control key** and hold down the **left mouse** button. Drag the mouse button to create a rectangle that encloses the area of interest. The Visualizer then translates and scales the area to fit the drawing area. (Zooming is shown in the next figure.)

Launching the Visualizer from the Command Line

To start the Visualizer from the shell, use the following syntax:

visualize [**–file** *filename* | **–persist**]

where:

–file *filename*	Reads data from *filename* instead of reading from standard input.
–persist	Continues to run after encountering an EOF (End-of-File) on standard input. If you don't use this option, the Visualizer exits as soon as it reads all of the data.

By default, the Visualizer reads its datasets from standard input and exits when it reads an EOF. When started by TotalView, the Visualizer reads its data from a pipe, ensuring that the Visualizer exits when TotalView does. If you want the Visualizer to continue to run after it exhausts all input, invoke it by using the **–persist** option.

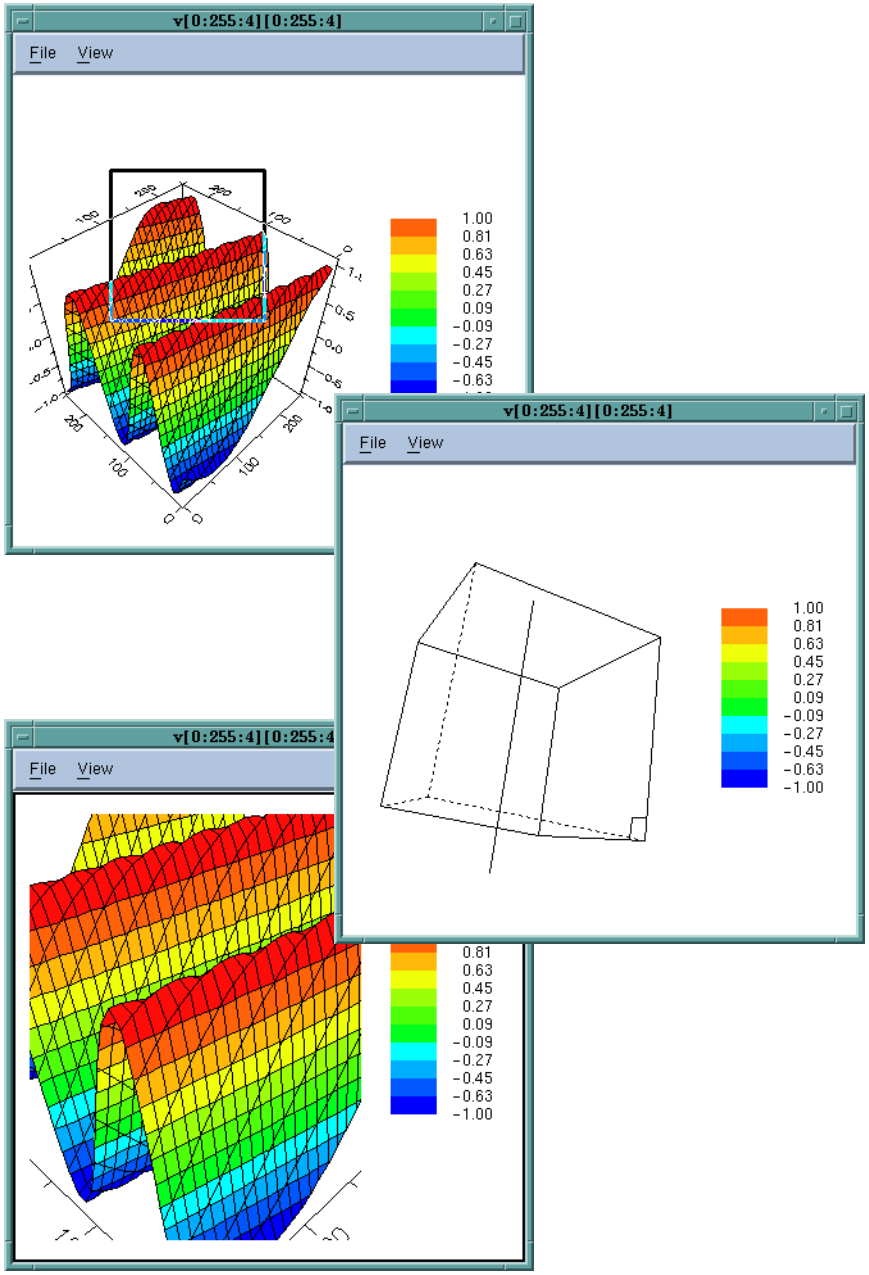
If you want to read data from a file, invoke the Visualizer with the **–file** option:

visualize –file *my_data_set_file*

The Visualizer reads all the datasets in the file. This means that the images you see represent the last versions of the datasets in the file.

The Visualizer supports the generic X toolkit command-line options. For example, you can start the Visualizer with the Directory Window minimized

Figure 121: Zooming and Rotating About an Axis



by using the **–iconic** option. Your system manual page for the X server or the X *Window System User's Guide* by O'Reilly & Associates lists the generic X command-line options in detail.

You can also customize the Visualizer by setting X resources in your resource files or on the command line with the **–xrm resource_setting** option. Using X resources to modify the default behavior of TotalView or the TotalView Visualizer is described in greater detail on our Web site at **http://www.etnus.com/Support/docs/xresources/XResources.html**.

Part IV: Using the CLI

The chapters in this part of the book deal exclusively with the CLI. Most CLI commands must have a process/thread focus for what they do. See Chapter 11: *"Using Groups, Processes, and Threads"* on page 205 for more information.

Chapter 8: Seeing the CLI at Work

While you can use the CLI as a stand-alone debugger, using the GUI is usually easier. You will most-often use the CLI when you need to debug programs using very communication liens or when you need to create debugging functions that are unique to your program. This chapter presents a few Tcl macros in which TotalView CLI commands are embedded.

Most of these examples are simple. They are designed to give you a feel for what you can do.

Chapter 9: Using the CLI

You can use CLI commands without knowing much about Tcl, which is the approach taken in this chapter. This chapter tells you how to enter CLI commands and how the CLI and TotalView interact with one another when used in a non-graphical way.

Seeing the CLI at Work

8

The CLI is a command-line debugger that is completely integrated with TotalView. You can use it and never use the TotalView GUI, or you can use it and the GUI simultaneously. Because the CLI is embedded in a Tcl interpreter, you can also create debugging functions that exactly meet your needs. When you do this, you can use these functions in the same way that you use the debugger's built-in CLI commands.

This chapter contains macros that show how the CLI programmatically interacts with your program and with TotalView. Reading examples without bothering too much with details gives you an appreciation for what the CLI can do and how you can use it. With a basic knowledge of Tcl, you can make full use of all CLI features.

In each macro in this chapter, all Tcl commands that are unique to the CLI are displayed in bold. These macros perform the following tasks:

- "*Setting the* CLI EXECUTABLE_PATH *Variable*" on page 157
- "*Initializing an Array Slice*" on page 158
- "*Printing an Array Slice*" on page 159
- "*Writing an Array Variable to a File*" on page 160
- "*Automatically Setting Breakpoints*" on page 161

Setting the CLI EXECUTABLE_PATH Variable

The following macro recursively descends through all directories, starting at a location that you enter. (This is indicated by the *root* argument.) The macro ignores directories named in the *filter* argument. The result is set as the value of the CLI **EXECUTABLE_PATH** state variable.

```
# Usage:
#
#  rpath [root] [filter]
#
# If root is not specified, start at the current
# directory. filter is a regular expression that removes
# unwanted entries. If it is not specified, the macro
# automatically filters out CVS/RCS/SCCS directories.
#
# The TotalView search path is set to the result.

proc rpath {{root "."} {filter "/(CVS|RCS|SCCS)(/|$)"}}
{

   # Invoke the UNIX find command to recursively obtain
   # a list of all directory names below "root".
   set find [split [exec find $root -type d -print] \n]

   set npath ""

   # Filter out unwanted directories.
   foreach path $find {
      if {! [regexp $filter $path]} {
         append npath ":"
         append npath $path
      }
   }

   # Tell TotalView to use it.
   dset EXECUTABLE_PATH $npath
}
```

In this macro, the last statement sets the **EXECUTABLE_PATH** state variable. This is the only statement that is unique to the CLI. All other statements are standard Tcl.

The **dset** command, like most interactive CLI commands, begins with the letter **d**. (The **dset** command is only used in assigning values to CLI state variables. In contrast, values are assigned to Tcl variables by using the standard Tcl **set** command.)

Initializing an Array Slice

The following macro initializes an array slice to a constant value:

```
array_set (var lower_bound upper_bound val) {
   for {set i $lower_bound} {$i <= $upper_bound} {incr i}{
      dassign $var\($i) $val
   }
}
```

The CLI **dassign** command assigns a value to a variable. In this case, it is setting the value of an array element. Use this function as follows:

```
d1.<> dprint list3
 list3 = {
   (1) = 1 (0x0000001)
   (2) = 2 (0x0000001)
   (3) = 3 (0x0000001)
 }
d1.<> array_set list 2 3 99
d1.<> dprint list3
 list3 = {
   (1) = 1 (0x0000001)
   (2) = 99 (0x0000063)
   (3) = 99 (0x0000063)
 }
```

Printing an Array Slice

The following macro prints a Fortran array slice. This macro, like others shown in this chapter, relies heavily on Tcl and uses unique CLI commands sparingly.

```
proc pf2Dslice {anArray i1 i2 j1 j2 {i3 1} {j3 1} \
         {width 20}} {
   for {set i $i1} {$i <= $i2} {incr i $i3} {
      set row_out ""
      for {set j $j1} {$j <= $j2} {incr j $j3} {
         set ij [capture dprint $anArray\($i,$j\)]
         set ij [string range $ij \
               [expr [string first "=" $ij] + 1] end]
         set ij [string trimright $ij]
         if {[string first "-" $ij] == 1} {
            set ij [string range $ij 1 end]}
         append ij "                    "
         append row_out " " \
               [string range $ij 0 $width] " "
      }
      puts $row_out
   }
}
```

The CLI's dprint command lets you specify a slice. For example, you can type: **dprint a(1:4,1:4)**.

After invoking this macro, the CLI prints a two-dimensional slice **(i1:i2:i3, j1:j2:j3)** of a Fortran array to a numeric field whose width is specified by the **width** argument. This width doesn't include a leading minus sign (-).

All but one line is standard Tcl. This line uses the **dprint** command to obtain the value of one array element. This element's value is then captured into a variable. The CLI **capture** command allows a value that is normally printed to be sent to a variable. For information on the difference

between values being displayed and values being returned, see "*About* CLI *Output*" on page 170.

The following shows how this macro is used:

```
d1.<> pf2Dslice a 1 4 1 4
    0.841470956802 0.909297406673 0.141120001673-
0.756802499294
    0.909297406673-0.756802499294-0.279415488243
0.989358246326
    0.141120001673-0.279415488243 0.412118494510-
0.536572933197
   -0.756802499294 0.989358246326-0.536572933197-
0.287903308868
d1.<> pf2Dslice a 1 4 1 4 1 1 17
    0.841470956802 0.909297406673 0.141120001673-
0.756802499294
    0.909297406673-0.756802499294-0.279415488243
0.989358246326
    0.141120001673-0.279415488243 0.412118494510-
0.536572933197
   -0.756802499294 0.989358246326-0.536572933197-
0.287903308868
d1.<> pf2Dslice a 1 4 1 4 2 2 10
    0.84147095 0.14112000
    0.14112000 0.41211849
d1.<> pf2Dslice a 2 4 2 4 2 2 10
   -0.75680249 0.98935824
    0.98935824-0.28790330
d1.<>
```

Writing an Array Variable to a File

It often occurs that you want to save the value of an array so that you can analyze its results at a later time. The following macro writes array values to a file:

```
proc save_to_file {var fname} {
   set values [capture dprint $var]
   set f [open $fname w]

   puts $f $values
   close $f
}
```

The following example shows how you might use this macro. Using the **exec** command tells the shell's **cat** command to display the file that was just written.

```
d1.<> dprint list3
 list3 = {
   (1) = 1 (0x00000001)
   (2) = 2 (0x00000002)
   (3) = 3 (0x00000003)
 }
```

```
d1.<> save_to_file list3 foo
d1.<> exec cat foo
 list3 = {
   (1) = 1 (0x00000001)
   (2) = 2 (0x00000002)
   (3) = 3 (0x00000003)
 }
d1.<>
```

Automatically Setting Breakpoints

In many cases, your knowledge of what a program is doing lets you make predictions as to where problems occurs. The following CLI macro parses comments that you can include in a source file and, depending on the comment's text, sets a breakpoint or an eval point.

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

```
# make_actions: Parse a source file, and insert
# evaluation and breakpoints according to comments.
#
proc make_actions {{filename ""}} {

   if {$filename == ""} {
      puts "You need to specify a filename"
      error "No filename"
   }

      # Open the program's source file and initialize a
      # few variables.
   set fname [set filename]
   set fsource [open $fname r]
   set lineno 0
   set incomment 0

      # Look for "signals" that indicate the type of
      # action point; they are buried in the comments.
   while {[gets $fsource line] != -1} {
      incr lineno
      set bpline $lineno

            # Look for a one-line eval point. The
            # format is ... /* EVAL: some_text */.
            # The text after EVAL and before the "*/" in
            # the comment is assigned to "code".
      if [regexp "/\\* EVAL: *(.*)\\*/" $line all code] {
         dbreak $fname\#$bpline -e $code
         continue
      }
```

```
            # Look for a multiline eval point.
         if [regexp "/\\* EVAL: *(.*)" $line all code] {
            # Append lines to "code".
            while {[gets $fsource interiorline] != -1} {
               incr lineno

               # Tabs will confuse dbreak.
               regsub -all \t $interiorline \
                     "  " interiorline

               # If "*/" is found, add the text to "code",
               # then leave the loop. Otherwise, add the
               # text, and continue looping.
               if [regexp "(.*)\\*/" $interiorline \
                     all interiorcode]{
                  append code \n $interiorcode
                  break
               } else {
                  append code \n $interiorline
               }
            }
            dbreak $fname\#$bpline -e $code
            continue
         }
            # Look for a breakpoint.
         if [regexp "/\\* STOP: .*" $line] {
            dbreak $fname\#$bpline
            continue
         }
            # Look for a command to be executed by Tcl.
         if [regexp "/\\* *CMD: *(.*)\\*/" $line all cmd] {
            puts "CMD: [set cmd]"
            eval $cmd
         }
      }
      close $fsource
   }
```

The only similarity between this macro and the previous three is that almost all of the statements are Tcl. The only purely CLI commands are the instances of the **dbreak** command that set eval points and breakpoints.

The following excerpt from a larger program shows how to embed comments in a source file that is read by the **make_actions** macro:

```
...
struct struct_bit_fields_only {
   unsigned f3 : 3;
   unsigned f4 : 4;
   unsigned f5 : 5;
   unsigned f20 : 20;
   unsigned f32 : 32;
} sbfo, *sbfop = &sbfo;
...
```

```
int main()
{
   struct struct_bit_fields_only *lbfop = &sbfo;
...
   int i;
   int j;
   sbfo.f3 = 3;
   sbfo.f4 = 4;
   sbfo.f5 = 5;
   sbfo.f20 = 20;
   sbfo.f32 = 32;
...
   /* TEST: Check to see if we can access all the
      values */
   i=i;     /* STOP: */
   i=1;     /* EVAL: if (sbfo.f3 != 3) $stop; */
   i=2;     /* EVAL: if (sbfo.f4 != 4) $stop; */
   i=3;     /* EVAL: if (sbfo.f5 != 5) $stop; */
   ...
   return 0;
}
```

The **make_actions** macro reads a source file one line at a time. As it reads these lines, the regular expressions look for comments that begin with **/* STOP**, **/* EVAL**, and **/* CMD**. After parsing the comment, it sets a breakpoint at a *stop* line or an eval point at an *eval* line, or executes a command at a *cmd* line.

Using eval points can be confusing because eval point syntax differs from that of Tcl. In this example, the **$stop** function is built into TotalView (and the CLI). Stated differently, you can end up with Tcl code that also contains C, C++, Fortran, and TotalView functions, variables, and statements. Fortunately, you only use this kind of mixture in a few places and you'll know what you're doing.

Using the CLI 9

The two components of the Command Line Interface (CLI) are the Tcl-based programming environment and the commands added to the Tcl interpreter that lets you debug your program. This chapter looks at how these components interact, and describes how you specify processes, groups, and threads.

This chapter emphasizes interactive use of the CLI rather than using the CLI as a programming language because many of its concepts are easier to understand in an interactive framework. However, everything in this chapter can be used in both environments.

This chapter contains the following sections:

- "*About the Tcl and the* CLI" on page 165
- "*Starting the* CLI" on page 167
- "*About* CLI *Output*" on page 170
- "*Using Command Arguments*" on page 171
- "*Using Namespaces*" on page 172
- "*About* CLI *Prompt*" on page 172
- "*Using Built-in and Group Aliases*" on page 173
- "*How Parallelism Affects Behavior*" on page 174
- "*Controlling Program Execution*" on page 175

About the Tcl and the CLI

The TotalView CLI is built in version 8.0 of Tcl, so the debugger's CLI commands are built into Tcl. This means that the CLI is not a library of commands that you can bring into other implementations of Tcl. Because the Tcl you are running is the standard 8.0 version, the TotalView CLI supports all libraries and operations that run using version 8.0 of Tcl.

Integrating CLI commands into Tcl makes them intrinsic Tcl commands. This lets you enter and execute all CLI commands in exactly the same way as you enter and execute built-in Tcl commands. As CLI commands are also

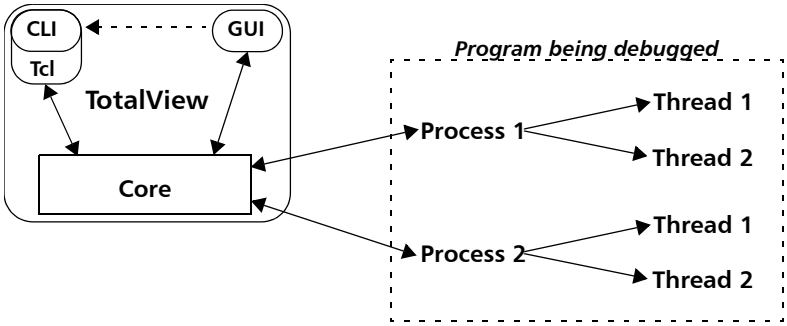
Tcl commands, you can embed Tcl primitives and functions in CLI commands, and embed CLI commands in sequences of Tcl commands.

For example, you can create a Tcl list that contains a list of threads, use Tcl commands to manipulate that list, and then use a CLI command that operates on the elements of this list. You can also create a Tcl function that dynamically builds the arguments that a process uses when it begins executing.

About The CLI and TotalView

The following figure illustrates the relationship between the CLI, the TotalView GUI, the TotalView core, and your program:

Figure 122: The CLI and TotalView



The CLI and GUI are components that communicate with the TotalView core, which is what actually does the work. In this figure, the dotted arrow between the GUI and the CLI indicates that you can invoke the CLI from the GUI. The reverse isn't true: you can't invoke the GUI from the CLI.

In turn, the TotalView core communicates with the processes that make up your program, receives information back from these processes, and passes information back to the component that sent the request. If the GUI is also active, the core also updates the GUI's windows. For example, stepping your program from within the CLI changes the PC in the Process Window, updates data values, and so on.

Using the CLI Interface

You interact with the CLI by entering a CLI or Tcl command. (Entering a Tcl command does exactly the same thing in the CLI as it does when interacting with a Tcl interpreter.) Typically, the effect of executing a CLI command is one or more of the following:

- The CLI displays information about your program.
- A change takes place in your program's state.
- A change takes place in the information that the CLI maintains about your program.

After the CLI executes your command, it displays a prompt. Although CLI commands are executed sequentially, commands executed by your program might not be. For example, the CLI doesn't require that your program be stopped when it prompts for and performs commands. It only requires that the last CLI command be complete before it can begin executing the

next one. In many cases, the processes and threads being debugged continue to execute after the CLI finished doing what you asked it to do.

If you need to stop an executing command or Tcl macro, press Ctrl+C while the command is executing. If the CLI is displaying its prompt, typing Ctrl+C stops executing processes.

Because actions are occurring constantly, state information and other kinds of messages that the CLI displays are usually mixed in with the commands that you type. You might want to limit the amount of information TotalView displays by setting the **VERBOSE** variable to **WARNING** or **ERROR**. (For more information, see the *"Variables"* chapter in the *TotalView Reference Guide*.)

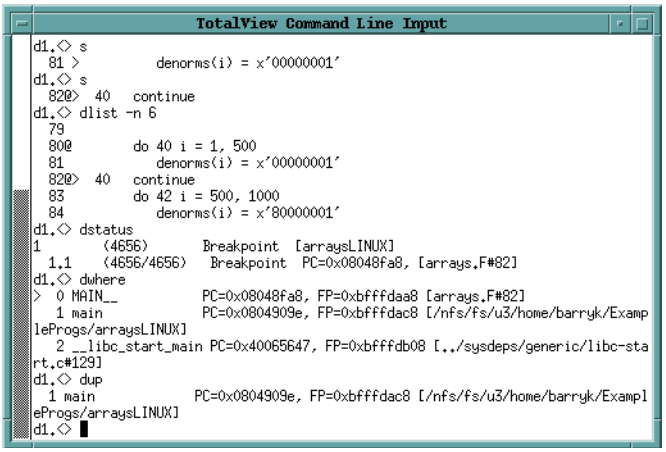
Starting the CLI

You can start the CLI in one of the following ways:

- You can start the CLI from the TotalView window by selecting the **Tools > Command Line** command in the Root or Process Windows. After selecting this command, TotalView opens a window into which you can enter CLI commands.
- You can start the CLI directly from a shell prompt by typing **totalviewcli**. (This assumes that the TotalView binary directory is in your path.)

Here is a snapshot of a CLI window that shows part of a program being debugged.

Figure 123: CLI *xterm Window*



If you have problems entering and editing commands, it might be because you invoked the CLI from a shell or process that manipulates your **stty** settings. You can eliminate these problems if you use the **stty sane** CLI command. (If the **sane** option isn't available, you have to change values individually.)

If you start the CLI with the **totalviewcli** command, you can use all of the command-line options that you can use when starting TotalView, except

those that have to do with the GUI. (In some cases, TotalView displays an error message if you try. In others, it just ignores what you did.

Startup Example

The following is a very small CLI script:

```
#
source make_actions.tcl
#
dload fork_loop
dset ARGS_DEFAULT {0 4 -wp}
dstep
catch {make_actions fork_loop.cxx} msg
puts $msg
```

This script begins by loading and interpreting the **make_actions.tcl** file, which was described in Chapter 8, "*Seeing the* CLI *at Work*," on page 157. It then loads the **fork_loop** executable, sets its default startup arguments, and steps one source-level statement.

If you stored this in a file named **fork_loop.tvd**, you can tell TotalView to start the CLI and execute this file by entering the following command:

```
totalviewcli -s fork_loop.tvd
```

Information on TotalView command-line options is in the "*TotalView Command Syntax*" chapter of the *TotalView Reference Guide*.

The following example places a similar set of commands in a file that you invoke from the shell:

```
#!/bin/sh
# Next line exec. by shell, but ignored by Tcl because: \
   exec totalviewcli -s "$0" "$@"
#
source make_actions.tcl
#
dload fork_loop
dset ARGS_DEFAULT {0 4 -wp}
dstep
catch {make_actions fork_loop.cxx} msg
puts $msg
```

The only real difference between the last two examples is the first few lines in the file. In this second example, the shell ignores the backslash continuation character; Tcl processes it. This means that the shell executes the **exec** command while Tcl will ignore it.

Starting Your Program

The CLI lets you start debugging operations in several ways. To execute your program from within the CLI, enter a **dload** command followed by the **drun** command. The following example uses the **totalviewcli** command to start the CLI. This is followed by **dload** and **drun** commands. Since this was not the first time the file was run, breakpoints exist from a previous session.

In this listing, the CLI prompt is "d1.<>". The information preceding the greater-than symbol (>) symbol indicates the processes and threads upon which the current command acts. The prompt is discussed in "About CLI Prompt" on page 172.

```
% totalviewcli
Copyright 1999-2004 by Etnus, LLC. ALL RIGHTS RESERVED.
Copyright 1999 by Etnus, Inc.
Copyright 1989-1996 by BBN Inc.
d1.<> dload arraysAlpha   #load the arraysAlpha program
1
d1.<> dactions            # Show the action points
No matching breakpoints were found
d1.<> dlist -n 10 75
   75          real16_array (i, j) = 4.093215 * j+2
   76 #endif
   77 26    continue
   78 27 continue
   79
   80 do 40 i = 1, 500
   81    denorms(i) = x'00000001'
   82 40 continue
   83 do 42 i = 500, 1000
   84    denorms(i) = x'80000001'
d1.<> dbreak 80            # Add two action points
1
d1.<> dbreak 83
2
d1.<> drun        # Run the program to the action point
```

This two-step operation of loading and running lets you set action points before execution begins. It also means that you can execute a program more than once. At a later time, you can use the **drerun** command to restart your program, perhaps sending it new command-line arguments. In contrast, reentering the **dload** command tells the CLI to reload the program into memory (for example, after editing and recompiling the program). The **dload** command always creates new processes. This means that you get a new process each time the CLI executes it. The CLI does not, however, remove older ones.

The **dkill** command terminates one or more processes of a program started by using a **dload**, **drun**, or **drerun** command. The following example continues where the previous example left off:

```
d1.<> dkill             # kills process
d1.<> drun              # runs program from start
d1.<> dlist -e -n 3     # shows lines about current spot
   79
   80@>      do 40 i = 1, 500
   81            denorms(i) = x'00000001'
d1.<> dwhat master_array # Tell me about master_array
In thread 1.1:
Name: master_array; Type: integer(100);
    Size: 400 bytes; Addr: 0x140821310
    Scope: ##arraysAlpha#arrays.F#check_fortran_arrays
    (Scope class: Any)
```

```
    Address class: proc_static_var
    (Routine static variable)
d1.<> dgo                        # Start program running
d1.<> dwhat denorms              # Tell me about denorms
In thread 1.1:
Name: denorms; Type: <void>; Size: 8 bytes;
    Addr: 0x1408214b8
    Scope: ##arraysAlpha#arrays.F#check_fortran_arrays
    (Scope class: Any)
    Address class: proc_static_var
    (Routine static variable)
d1.<> dprint denorms(0)        # Show me what is stored
 denorms(0) = 0x0000000000000001 (1)
d1.<>
```

Because information is interleaved, you may not realize that the prompt has appeared. It is always safe to use the Enter key to have the CLI redisplay its prompt. If a prompt isn't displayed after you press Enter, you know that the CLI is still executing.

About CLI Output

A CLI command can either print its output to a window or return the output as a character string. If the CLI executes a command that returns a string value, it also prints the returned string. Most of the time, you won't care about the difference between *printing* and *returning-and-printing*. Either way, the CLI displays information in your window. And, in both cases, printed output is fed through a simple *more* processor. (This is discussed in more detail in the next section.)

In the following two cases, it matters whether TotalView directly prints output or returns and then prints it:

- When the Tcl interpreter executes a list of commands, TotalView only prints the information returned from the last command. It doesn't show information returned by other commands.
- You can only assign the output of a command to a variable if the CLI returns a command's output. You can't assign output that the interpreter prints directly to a variable, or otherwise manipulate it, unless you save it using the **capture** command.

For example, the **dload** command returns the ID of the process object that was just created. The ID is normally printed—unless, of course, the **dload** command appears in the middle of a list of commands; for example:

```
{dload test_program;dstatus}
```

In this example, the CLI doesn't display the ID of the loaded program, since the **dload** command was not the last command.

When information is returned, you can assign it to a variable. For example, the next command assigns the ID of a newly created process to a variable:

```
set pid [dload test_program]
```

Because you can't assign the output of the **help** command to a variable, the following doesn't work:

```
set htext [help]
```

This statement assigns an empty string to **htext** because the **help** command doesn't return text. It just prints it.

To save the output of a command that prints its output, use the **capture** command. For example, the following example writes the **help** command's output into a variable:

```
set htext [capture help]
```

You can only capture the output from commands. You can't capture the informational messages displayed by the CLI *that describe process state. If you are using the* GUI, *TotalView also writes this information to the Root Window Log Pane. If it is being written there, you can use the* **File > Save Pane** *command to write this information to a file.*

'more' Processing

When the CLI displays output, it sends data through a simple *more*-like process. This prevents data from scrolling off the screen before you view it. After you see the **MORE** prompt, press Enter to see the next screen of data. If you type **q** (followed by pressing the Enter key), the CLI discards any data it hasn't yet displayed.

You can control the number of lines displayed between prompts by using the **dset** command to set the **LINES_PER_SCREEN** CLI variable. (For more information, see the *TotalView Reference Guide*.)

Using Command Arguments

The default command arguments for a process are stored in the **ARGS**(*num*) variable, where *num* is the CLI ID for the process. If you don't set the **ARGS**(*num*) variable for a process, the CLI uses the value stored in the **ARGS_DEFAULT** variable. TotalView sets the **ARGS_DEFAULT** variable when you use the **–a** option when starting the CLI or the GUI.

The **–a** *option tells TotalView to pass everything that follows on the command line to the program.*

For example:

```
totalviewcli –a argument-1, argument-2, ...
```

To set (or clear) the default arguments for a process, you can use the **dset** command to modify the **ARGS()** variables directly, or you can start the process with the **drun** command. For example, the following clears the default argument list for process 2:

```
dunset ARGS(2)
```

The next time process 2 is started, the CLI uses the arguments contained in **ARGS_DEFAULT**.

You can also use the **dunset** command to clear the **ARGS_DEFAULT** variable; for example:

```
dunset ARGS_DEFAULT
```

All commands (except the **drun** command) that can create a process—including the **dgo**, **drerun**, **dcont**, **dstep**, and **dnext** commands—pass the default arguments to the new process. The **drun** command differs in that it replaces the default arguments for the process with the arguments that are passed to it.

Using Namespaces

CLI interactive commands exist in the primary Tcl namespace (::). Some of the TotalView state variables also reside in this namespace. Seldom-used functions and functions that are not primarily used interactively reside in other namespaces. These namespaces also contain most TotalView state variables. (The variables that appear in other namespaces are usually related to TotalView preferences.) TotalView uses the following namespaces:

TV::	Contains commands and variables that you use when creating functions. They can be used interactively, but this is not their primary role.
TV::GUI::	Contains state variables that define and describe properties of the user interface, such as window placement and color.

If you discover other namespaces beginning with **TV**, you have found a namespace that contains private functions and variables. These objects can (and will) disappear, so don't use them. Also, don't create namespaces that begin with **TV**, since you can cause problems by interfering with built-in functions and variables.

The CLI **dset** command lets you set the value of these variables. You can have the CLI display a list of these variables by specifying the namespace; for example:

```
dset TV::
```

About CLI Prompt

The appearance of the CLI prompt lets you know that the CLI is ready to accept a command. This prompt lists the current focus, and then displays a greater-than symbol (>) and a blank space. (The *current focus* is the processes and threads to which the next command applies.) For example:

d1.<>	The current focus is the default set for each command, focusing on the first user thread in process 1.
g2.3>	The current focus is process 2, thread 3; commands act on the entire group.

t1.7>	The current focus is thread 7 of process 1.
gW3.>	The current focus is all worker threads in the control group that contains process 3.
p3/3	The current focus is all processes in process 3, group 3.

You can change the prompt's appearance by using the **dset** command to set the **PROMPT** state variable; for example:

```
dset PROMPT "Kill this bug! > "
```

Using Built-in and Group Aliases

Many CLI commands have an alias that let you abbreviate the command's name. (An alias is one or more characters that Tcl interprets as a command or command argument.)

The **alias** *command, which is described in the TotalView Reference Guide, lets you create your own aliases.*

For example, the following command tells the CLI to halt the current group:

```
dfocus g dhalt
```

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

```
f g h
```

You often type less-used commands in full, but some commands are almost always abbreviated. These commands include **dbreak** (**b**), **ddown** (**d**), **dfocus** (**f**), **dgo** (**g**), **dlist** (**l**), **dnext** (**n**), **dprint** (**p**), **dstep** (**s**), and **dup** (**u**).

The CLI also includes uppercase group versions of aliases for a number of commands, including all stepping commands. For example, the alias for **dstep** is **s**; in contrast, **S** is the alias for **dfocus g dstep**. (The first command tells the CLI to step the process. The second steps the control group.)

Group aliases differ from the group-level command that you type interactively, as follows:

- They do not work if the current focus is a list. The **g** focus specifier modifies the current focus, and it can only be applied if the focus contains just one term.
- They always act on the group, no matter what width is specified in the current focus. Therefore, **dfocus t S** does a step-group command.

How Parallelism Affects Behavior

A parallel program consists of some number of processes, each involving some number of threads. Processes fall into two categories, depending on when they are created:

- **Initial process**
  A preexisting process from the normal run-time environment (that is, created outside TotalView) or one that was created as TotalView loaded the program.
- **Spawned process**
  A new process created by a process executing under CLI control.

TotalView assigns an integer value to each individual process and thread under its control. This *process/thread identifier* can be the system identifier associated with the process or thread. However, it can be an arbitrary value created by the CLI. Process numbers are unique over the lifetime of a debugging session; in contrast, thread numbers are only unique while the process exists.

Process/thread notation lets you identify the component that a command targets. For example, if your program has two processes, and each has two threads, four threads exist:

Thread 1 of process 1
Thread 2 of process 1
Thread 1 of process 2
Thread 2 of process 2

You identify the four threads as follows:

1.1—Thread 1 of process 1
1.2—Thread 2 of process 1
2.1—Thread 1 of process 2
2.2—Thread 2 of process 2

Types of IDs

Multithreaded, multiprocess, and distributed programs contain a variety of IDs. The following types are used in the CLI and TotalView:

System PID This is the process ID and is generally called the PID.

System TID This is the ID of the system kernel or user thread. On some systems (for example, AIX), the TIDs have no obvious meaning. On other systems, they start at 1 and are incremented by 1 for each thread.

TotalView thread ID This is usually identical to the system TID. On some systems (such as AIX) where the threads have no obvious meaning), TotalView uses its own IDs.

pthread ID This is the ID assigned by the Posix pthreads package. If this differs from the system TID, it is a pointer value that points to the pthread ID.

Debugger PID This is an ID created by TotalView that lets it identify processes. It is a sequentially numbered value beginning at 1 that is incremented for each new process. If the target process is killed and restarted (that is, you use the **dkill** and **drun** commands), the debugger PID doesn't change. The system PID changes, since the operating system has created a new target process.

Controlling Program Execution

Knowing what's going on and where your program is executing is simple in a serial debugging environment. Your program is either stopped or running. When it is running, an event such as arriving at a breakpoint can occur. This event tells the debugger to stop the program. Sometime later, you tell the serial program to continue executing. multiprocess and multithreaded programs are more complicated. Each thread and each process has its own execution state. When a thread (or set of threads) triggers a breakpoint, TotalView must decide what it should do about the other threads and processes. Some may stop; some may continue to run.

Advancing Program Execution

Debugging begins by entering a **dload** or **dattach** command. If you use the **dload** command, you must use the **drun** command to start the program executing. These three commands work at the process level and you can't use them to start individual threads. (This is also true for the **dkill** command.)

To advance program execution, you enter a command that causes one or more threads to execute instructions. The commands are applied to a P/T set. (P/T sets are discussed in Chapter 2, "*About Threads, Processes, and Groups*," on page 15 and Chapter 11, "*Using Groups, Processes, and Threads*," on page 205.) Because the set doesn't have to include all processes and threads, you can cause some processes to be executed while holding others back. You can also advance program execution by increments, *stepping* the program forward, and you can define the size of the increment. For example, **dnext 3** executes the next three statements, and then pauses what you've been stepping.

Typically, debugging a program means that you have the program run, and then you stop it and examine its state. In this sense, a debugger can be thought of as tool that lets you alter a program's state in a controlled way. And debugging is the process of stopping a process to examine its state. However, the term *stop* has a slightly different meaning in a multiprocess, multithreaded program; in these programs, *stopping* means that the CLI holds one or more threads at a location until you enter a command that tells them to start executing again.

For more information, see Chapter 10, "*Debugging Programs*," on page 179.

Using Action Points

Action points tell the CLI to stop a program's execution. You can specify the following types of action points:

- A *breakpoint* (see **dbreak** in the *TotalView Reference Guide*) stops the process when the program reaches a location in the source code.
- A *watchpoint* (see **dwatch** in the *TotalView Reference Guide*) stops the process when the value of a variable is changed.
- A *barrier point* (see **dbarrier** in the *TotalView Reference Guide*), as its name suggests, effectively prevents processes from proceeding beyond a point until all other related processes arrive. This gives you a method for synchronizing the activities of processes. (You can only set a barrier POINT on processes; you can't set then on individual threads.)
- An *eval point* (see **dbreak** in the *TotalView Reference Guide*) lets you programmatically evaluate the state of the process or variable when execution reaches a location in the source code. eval points typically do not stop the process; instead, they perform an action. In most cases, an eval point stops the process when some condition that you specify is met.

For extensive information on action points, see "Setting Action Points" on page 293.

Each action point is associated with an *action point identifier*. You use these identifiers when you need to refer to the action point. Like process and thread identifiers, action point identifiers are assigned numbers as they are created. The ID of the first action point created is 1; the second ID is 2, and so on. These numbers are never reused during a debugging session.

The CLI and TotalView only let you assign one action point to a source code line, but you can make this action point as complex as you need it to be.

Part V: Debugging

The chapters in this part of the *TotalView Users Guide* describe how you actually go about debugging your programs. The preceding chapters describe, for the most part, what you need to do before you get started with TotalView. In contrast, the chapters in this part are what TotalView is really about.

Chapter 10: Debugging Programs
: Read this chapter to help you find your way around your program. This chapter describes how to your program under the debugger's control, the ways to step your program's execution, and how to halt, terminate, and restart your program.

Chapter 11: Using Groups, Processes, and Threads
: The stepping information in Chapter 10 describes the commands and the different types of stepping. In a multiprocess, multithreaded program, you may need to finely control what is executing. This chapter tells you how to do this.

Chapter 12: Examining and Changing Data
: As your program executes, you will want to examine what the value stored in a variable is. This chapter tells you how.

Chapter 13: Examining Arrays
: Displaying information in arrays presents special problems. This chapter tells how TotalView solves these problems.

Chapter 14: Setting Action Points
: This chapter discusses action points. *Action points* let you control how your programs execute and what happens when your program reaches statements that you define as important. Action points also let you monitor changes to a variable's value.

Debugging Programs 10

This chapter explains how to perform basic debugging tasks with TotalView.

This chapter contains the following sections:

- "*Searching and Looking For Program Elements*" on page 179
- "*Editing Source Text*" on page 183
- "*Manipulating Processes and Threads*" on page 183
- "*Using Stepping Commands*" on page 193
- "*Executing to a Selected Line*" on page 195
- "*Executing Out of a Function*" on page 196
- "*Continuing with a Specific Signal*" on page 196
- "*Deleting (Killing) Programs*" on page 197
- "*Restarting Programs*" on page 197
- "*Checkpointing*" on page 197
- "*Fine-Tuning Shared Library Use*" on page 198
- "*Setting the Program Counter*" on page 202
- "*Interpreting the Status and Control Registers*" on page 203

Searching and Looking For Program Elements

TotalView provides several ways for you to navigate and find information in your source file.

Topics in this section are:

- "*Searching for Text*" on page 180
- "*Looking for Functions and Variables*" on page 180
- "*Finding the Source Code for Functions*" on page 181
- "*Finding the Source Code for Files*" on page 182
- "*Resetting the Stack Frame*" on page 182

Searching for Text

GUI

You can search for text strings in most windows using the **Edit** > **Find** command. which displays the following dialog box.

Figure 124: Edit > Find Dialog Box

Controls in this dialog box let you:

- Perform case-sensitive searches.
- Continue searching from the beginning of the file if the string isn't found in the region beginning at the currently selected line and ending at the last line of the file.
- Keep the dialog box displayed.
- Tell TotalView to search towards the bottom of the file (**Down**) or the top (**Up**).

After you have found a string, you can find another instance of it by using the **Edit** > **Find Again** command.

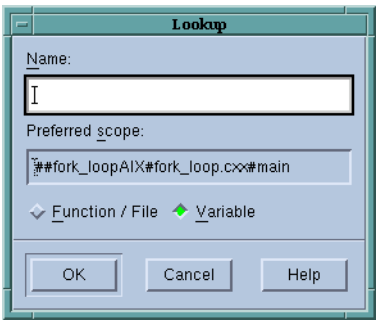
If you searched for the same string previously, you can select it from the pulldown list on the right side of the **Find** text box.

Looking for Functions and Variables

GUI

Having TotalView locate a variable or a function is usually easier than scrolling through your sources to look for it. Do this with the **View** > **Lookup Function** and **View** > **Lookup Variable** commands. Here's the dialog box that these commands display:

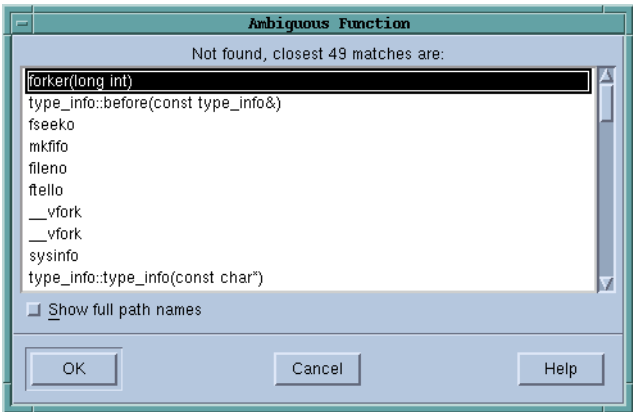
Figure 125: View > Lookup Variable Dialog Box



CLI: **dprint** *variable*

If TotalView doesn't find the name and it can find something similar, it displays a dialog box that contains the names of functions that might match.

Figure 126: Ambiguous Function Dialog Box



If the one you want is listed, click on its name and then choose **OK** to display it in the Source Pane.

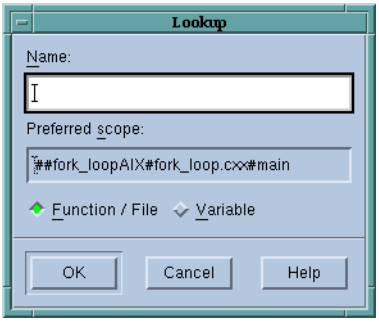
Finding the Source Code for Functions

Use he **File** > **Open Source** command to search for a function's declaration.

CLI: **dlist** *function-name*

This command tells TotalView to display the following dialog box:

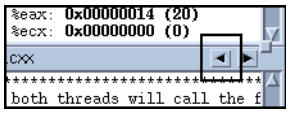
Figure 127: View > Lookup Function Dialog Box



After locating your function, TotalView displays it in the Source Pane. If you didn't compile the function using the **–g** command-line option, TotalView displays disassembled machine code.

When you want to return to the previous contents of the Source Pane, use the Backward button located in the upper-right corner of the Source Pane and just below the Stack Frame Pane. In the following figure, a square surrounds this button.

Figure 128: Undive/Dive Controls



You can also use the **View > Reset** command to discard the dive stack so that the Source Pane is displaying the PC it displayed when you last stopped execution.

Another method of locating a function's source code is to dive into a source statement in the Source Pane that shows the function being called. After diving, you see the source.

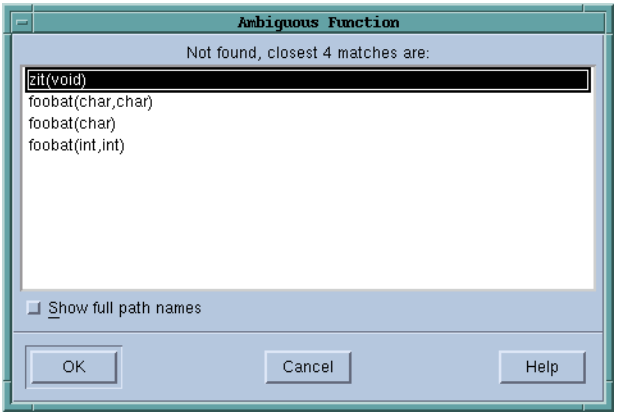
Resolving Ambiguous Names

Sometimes the function name you specify is ambiguous. For example, you might have specified the name of:

- A static function, and your program contains different versions of it.
- A member function in a C++ program, and multiple classes have a member function with that name.
- An overloaded function or a template function.

The following figure shows the dialog box that TotalView displays when it encounters an ambiguous function name. You can resolve the ambiguity by clicking the function name.

Figure 129: Ambiguous Function Dialog Box



If the name being displayed isn't enough to identify which name you need to select, select the **Show full path names** check box to display additional information.

Finding the Source Code for Files

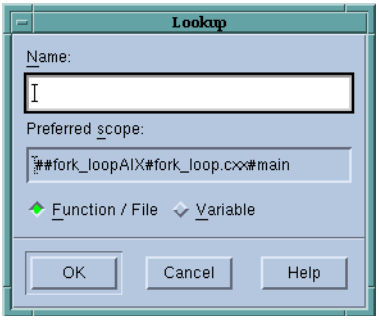
You can display a file's source code by selecting the **View > Lookup Function** command and entering the file name in the dialog box shown in the figure on th next page.

If a header file contains source lines that produce executable code, you can display the file's code by typing the file name here.

Resetting the Stack Frame

After moving around your source code to look at what's happening in different places, you can return to the executing line of code for the current stack frame by selecting the **View > Reset** command. This command places the PC arrow on the screen.

Figure 130: View > Lookup Function Dialog Box



This command is also useful when you want to undo the effect of scrolling, or when you move to different locations using, for example, the **View > Lookup Function** command.

If the program hasn't started running, the **View > Reset** command displays the first executable line in your main program. This is useful when you are looking at your source code and want to get back to the first statement that your program executes.

Editing Source Text

Use the **File > Edit Source** command to examine the current routine in a text editor. TotalView uses an *editor launch string* to determine how to start your editor. TotalView expands this string into a command that TotalView sends to the **sh** shell.

The default editor is **vi**. However, TotalView uses the editor named in an **EDITOR** environment variable, or the editor you name in the Source Code Editor field of the **File > Preferences** Launch Strings Page. The online Help for this page contains information on setting this preference.

Manipulating Processes and Threads

Topics discussed in this section are:

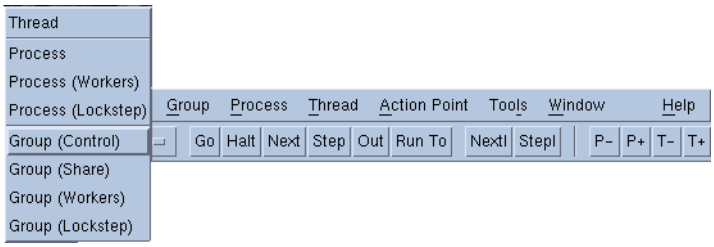
- "*Using the Toolbar to Select a Target*" on page 184
- "*Stopping Processes and Threads*" on page 184
- "*Updating Process Information*" on page 185
- "*Holding and Releasing Processes and Threads*" on page 185
- "*Using Barrier Points*" on page 187
- "*Holding Problems*" on page 188
- "*Examining Groups*" on page 188
- "*Displaying Groups*" on page 190
- "*Placing Processes in Groups*" on page 190
- "*Starting Processes and Threads*" on page 190
- "*Creating a Process Without Starting It*" on page 191

- "*Creating a Process by Single-Stepping*" on page 191
- "*Stepping and Setting Breakpoints*" on page 192

Using the Toolbar to Select a Target

The Process Window toolbar has three sets of buttons. The first set is a single pulldown list. It defines the *focus* of the command selected in the second set of the toolbar. The third set changes the process and thread being displayed. The following figure shows this toolbar.

Figure 131: The Toolbar



When you are doing something to a multiprocess, multithreaded program, TotalView needs to know which processes and threads it should act on. In the CLI, you specify this target using the **dfocus** command. When using the GUI, you specify the focus using this pulldown. For example, if you select **Thread**, and then select the **Step** button, TotalView steps the current thread. In contrast, if you select **Process Workers** and then select the **Go** button, TotalView tells all the processes that are in the same workers group as the current thread to start executing. (This thread is called the *thread of interest*.)

Chapter 11, "Using Groups, Processes, and Threads," on page 205 describes how TotalView manages processes and threads. While TotalView gives you the ability to control the precision your application requires, most applications do not need this level of interaction. In almost all cases, using the controls in the toolbar gives you all the control you need.

Stopping Processes and Threads

To stop a group, process, or thread, select a **Halt** command from the **Group**, **Process**, or **Thread** pulldown menu on the toolbar.

CLI:	**dhalt** **Halts a group, process, or thread. Setting the focus changes the scope.**

The three **Halt** commands differ in the scope of what they halt. In all cases, TotalView uses the current thread, which is called the thread of interest or TOI, to determine what else it will halt. For example, selecting **Process > Halt** tells TotalView to determine the process in which the TOI is running. It then halts this process. Similarly, if you select **Group > Share > Halt**, TotalView determines what processes are in the share group that the current thread participates in. It then stops all of these processes.

For more information on the Thread of Interest, see "Defining the GOI, POI, and TOI" on page 205.

When you select the **Halt** button in the toolbar instead of the commands in the menubar, TotalView decides what it should stop based on what is set in the two toolbar pulldown lists.

After entering a **Halt** command, TotalView updates any windows that can be updated. When you restart the process, execution continues from the point where TotalView stopped the process.

Updating Process Information

GUI

Normally, TotalView only updates information when the thread being executed stops executing. You can force TotalView to update a window by using the **Window > Update** command. You need to use this command if you want to see what a variable's value is while your program is executing.

When you use this command, TotalView momentarily stops execution so that it can obtain the information that it needs. It then restarts the thread.

Holding and Releasing Processes and Threads

Many times when you are running a multiprocess or multithreaded program, you want to synchronize execution to the same statement. You can do this manually using a *hold* command, or you can let TotalView do this by setting a barrier point.

When a process or a thread is *held*, any command that it receives that tells it to execute are ignored. For example, assume that you place a hold on a process in a control group that contains three processes. After you select **Group > Share > Go**, two of the three processes resume executing. The held process ignores the **Go** command.

At a later time, you will want to run what is being held. Do this using a **Release** command. When you release a process or a thread, you are telling it that it can run. But you still need to tell it to execute, which means that it is waiting to receive an execution command, such as **Go**, **Out**, or **Step**.

Manually holding and releasing processes and threads is useful in the following instances:

- When you need to run a subset of the processes and threads. You can manually hold all but the ones you want to run.
- When a process or thread is held at a barrier point and you want to run it without first running all the other processes or threads in the group to that barrier. In this case, you release the process or the thread manually, and then run it.

TotalView can also hold a process or thread if it stops at a barrier breakpoint. You can manually release a process or thread being held at a barrier breakpoint. See "*Setting Barrier Points*" on page 303 for more information on manually holding and releasing barrier breakpoint.

When TotalView is holding a process, the Root and Process Windows display a held indicator, which is the uppercase letter **H**. When TotalView is holding a thread, it displays a lowercase **h**.

You can hold or release a thread, process, or group of processes in one of the following ways:

- You can hold a group of processes using the **Group** > **Hold** command.
- You can release a group of processes using the **Group** > **Release** command.
- You can toggle the hold/release state of a process by selecting and clearing the **Process** > **Hold** command.
- You can toggle the hold/release state of a thread by selecting and clearing the **Thread** > **Hold** command.

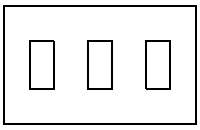
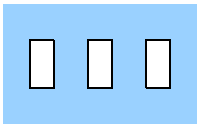
CLI: dhold and dunhold
Setting the focus changes the scope.

If a process or a thread is running when you use a hold or release command, TotalView stops the process or thread, and then holds it.TotalView lets you hold and release processes independently from threads.

The Process pulldown menu contains a **Hold Threads** and a **Release Threads** command. Although they appear to do the same thing, they are used in slightly different ways. When you use the **Hold Threads** commands on a multithreaded process, you place a hold on all threads. This is seldom what you want as you really do want something to run. After selecting this command, go to the thread that you want to run and then clear the **Thread** > **Hold** command so that TotalView lets it run. This may appear awkward, but it is actually an easy way to select one or two threads when your program has a lot of threads. You can verify that you're doing the right thing by looking at the thread's status in the Root Window Attached Pane.

CLI: dhold –thread
dhold –process
dunhold –thread

The following set of drawings presents examples of using hold commands:

Held/Release State	What Can Be Run Using Process > Go
	This figure shows a process with three threads. Before you do anything, all threads in the process can be run.
	Select the **Process** > **Hold** toggle. The blue indicates that you held the process. (Or, at least its in blue if you are viewing this online.) Nothing runs when you select **Process** > **Go**.

Held/Release State	What Can Be Run Using Process > Go
	Go to the **Threads** menu. The button next to the **Hold** command isn't selected. This is because the *thread hold* state is independent from the *process hold* state. Select it. The circle indicates that thread 1 is held. At this time, there are two different holds on thread 1. One is at the process level; the other is at thread level. Nothing will run when you select **Process** > **Go**.
	Select the **Process** > **Hold** command. Select **Process** > **Go**. The second and third threads run.
	Select **Process** > **Release Threads**. This releases the hold placed on the first thread by the **Thread** > **Hold** command. After you select **Process** > **Go**, all threads run.

Using Barrier Points

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

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

Why breakpoints don't work (part 1)

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

Why breakpoints don't work (part 2)

If you set the breakpoint's property so that only the process hitting the breakpoint stops, you have a better chance of getting processes there. However, you should not have other breakpoints between where the program is currently at and this breakpoint. If processes hit these breakpoints, you are once again left running individual processes to the breakpoint.

Why single stepping doesn't work

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

Why Barrier points work

If you use a barrier point, you can use the **Group** > **Go** command as many times as it takes to get all of your processes to the barrier, and you won't have to worry about a process running past the barrier.

The Root Window shows you which processes have hit the barrier. It marks all held processes with the letter **H** (meaning hold) in the column immedi-

ately to the right of the state codes. When all processes reach the barrier, TotalView removes all holds.

Holding Problems

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

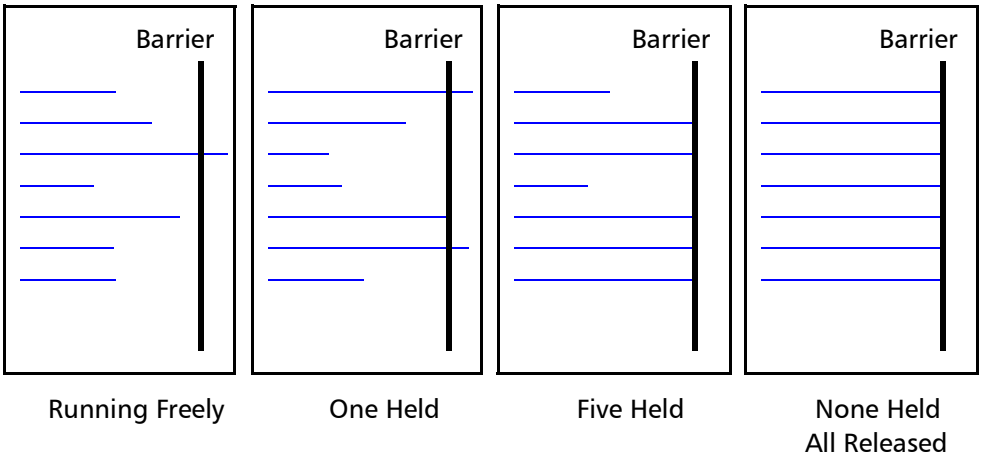
When a process is *held*, it ignores commands that tell it to execute. This means, for example, that you can't tell it to go or to step. If, for some reason, you want the process to execute, you can manually release it using either the **Group > Release** or **Process > Release Threads** command.

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

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

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

Figure 132: Running To Barriers



Examining Groups

When you debug a multiprocess program, TotalView adds processes to both a control and a share group as the process starts. These groups are not related to either UNIX process groups or PVM groups. (See Chapter 2, *"About Threads, Processes, and Groups,"* on page 15 for information on groups.)

Because a program can have more than one control group and more than one share group, TotalView decides where to place a process based on the

type of system call—which can either be **fork()** or **execve()**—that created or changed the process. The two types of process groups are:

Control Group The parent process and all related processes. A control group includes children that a process forks (processes that share the same source code as the parent). It also includes forked children that subsequently call a function such as **execve()**. That is, a control group can contain processes that don't share the same source code as the parent.

Control groups also include processes created in parallel programming disciplines like MPI.

Share Group The set of processes in a control group that shares the same source code. Members of the same share group share action points.

See Chapter 11, "Using Groups, Processes, and Threads," on page 205 for a complete discussion of groups.

TotalView automatically creates share groups when your processes fork children that call the **execve()** function, or when your program creates processes that use the same code as some parallel programming models such as MPI do.

TotalView names processes according to the name of the source program, using the following naming rules:

- TotalView names the parent process after the source program.
- The name for forked child processes differs from the parent in that TotalView appends a numeric suffix (.*n*). If you're running an MPI program, the numeric suffix is the process's rank in **COMM_WORLD**.
- If a child process calls the **execve()** function after it is forked, TotalView places a new executable name in angle brackets (<>).

In the following figure, assume that the **generate** process doesn't fork any children, and that the **filter** process forks two child processes. Later, the first child forks another child, and then calls the **execve()** function to execute the **expr** program. In this figure, the middle column shows the names that TotalView uses.

Figure 133: Control and Share Groups Example

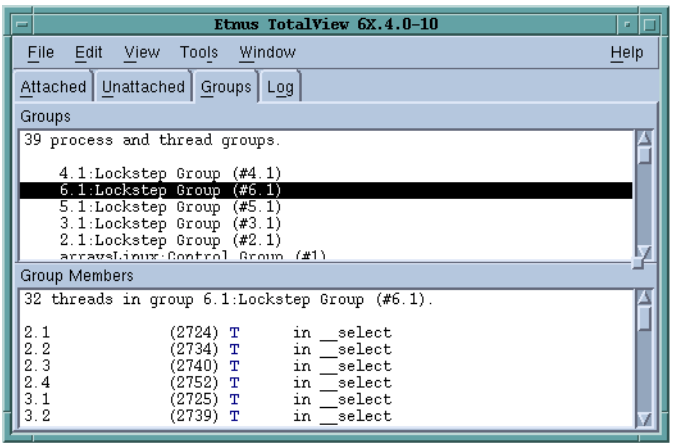
Process Groups		Process Names	Relationship
Control Group 1	Share Group 1	filter	parent process #1
		filter.1	child process #1
		filter.2	child process #2
	Share Group 2	filter<expr>.1.1	grandchild process #1
Control Group 2	Share Group 3	generate	parent process #2

Displaying Groups

To display a list of process and thread groups, select the Root Window Groups tab. Here is an example:

CLI: dptsets

Figure 134: Root Window: Groups Page



After you select a group in the top pane, TotalView displays members of the group in the bottom

When you select a group in the top list pane, TotalView updates the bottom pane to show the group's members. After TotalView updates the bottom pane, you can dive into anything shown there.

Placing Processes in Groups

TotalView uses your executable's name to determine the share group that the program belongs to. If the path names are identical, TotalView assumes that they are the same program. If the path names differ, TotalView assumes that they are different, even if the file name in the path name is the same, and places them in different share groups.

Starting Processes and Threads

To start a process, select a **Go** command from the **Group**, **Process**, or **Thread** pulldown menus.

After you select a **Go** command, TotalView decides what to run based on the current thread. It uses this thread, which is called the Thread of Interest (TOI), to decide what other threads it should run. For example, if you select **Group** > **Workers** > **Go**, TotalView continues all threads in the workers group that are associated with this thread.

CLI: dfocus g dgo
Abbreviation: G

The commands you will use most often are **Group** > **Go** and **Process** > **Go**. The **Group** > **Go** command creates and starts the current process and all other processes in the multiprocess program. There are some limitations, however. TotalView only resumes a process if the following are true:

- The process is not being held.
- The process is already exists and is stopped.
- The process is at a breakpoint.

Using a **Group** > **Go** command on a process that's already running starts the other members of the process's control group.

CLI: dgo

If the process hasn't yet been created, **Go** commands creates and starts it. *Starting* a process means that all threads in the process resume executing unless you are individually holding a thread.

TotalView disables the **Thread** > **Go** *command if asynchronous thread control is not available. If you enter a thread-level command in the* CLI *when asynchronous thread controls aren't available, TotalView tries to perform an equivalent action. For example, it continues a process instead of a thread.*

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

Creating a Process Without Starting It

The **Process** > **Create** command creates a process and stops it before the first statement in your program executes. If you link a program with shared libraries, TotalView allows the dynamic loader to map into these libraries. Creating a process without starting it is useful when you need to do the following:

- Create watchpoints or change the values of global variables after a process is created, but before it runs.
- Debug C++ static constructor code.

CLI: dstepi

While there is no CLI equivalent to the Process > Create command, executing the dstepi command produces the same effect.

Creating a Process by Single-Stepping

The TotalView single-stepping commands lets you create a process and run it to the beginning of your programs. The single-stepping commands available from the **Process** menu are as shown in the following table:

GUI command	CLI command	Creates the process and ...
Process > Step	dfocus p dstep	Runs it to the first line of the **main()** routine.
Process > Next	dfocus p dnext	Runs it to the first line of the **main()** routine; this is the same as **Process** > **Step**.
Process > Step Instruction	dfocus p dstepi	Stops it before any of your program executes.

GUI command	CLI command	Creates the process and ...
Process > Next Instruction	dfocus p dnexti	Runs it to the first line of the main() routine. This is the same as Process > Step.
Process > Run To	dfocus p duntil	Runs it to the line or instruction selected in the Process Window.

If a group-level or thread-level stepping command creates a process, the behavior is the same as if it were a process-level command.

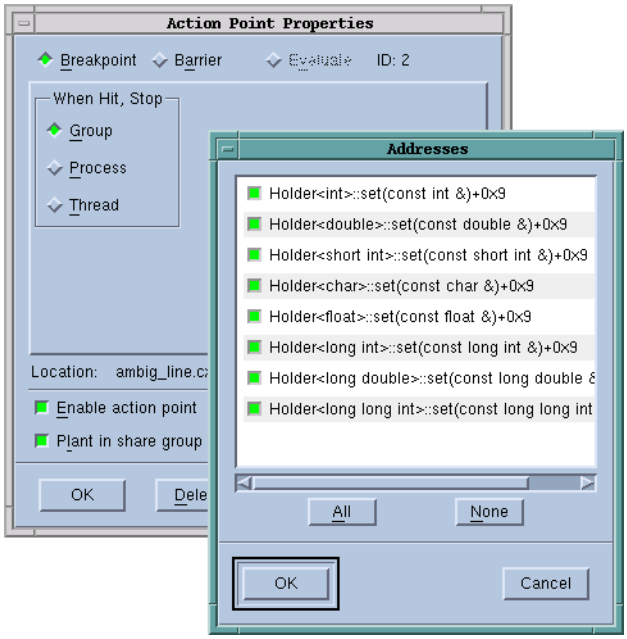
Chapter 11, "Using Groups, Processes, and Threads," on page 205 contains a detailed discussion of setting the focus for stepping commands.

Stepping and Setting Breakpoints

GUI

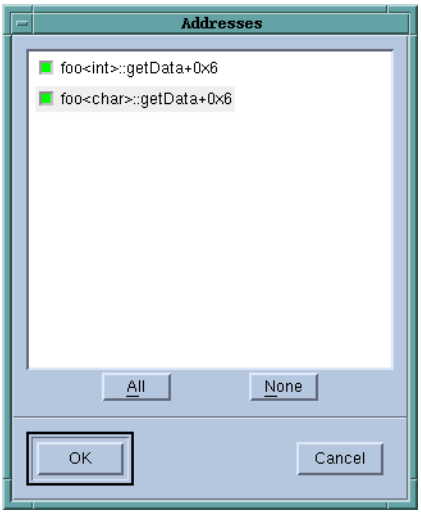
Several of the single-stepping commands require that you select a source line or machine instruction in the Source Pane. To select a source line, place the cursor over the line and click your left mouse button. If you select a source line that has more than one instantiation, TotalView will try to do the right thing. For example, if you select a line within a template so you can set a breakpoint on it, you'll actually set a breakpoint on all of the template's instantiations. If this isn't what you want, select the **Location** button in the **Action Point > Properties** Dialog Box to change which instantiations will have a breakpoint. (See "*Setting Breakpoints and Barriers*" on page 295.)

Figure 135: Action Point and Addresses Dialog Boxes



If TotalView cannot figure out which instantiation to set a breakpoint at, it will display its **Address** Dialog Box.

Figure 136: Ambiguous Address Dialog Box



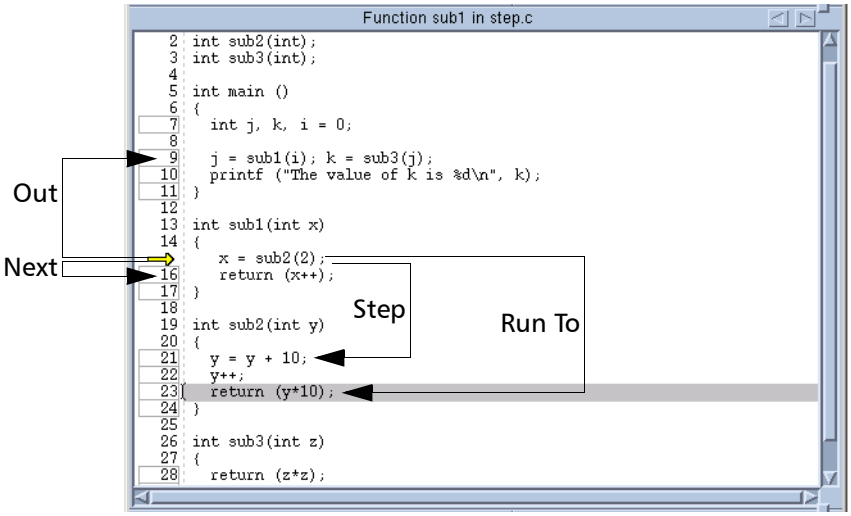
Using Stepping Commands

While different programs have different requirements, the most common stepping mode is to set group focus to **Control** and the target to **Process** or **Group**. You can now select stepping commands from the **Process** or **Group** menus or use commands in the toolbar.

CLI: dfocus g
dfocus p

The following figure graphically illustrates the different types of stepping commands.

Figure 137: Stepping Illustrated



The arrow indicates that the PC is at line 14. The four stepping commands do the following:

- **Next** executes line 14. After stepping, the PC is at line 15.
- **Step** moves into the **sub2()** function. The PC is at line 20.
- **Run To** executes all lines until the PC reaches the selected line, which is line 22.
- **Out** executes all statements within **sub1()** and exits from the function. The PC is at line 8. If you now execute a Step command, TotalView steps into **sub3()**.

Remember the following things about single-stepping commands:

- To cancel a single-step command, select **Group** > **Halt** or **Process** > **Halt**.

CLI:	dhalt

- If your program reaches a breakpoint while stepping over a function, TotalView cancels the operation and your program stops at the breakpoint.
- If you enter a source-line stepping command and the primary thread is executing in a function that has no source-line information, TotalView performs an assembler-level stepping command.

When TotalView steps through your code, it steps one line at-a-time. This means that if you have more than one statement on a line, a step instruction executes all of the instructions on that line.

Stepping into Function Calls

The stepping commands execute one line in your program. If you are using the CLI, you can use a numeric argument that indicates how many source lines TotalView steps. For example, here's the CLI instruction for stepping three lines:

```
dstep 3
```

If the source line or instruction contains a function call, TotalView steps into it. If TotalView can't find the source code and the function was compiled with **–g**, it displays the function's machine instructions.

You might not realize that your program is calling a function. For example, if you overloaded an operator, you'll step into the code that defines the overloaded operator.

If the function being stepped into wasn't compiled with the –g command-line option, TotalView always steps over the function.

The TotalView GUI has eight **Step** commands and eight **Step Instruction** commands. These commands are located on the **Group**, **Process**, and **Thread** pulldowns. The difference between them is the focus.

CLI:	dfocus ... dstep
	dfocus ... dstepi

Stepping Over Function Calls

When you step over a function, TotalView stops execution when the primary thread returns from the function and reaches the source line or instruction after the function call.

The TotalView GUI has eight **Next** commands that execute a single source line while stepping over functions, and eight **Next Instruction** commands that execute a single machine instruction while stepping over functions. These commands are on the **Group**, **Process**, and **Thread** menus.

CLI: dfocus ... dnext
dfocus ... dnexti

If the PC is in assembler code—this can happen, for example, if you halt your program while it's executing in a library—a **Next** operation executes the next instruction. If you want to execute out of the assembler code so your back in your code, select the **Out** command. You might need to select **Out** a couple of times until your back to where you want to be.

Executing to a Selected Line

If you don't need to stop execution every time execution reaches a specific line, you can tell TotalView to run your program to a selected line or machine instruction. After selecting the line on which you want the program to stop, invoke one of the eight **Run To** commands defined within the TotalView GUI. These commands are on the **Group**, **Process**, and **Thread** menus.

CLI: dfocus ... duntil

Executing to a selected line is discussed in greater depth in Chapter 11, "*Using Groups, Processes, and Threads*," on page 205.

If your program reaches a breakpoint while running to a selected line, TotalView stops at that breakpoint.

If your program calls recursive functions, you can select a nested stack frame in the Stack Trace Pane. When you do this, TotalView determines where to stop execution by looking at the following:

- The frame pointer (FP) of the selected stack frame.
- The selected source line or instruction.

CLI: dup and ddown

Executing Out of a Function

You can step your program out of a function by using the **Out** commands. The eight **Out** commands in the TotalView GUI are located on the **Group**, **Process**, and **Thread** menus.

CLI: dfocus ... dout

If the source line that is the *goal* of the **Out** operation has more than one statement, TotalView will stop execution just after the routine from which it just emerged. For example, suppose that the following is your source line:

```
routine1; routine2;
```

Suppose you step into **routine1**, then use an **Out** command. While the PC arrow in the Source Pane still points to this same source line, the actual PC is just after **routine1**. This means that if you use a step command, you will step into **routine2**.

The PC arrow does not move when the source line only has one statement on it. The internal PC does, of course, change.

GUI

You can also return out of several functions at once, by selecting the routine in the Stack Trace Pane that you want to go to, and then selecting an **Out** command.

If your program calls recursive functions, you can select a nested stack frame in the Stack Trace Pane to indicate which instance you are running out of.

GUI

Continuing with a Specific Signal

Letting your program continue after sending it a signal is useful when your program contains a signal handler. Here's how you tell TotalView to do this:

1 Select the Process Window's **Thread** > **Continuation Signal** command.

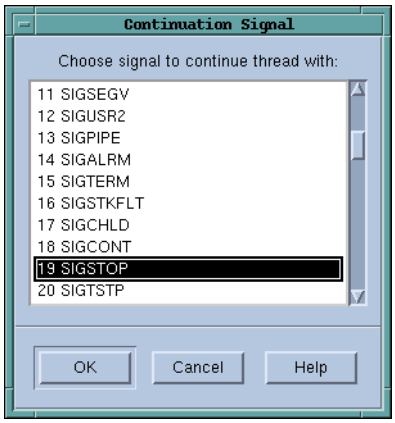


Figure 138: Thread > Continuation Signal Dialog Box

2 Select the signal to be sent to the thread and then select **OK**.

The continuation signal is set for the thread contained in the current Process Window. If the operating system can deliver multithreaded signals, you can set a separate continuation signal for each thread. If it can't, this command clears continuation signals set for other threads in the process.

3 Continue execution of your program with commands such as **Process** > **Go**, **Step**, **Next**, or **Detach**.

TotalView continues the threads and sends the specified signals to your process.

You can clear the continuation signal by selecting **signal 0** *from this dialog box.*

You can change the way TotalView handles a signal by setting the **TV::signal_handling_mode** variable in a **.tvdrc** startup file. For more information, see Chapter 4 of the "TotalView Reference Guide."

Deleting (Killing) Programs

To delete (or kill) all the processes in a control group, use the **Group** > **Delete** command. The next time you start the program, for example, by using the **Process** > **Go** command, TotalView creates and starts a fresh master process.

CLI: **dfocus g dkill**

Restarting Programs

You can use the **Group** > **Restart** command to restart a program that is running or one that is stopped but hasn't exited.

CLI: **drerun**

If the process is part of a multiprocess program, TotalView deletes all related processes, restarts the master process, and runs the newly created program.

The **Group** > **Restart** command is equivalent to the **Group** > **Delete** command followed by the **Process** > **Go** command.

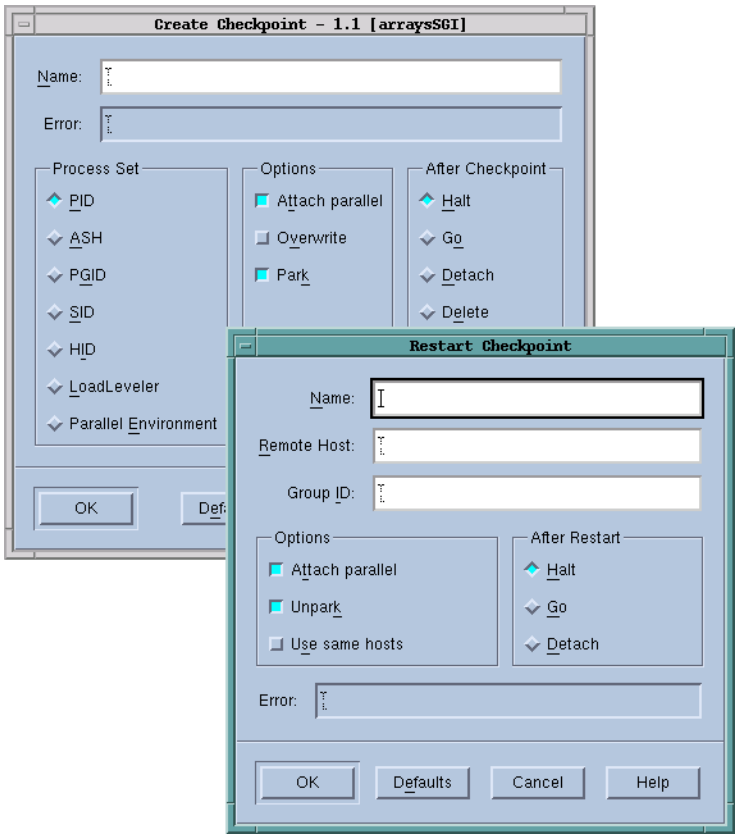
Checkpointing

On SGI IRIX and IBM RS/6000 platforms, you can save the state of selected processes and then use this saved information to restart the processes from the position where they were saved. For more information, see the

Process Window **Tools** > **Create Checkpoint** and **Tools** > **Restart Checkpoint** commands in the online Help.

CLI: dcheckpoint
drestart

Figure 139: Create Checkpoint and Restart Checkpoint Dialog Boxes



Fine-Tuning Shared Library Use

When TotalView encounters a reference to a shared library, it normally reads all of that library's symbols. In some cases, you might need to explicitly read in this library's information before TotalView automatically reads it.

On the other hand, you may not want TotalView to read and process a library's loader and debugging symbols. In most cases, reading these symbols occurs quickly. However, if your program uses large libraries, you can increase performance by telling TotalView not to read these symbols.

For more information, see "*Preloading Shared Libraries*" on page 198 and "*Controlling Which Symbols TotalView Reads*" on page 200.

Preloading Shared Libraries

As your program executes, it can call the **dlopen()** function to access code contained in shared libraries. In some cases, you might need to do something from within TotalView that requires you to preload library information.

For example, you might need to refer to one of a library's functions in an eval point or in a **Tools > Evaluate** command. If you use the function's name before TotalView reads the dynamic library, TotalView displays an error message.

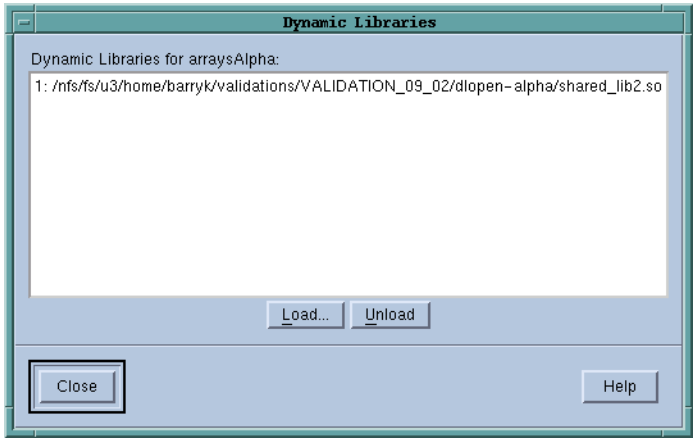
Use the **Tools > Debugger Loaded Libraries** command to tell TotalView to open a library.

CLI: ddlopen

This CLI command gives you additional ways to control how a library's symbols are used.

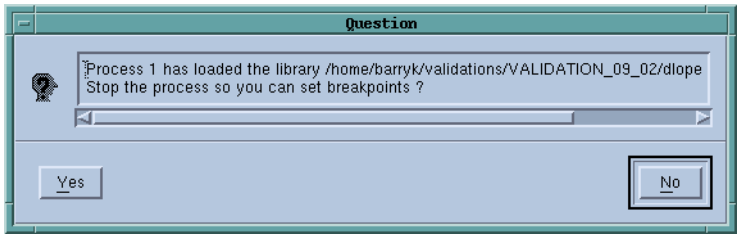
After selecting this command, TotalView displays the following dialog box:

Figure 140: Tools > Debugger Loaded Libraries Dialog Box



Selecting the **Load** button tells TotalView to display a file explorer dialog box that lets you navigate through your computer's file system to locate the library. After selecting a library, TotalView reads it and displays a question box that lets you stop execution to set a breakpoint:

Figure 141: Stopping to Set a Breakpoint Question Box



TotalView might not read in information symbol and debugging information when you use this command. See "Controlling Which Symbols TotalView Reads" on page 200 for more information.

Controlling Which Symbols TotalView Reads

When debugging large programs with large libraries, reading and parsing symbols can impact performance. This section describes how you can minimize the impact that reading this information has on your debugging session.

Using the preference settings and variables described in this section always slow down performance. However, for most programs, even large ones, the difference is often inconsequential. If, however, you are debugging a very large program with large libraries, significant performance improvements can occur.

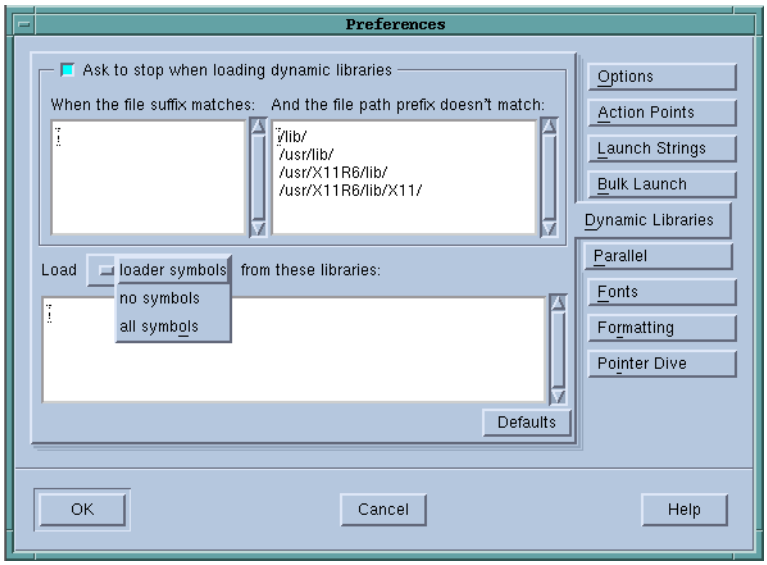
A shared library contains, among other things, loader and debugging symbols. Typically, loader symbols are read quite quickly. Debugging symbols can require considerable processing. The default behavior is to read all symbols. You can change this behavior by telling TotalView to only read in loader symbols or even that it should not read in any symbols.

Saying "TotalView reads all symbols" isn't quite true as TotalView often just reads in loader symbols for some libraries. For example, it only reads in loader symbols if the library resides in the /usr/lib *directory. (These libraries are typically those provided with the operating system.) You can override this behavior by adding a library name to the* All Symbols *list that is described in the next section.*

Specifying Which Libraries are Read

After invoking the **File > Preferences** command, select the Dynamic Libraries Page.

Figure 142: File > Preferences: Dynamic Libraries Page



The lower portion of this page lets you enter the names of libraries for which you need to manage the information that TotalView reads.

When you enter a library name, you can use the * (asterisk) and ? (question mark) wildcard characters. These characters have their standard meaning. Placing entries into these areas does the following:

all symbols	This is the default operation. You only need to enter a library name here if it would be excluded by a wildcard in the **loader symbols** and **no symbols** areas.
loader symbols	TotalView reads loader symbols from these libraries. If your program uses a number of large shared libraries that you will not be debugging, you might set this to asterisk (*). You then enter the names of DLLs that you need to debug in the **all symbols** area.
no symbols	Normally, you wouldn't put anything on this list since TotalView might not be able to create a backtrace through a library if it doesn't have these symbols. However, you can increase performance if you place the names of your largest libraries here.

When reading a library, TotalView looks at these lists in the following order:

1. all symbols
2. loader symbols
3. no symbols

If a library is found in more than one area, it does the first thing it is told to do and ignores any other requests. For example, after TotalView reads a library's symbols, it cannot honor a request to not load in symbols, so it ignores a request to not read them.

CLI:
```
dset TV::dll_read_all_symbols
dset TV::dll_read_loader_symbols_only
dset TV::dll_read_no_symbols
```

See the online Help for additional information.

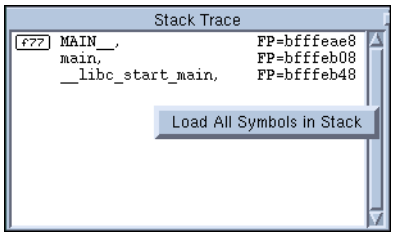
If your program stops in a library that has not already had its symbols read, TotalView reads the library's symbols. For example, if your program SEGVs in a library, TotalView reads the symbols from that library before it reports the error. In all cases, however, TotalView always reads the loader symbols for shared system libraries.

Reading Excluded Information

While you are debugging your program, you might find that you do need the symbol information that you told TotalView that it shouldn't read. Tell

TotalView to read them by right-clicking your mouse in the Stack Trace Pane and then selecting the **Load All Symbols in Stack** command from the context menu.

Figure 143: Load All Symbols in Stack Context menu



After selecting this command, TotalView examines all active stack frames and, if it finds unread libraries in any frame, TotalView reads them.

CLI: TV::read_symbols
This CLI command also gives you finer control over how TotalView reads in library information.

Setting the Program Counter

TotalView lets you resume execution at a different statement than the one at which it stopped execution by resetting 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 is in an error state.

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

In TotalView, you can set the PC of a stopped thread to a selected source line or a selected instruction. When you set the PC to a selected line, the PC points to the memory location where the statement begins. For most situations, setting the PC to a selected line of source code is all you need to do.

To set the PC to a selected line:

1. If you need to set the PC to a location somewhere in a line of source code, select the **View > Source As > Both** command.
   TotalView responds by displaying assembler code.
2. Select the source line or instruction in the Source Pane.
   TotalView highlights the line.

3 Select the **Thread** > **Set PC** command.

TotalView asks for confirmation, resets the PC, and moves the PC symbol to the selected line.

When you select a line and ask TotalView 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, TotalView asks if it can unwind the stack:

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

If you select **Yes**, TotalView discards deeper stack frames (that is, all stack frames that are more deeply nested than the selected stack frame) and resets the machine registers to their values for the selected frame. If you select **No**, TotalView sets the PC to the selected line, but it leaves the stack and registers in their current state. Since you can't assume that the stack and registers have the right values, selecting **No** is almost always the wrong thing to do.

Interpreting the Status and Control Registers

The Stack Frame Pane in the Process Window lists the contents of CPU registers for the selected frame—you might need to scroll past the stack local variables to see them.

CLI: **dprint** *register*

You must quote the initial $ character in the register name; for example, dprint \$r1.

For your convenience, TotalView displays the bit settings of many CPU registers symbolically. For example, TotalView symbolically displays registers that control rounding and exception enable modes. You can edit the values of these registers and then resume program execution. For example, you might do this to examine the behavior of your program with a different rounding mode.

Since the registers that are displayed vary from platform to platform, see "*Architectures*" in the *TotalView Reference Guide* for information on how TotalView displays this information on your CPU. For general information on editing the value of variables (including registers), see "*Displaying Areas of Memory*" on page 246. To learn about the meaning of these registers, you see the documentation for your CPU.

Using Groups, Processes, and Threads

11

The specifics of how multiprocess, multithreaded programs execute differ greatly from platform to platform and environment to environment, but all share some general characteristics. This chapter discusses the TotalView process/thread model. It also describes the way in which you tell the GUI and the CLI what processes and threads to direct a command to.

This chapter contains the following sections:

- "*Defining the* GOI, POI, *and* TOI" on page 205
- "*Setting a Breakpoint*" on page 206
- "*Stepping* (*Part* I)" on page 207
- "*Using* P/T *Set Controls*" on page 210
- "*Setting Process and Thread Focus*" on page 211
- "*Setting Group Focus*" on page 216
- "*Stepping* (*Part* II): *Examples*" on page 227
- "*Using* P/T *Set Operators*" on page 228
- "*Using the* P/T *Set Browser*" on page 229
- "*Using the Group Editor*" on page 231

Defining the GOI, POI, and TOI

This chapter consistently uses the following three related acronyms:

- GOI—Group of Interest
- POI—Process of Interest
- TOI—Thread of Interest

These terms are important in the TotalView process/thread model because TotalView must determine the scope of what it does when it executes a command. For example, Chapter 2, "*About Threads, Processes, and Groups*" introduced the types of groups contained with TotalView. That chapter ignored what happens when you execute a TotalView command on a group.

For example, what does "stepping a group" actually mean? What happens to processes and threads that aren't in this group?

Associated with these three terms is a fourth term: *arena*. The *arena* is the collection of processes, threads, and groups that are affected by a debugging command. This collection is called an *arena list*.

In the GUI, the arena is most often set using the pulldown list in the toolbar. You can set the arena using commands in the menubar. For example, there are eight *next* commands. The difference between them is the arena; that is, the difference between the *next* commands is the processes and threads that are the target of what the *next* command runs.

When TotalView executes any action command, the arena decides the scope of what can run. It doesn't, however, determine what does run. Depending on the command, TotalView determines the TOI, POI, or GOI, and then executes the command's action on that thread, process, or group. For example, suppose TotalView steps the current control group:

- TotalView needs to know what the TOI is so that it can determine what threads are in the lockstep group—TotalView only lets you step a lockstep group.
- The lockstep group is part of a share group.
- This share group is also contained in a control group.

By knowing what the TOI is, the TotalView GUI also knows what the GOI is. This is important because, while TotalView knows what it will step (the threads in the lockstep group), it also knows what it will allow to run freely while it is stepping these threads. In the CLI, the P/T set determines the TOI.

Setting a Breakpoint

You can set breakpoints in your program by selecting the boxed line numbers in the Source Code pane of a Process window. A boxed line number indicates that the line generates executable code. A STOP icon masking a line number indicates that a breakpoint is set on the line. Selecting the STOP icon clears the breakpoint.

When a program reaches a breakpoint, it stops. You can let the program resume execution in any of the following ways:

- Use the single-step commands described in "*Using Stepping Commands*" on page 193.
- Use the set program counter command to resume program execution at a specific source line, machine instruction, or absolute hexadecimal value. See "*Setting the Program Counter*" on page 202.
- Set breakpoints at lines you choose, and let your program execute to that breakpoint. See "*Setting Breakpoints and Barriers*" on page 295.
- Set conditional breakpoints that cause a program to stop after it evaluates a condition that you define, for example "stop when a value is less than eight. See "*Setting Eval Points*" on page 308.

TotalView provides additional features for working with breakpoints, process barrier breakpoints, and eval points. For more information, see Chapter 14, "*Setting Action Points*," on page 293.

Stepping (Part I)

You can use TotalView stepping commands to:

- Execute one source line or machine instruction at a time; for example, **Process** > **Step** in the GUI and **dstep** in the CLI.

  CLI: **dstep**

- Run to a selected line, which acts like a temporary breakpoint; for example, **Process** > **Run To**.

  CLI: **duntil**

- Run until a function call returns; for example, **Process** > **Out**.

  CLI: **dout**

In all cases, stepping commands operate on the Thread of Interest (TOI). In the GUI, the TOI is the selected thread in the current Process Window. In the CLI, the TOI is the thread that TotalView uses to determine the scope of the stepping operation.

On all platforms except SPARC Solaris, TotalView uses *smart* single-stepping to speed up stepping of one-line statements that contain loops and conditions, such as Fortran 90 array assignment statements. *Smart stepping* occurs when TotalView realizes that it doesn't need to step through an instruction. For example, assume that you have the following statements:

```
integer iarray (1000,1000,1000)
iarray = 0
```

These two statements define one billion scalar assignments. If your computer steps every instruction, you will probably never get past this statement. *Smart stepping* means that TotalView single-steps through the assignment statement at a speed that is very close to your computer's native speed.

Other topics in this section are:

- "*Understanding Group Widths*" on page 208
- "*Understanding Process Width*" on page 208
- "*Understanding Thread Width*" on page 208
- "*Using Run To and duntil Commands*" on page 209

Understanding Group Widths

TotalView behavior when stepping at group width depends on whether the Group of Interest (GOI) is a process group or a thread group. In the following lists, *goal* means the place at which things should stop executing. For example, if you selected a *step* command, the goal is the next line. If you selected a *run to* command, the goal is the selected line.

The actions that TotalView performs on the GOI are dependent on the type of process group that is the focus, as follows:

- *Process group*—TotalView examines the group, and identifies which of its processes has a thread stopped at the same location as the TOI (a *matching* process). TotalView runs these matching processes until one of its threads arrives at the goal. When this happens, TotalView stops the thread's process. The command finishes when it has stopped all of these *matching* processes.
- *Thread group*—TotalView runs all processes in the control group. However, as each thread arrives at the goal, TotalView only stops that thread; the rest of the threads in the same process continue executing. The command finishes when all threads in the GOI arrive at the goal. When the command finishes, TotalView stops all processes in the control group.

  TotalView doesn't wait for threads that are not in the same share group as the TOI, since they are executing different code and can never arrive at the goal.

Understanding Process Width

TotalView behavior when stepping at process width (which is the default) depends on whether the Group of Interest (GOI) is a process group or a thread group.

The actions that TotalView performs on the GOI are dependent on the type of process group that is the focus, as follows:

- *Process group*—TotalView runs all threads in the process, and execution continues until the TOI arrives at its goal, which can be the next statement, the next instruction, and so on. Only when the TOI reaches the goal does TotalView stop the other threads in the process.
- *Thread group*—TotalView lets all threads in the GOI and all manager threads run. As each member of the GOI arrives at the goal, TotalView stops it; the rest of the threads continue executing. The command finishes when all members of the GOI arrive at the goal. At that point, TotalView stops the whole process.

Understanding Thread Width

When TotalView performs a stepping command, it decides what it steps based on the *width*. Using the toolbar, you specify width using the left-most pulldown. This pulldown has three items: **Group**, **Process**, and **Thread**.

Stepping at thread width tells TotalView to only run that thread. It does not step other threads. In contrast, process width tells TotalView to run all threads in the process that are allowed to run while the TOI is stepped. While TotalView is stepping the thread, manager threads run freely.

Stepping a thread isn't the same as stepping a thread's process, because a process can have more than one thread.

Thread-stepping is not implemented on Sun platforms. On SGI platforms, thread-stepping is not available with pthread programs. If, however, your program's parallelism is based on SGI's sprocs, thread-stepping is available.

Thread-level single-step operations can fail to complete if the TOI needs to synchronize with a thread that isn't running. For example, if the TOI requires a lock that another held thread owns, and steps over a call that tries to acquire the lock, the primary thread can't continue successfully. You must allow the other thread to run in order to release the lock. In this case, you use process-width stepping instead.

Using Run To and duntil Commands

The **duntil** and Run To commands differ from other step commands when you apply them to a process group. (These commands tell TotalView to execute program statements *until* a selected statement is reached.) When applied to a process group, TotalView identifies all processes in the group that already have a thread stopped at the goal. These are the *matching* processes. TotalView then runs only the nonmatching processes. Whenever a thread arrives at the goal, TotalView stops its process. The command finishes when it has stopped all members of the group. This lets you synchronize all the processes in a group in preparation for group-stepping them.

You need to know the following if you're running at process width:

Process group If the Thread of Interest (TOI) is already at the goal location, TotalView steps the TOI past the line before the process runs. This lets you use the Run To command repeatedly in loops.

Thread group If any thread in the process is already at the goal, TotalView temporarily holds it while other threads in the process run. After all threads in the thread group reach the goal, TotalView stops the process. This lets you synchronize the threads in the POI at a source line.

If you're running at group width:

Process group TotalView examines each process in the process and share group to determine whether at least one thread is already at the goal. If a thread is at the goal, TotalView holds its process. Other processes are allowed to run. When at least one thread from each of these processes is held, the command completes. This lets you synchronize at least one thread in each of these processes at a source line. If you're running a control group, this synchronizes all processes in the share group.

Thread group TotalView examines all the threads in the thread group that are in the same share group as the TOI to determine whether a thread is already at the goal. If it is, TotalView holds it. Other threads are allowed to run. When all of the threads in the TOI's share group reach the goal, TotalView stops the TOI's *control* group and the

command completes. This lets you synchronize thread group members. If you're running a workers group, this synchronizes all worker threads in the share group.

The process stops when the TOI and at least one thread from each process in the group or process being run reach the command stopping point. This lets you synchronize a group of processes and bring them to one location.

You can also run to a selected line in a nested stack frame, as follows:

1. Select a nested frame in the Stack Trace Pane.
2. Select a source line or instruction in the function.
3. Enter a **Run To** command.

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

Using P/T Set Controls

A few TotalView windows have P/T set control elements. For example, the following figure shows the top portion of the Process Window.

Figure 144: The P/T Set Control in the Process Window

This pulldown menu differs from the P/T set controls located on some dialog boxes. On dialog boxes, you will see two pulldowns. However, in the context of the Process Window, elements from the two pulldowns have combined both to eliminate actions that don't have meaning. When you select a group and a modifier, you are telling TotalView that when you press one of the remaining buttons on the toolbar, this element names the focus on which TotalView acts. For example, if **Thread** is selected and you select **Step**, TotalView steps the current thread. If **Process (workers)** is selected and you select **Halt**, TotalView halts all processes associated with the current threads workers group. If you were running a multiprocess program, other processes continue to execute.

Other windows have similar controls; for example:

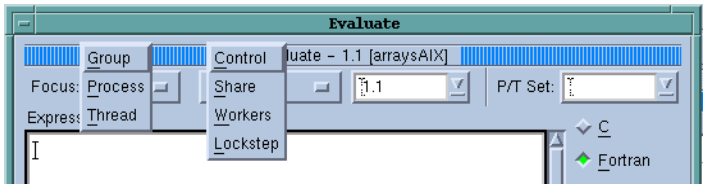


Figure 145: The P/T Set Control in the Tools > Evaluate Window

The first pulldown menu, which is called the *Width Pulldown*, has three elements on it: **Group**, **Process**, and **Thread**. Your choices indicate the width of the command. For example, if **Group** is selected, a **Go** command continues the group. Which group TotalView continues is set by the choices on the second pulldown menu. The *Width Pulldown* tells TotalView where to look when it tries to determine what it manipulates. The second pulldown, which is called the *Scope Pulldown*, tells TotalView which processes and threads in the scope defined by the *Width Pulldown* to manipulate. For example, you can tell TotalView to step the threads defined in the current workers group that are contained in the current process.

Finally, the *Thread Selector* (the third pulldown menu from the left) lets you change the focus of the action from the currently defined process and threads to any other process and thread that TotalView controls. That is, this changes the POI and TOI.

The **P/T Set** box on the right lets you directly enter a P/T set expression. The other P/T set controls modify the focus of what you enter.

What is selected can be quite complicated when you use the GUI to set these controls, or when you specify a focus using the CLI.

Setting Process and Thread Focus

The previous sections emphasize the GUI; *this section and the ones that follow emphasize the* CLI. *Here you will find information on how to have full asynchronous debugging control over your program. Fortunately, having this level of control is seldom necessary. In other words, don't read the rest of this chapter unless you have to.*

When TotalView executes a command, it must decide which processes and threads to act on. Most commands have a default set of threads and processes and, in most cases, you won't want to change the default. In the GUI, the default is the process and thread in the current Process Window. In the CLI, this default is indicated by the focus, which is shown in the CLI prompt.

There are times, however, when you need to change this default. This section begins a rather intensive look at how you tell TotalView what processes and threads to use as the target of a command.

Topics in this section are:

- "*Understanding Process/Thread Sets*" on page 211
- "*Specifying Arenas*" on page 213
- "*Specifying Processes and Threads*" on page 213

Understanding Process/Thread Sets

All TotalView commands operate on a set of processes and threads. This set is called a *Process/Thread* (P/T) *set*. The right-hand text box in windows that contain P/T set controls lets you construct these sets. In the CLI, you specify a P/T set as an argument to a command such as **dfocus**. If you're using the GUI, TotalView creates this list for you based on which Process Window has focus.

Unlike a serial debugger in which each command clearly applies to the only executing thread, TotalView can control and monitor many threads with their PCs at many different locations. The P/T set indicates the groups, processes, and threads that are the target of the CLI command. No limitation exists on the number of groups, processes, and threads in a set.

A P/T set is a list that contains one or more P/T identifiers. (The next section, "*Specifying Arenas*" on page 213, explains what a P/T identifier is.) Tcl lets you create lists in the following ways:

- You can enter these identifiers within braces (**{ }**).
- You can use Tcl commands that create and manipulate lists.

These lists are then used as arguments to a command. If you're entering one element, you usually do not have to use the Tcl list syntax.

For example, the following list contains specifiers for process 2, thread 1, and process 3, thread 2:

```
{p2.1 p3.2}
```

If you do not explicitly specify a P/T set in the CLI, TotalView defines a target set for you. (In the GUI, the default set is determined by the current Process Window.) This set is displayed as the default CLI prompt. (For information on this prompt, see "*About* CLI *Prompt*" on page 172.)

You can change the focus on which a command acts by using the **dfocus** command. If the CLI executes the **dfocus** command as a unique command, it changes the default P/T set. For example, if the default focus is process 1, the following command changes the default focus to process 2:

```
dfocus p2
```

After TotalView executes this command, all commands that follow focus on process 2.

In the GUI, *you set the focus by displaying a Process Window that contains this process. Do this by using the* **P+** *and* **P–** *buttons on the toolbar or by making a selection in the Root Window.*

If you begin a command with **dfocus**, TotalView changes the target only for the command that follows. After the command executes, TotalView restores the *former* default. The following example shows both of these ways to use the **dfocus** command. Assume that the current focus is process 1, thread 1. The following commands change the default focus to group 2 and then step the threads in this group twice:

```
dfocus g2
dstep
dstep
```

In contrast, if the current focus is process 1, thread 1, the following commands step group 2 and then step process 1, thread 1:

```
dfocus g2 dstep
dstep
```

Some commands only operate at the process level; that is, you cannot apply them to a single thread (or group of threads) in the process, but must apply them to all or to none.

Specifying Arenas

A P/T identifier often indicates a number of groups, processes, and threads. For example, assume that two threads executing in process 2 are stopped at the same statement. This means that TotalView places the two stopped threads into lockstep groups. If the default focus is process 2, stepping this process actually steps both of these threads.

TotalView uses the term *arena* to define the processes and threads that are the target of an action. In this case, the arena has two threads. Many CLI commands can act on one or more arenas. For example, the following command has two arenas:

```
dfocus {p1 p2}
```

The two arenas are process 1 and process 2.

When there is an arena list, each arena in the list has its own GOI, POI, and TOI.

Specifying Processes and Threads

The previous sections described P/T sets as being lists; however, these discussions ignored what the individual elements of the list are. A better definition is that a P/T set is a list of arenas, where an *arena* consists of the processes, threads, and groups that are affected by a debugging command. Each *arena specifier* describes a single arena in which a command acts; the *list* is just a collection of arenas. Most commands iterate over the list, acting individually on an arena. Some CLI output commands, however, combine arenas and act on them as a single target.

An arena specifier includes a *width* and a TOI. (Widths are discussed later in this section.) In the P/T set, the TOI specifies a target thread, while the width specifies how many threads surrounding the thread of interest are affected.

Defining the Thread of Interest (TOI)

The TOI is specified as **p.t**, where **p** is the TotalView process ID (PID) and **t** is the TotalView thread ID (TID). The **p.t** combination identifies the POI (Process of Interest) and TOI. The TOI is the primary thread affected by a command. This means that it is the primary focus for a TotalView command. For example, while the **dstep** command always steps the TOI, it can run the rest of the threads in the POI and step other processes in the group.

In addition to using numerical values, you can also use two special symbols:

- The less-than character (<) indicates the *lowest numbered worker thread* in a process, and is used instead of the TID value. If, however, the arena explicitly names a thread group, the < symbol means the lowest numbered member of the thread group. This symbol lets TotalView select the first user thread, which might not be thread 1; for example, the first and only user thread might be thread number 3 on HP Alpha systems.

- A dot (.) indicates the current set. Although you seldom use this symbol interactively, it can be useful in scripts.

About Process and Thread Widths

You can enter a P/T set in two ways. If you're not manipulating groups, the format is as follows:

[*width_letter*][*pid*][.*tid*]

"Specifying Groups in P/T Sets" on page 217 extends this format to include groups. When using P/T sets, you can create sets with just width indicators or just group indicators, or both.

For example, **p2.3** indicates process 2, thread 3.

Although the syntax seems to indicate that you do not need to enter any element, TotalView requires that you enter at least one. Because TotalView tries to determine what it can do based on what you type, it tries to fill in what you omit. The only requirement is that when you use more than one element, you use them in the order shown here.

You can leave out parts of the P/T set if what you do enter is unambiguous. A missing width or PID is filled in from the current focus. A missing TID is always assumed to be <. For more information, see "*Naming Incomplete Arenas*" on page 225.

The *width_letter* indicates which processes and threads are part of the arena. You can use the following letters:

t *Thread width*

A command's target is the indicated thread.

p *Process width*

A command's target is the process that contains the TOI.

g *Group width*

A command's target is the group that contains the POI. This indicates control and share groups.

a *All processes*

A command's target is all threads in the GOI that are in the POI.

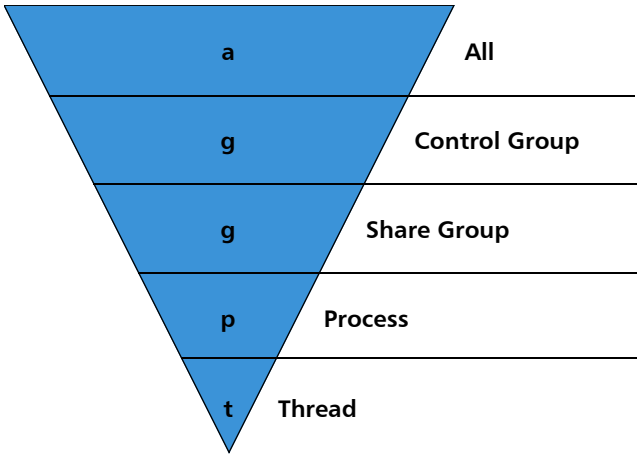
d *Default width*

A command's target depends on the default for each command. This is also the width to which the default focus is set. For example, the **dstep** command defaults to process width (run the process while stepping one thread), and the **dwhere** command defaults to thread width.

You must use lowercase letters to enter these widths.

The following figure illustrates how these specifiers relate to one another.

Figure 146: Width Specifiers



The **g** specifier indicates control and share groups. This inverted triangle indicates that the arena focuses on a greater number of entities as you move from **Thread** level at the bottom to **All** level at the top.

As mentioned previously, the TOI specifies a target thread, while the width specifies how many threads surrounding the TOI are also affected. For example, the **dstep** command always requires a TOI, but entering this command can do the following:

- Step just the TOI during the step operation (thread-level single-step).
- Step the TOI and step all threads in the process that contain the TOI (process-level single-step).
- Step all processes in the group that have threads at the same PC as the TOI (group-level single-step).

This list doesn't indicate what happens to other threads in your program when TotalView steps your thread. For more information, see "*Stepping* (*Part* II): *Examples*" on page 227.

To save a P/T set definition for later use, assign the specifiers to a Tcl variable; for example:

```
set myset {g2.3 t3.1}
dfocus $myset dgo
```

As the **dfocus** command returns its focus set, you can save this value for later use; for example:

```
set save_set [dfocus]
```

Specifier Examples

The following are some sample specifiers:

g2.3	Select process 2, thread 3, and set the width to *group*.
t1.7	Commands act only on thread 7 or process 1.
d1.<	Use the default set for each command, focusing on the first user thread in process 1. The less-than symbol (<) sets the TID to the first user thread.

Setting Group Focus

TotalView has two types of groups: process groups and thread groups. Process groups only contain processes, and thread groups only contain threads. The threads in a thread group can be drawn from more than one process.

Topics in this section are:

- "*Specifying Groups in P/T Sets*" on page 217
- "*About Arena Specifier Combinations*" on page 218
- "*'All' Does Not Always Mean 'All'*" on page 220
- "*Setting Groups*" on page 222
- "*Using the g Specifier: An Extended Example*" on page 222
- "*Merging Focuses*" on page 225
- "*Naming Incomplete Arenas*" on page 225
- "*Naming Lists with Inconsistent Widths*" on page 226

TotalView has four predefined groups. Two of these only contain processes, while the other two only contain threads. TotalView also lets you create your own groups, and these groups can have elements that are processes and threads. The following are the predefined process groups:

- **Control Group**

  Contains the parent process and all related processes. A control group includes children that were forked (processes that share the same source code as the parent) and children that were forked but subsequently called the **execve()** function.

  Assigning a new value to the **CGROUP (dpid)** variable for a process changes that process's control group. In addition, the **dgroups –add** command lets you add members to a group in the CLI. In the GUI, you use the **Group > Edit Group** command.

- **Share Group**

  Contains all members of a control group that share the same executable. TotalView automatically places processes in share groups based on their control group and their executable.

You can't change a share group's members. However, the dynamically loaded libraries used by group members can be different.

The following are the predefined thread groups:

- **Workers Group**

  Contains all worker threads from all processes in the control group. The only threads not contained in a worker's group are your operating system's manager threads.

- **Lockstep Group**

  Contains every stopped thread in a share group that has the same PC. TotalView creates one lockstep group for every thread. For example, suppose two threads are stopped at the same PC. TotalView creates two lockstep groups. While each lockstep group has the same two members, they differ in that each has a different TOI. While there are some circumstances

where this is important, you can usually ignore this distinction. That is, while two lockstep groups exist if two threads are stopped at the same PC, ignoring the second lockstep group is almost never harmful.

The group ID value for a lockstep group differs from the ID of other groups. Instead of having an automatically and transient integer ID, the lockstep group ID is **pid.tid**, where **pid.tid** identifies the thread with which it is associated. For example, the lockstep group for thread 2 in process 1 is **1.2**.

In general, if you're debugging a multiprocess program, the control group and share group differ only when the program has children that it forked with by calling the **execve()** function.

Specifying Groups in P/T Sets

This section extends the arena specifier syntax to include groups.

If you do not include a group specifier, the default is the control group. For example, the CLI only displays a target group in the focus string if you set it to something other than the default value.

You most often use target group specifiers with the stepping commands, as they give these commands more control over what's being stepped.

Use the following format to add groups to an arena specifier:

[*width_letter*][*group_indicator*][*pid*][.*tid*]

This format adds the *group_indicator* to what was discussed in "*Specifying Processes and Threads*" on page 213.

In the description of this syntax, everything appears to be optional. But, while no single element is required, you must enter at least one element. TotalView determines other values based on the current focus.

TotalView lets you identify a group by using a letter, number, or name.

A Group Letter

You can name one of the debugger's predefined sets. Each set is identified by a letter. For example, the following command sets the focus to the workers group:

```
dfocus W
```

The following are the group letters. These letters are in uppercase:

C — *Control group*

All processes in the control group.

D — *Default control group*

All processes in the control group. The only difference between this specifier and the C specifier is that this letter tells the CLI not to display a group letter in the CLI prompt.

S — *Share group*

The set of processes in the control group that have the same executable as the arena's TOI.

W	*Workers group* The set of all worker threads in the control group.
L	*Lockstep group* A set that contains all threads in the share group that have the same PC as the arena's TOI. If you step these threads as a group, they proceed in lockstep.

A Group Number

You can identify a group by the number TotalView assigns to it. The following example sets the focus to group 3:

```
dfocus 3/
```

The trailing slash tells TotalView that you are specifying a group number instead of a PID. The slash character is optional if you're using a *group_letter*. However, you must use it as a separator when entering a numeric group ID and a **pid.tid** pair. For example, the following example identifies process 2 in group 3:

```
p3/2
```

A Group Name

You can name a set that you define. You enter this name with slashes. The following example sets the focus to the set of threads contained in process 3 that are also contained in a group called **my_group**:

```
dfocus p/my_group/3
```

About Arena Specifier Combinations

The following table lists what's selected when you use arena and group specifiers to step your program:

Specifier	Specifies
aC	All threads.
aS	All threads.
aW	All threads in all workers groups.
aL	All threads. Every thread is a member of a control group and a member of a share group and a member of a lockstep group. Consequently, three of these definitions mean "all threads."
gC	All threads in the Thread of Interest (TOI) control group.
gS	All threads in the TOI share group.
gW	All worker threads in the control group that contains the TOI.
gL	All threads in the same share group within the process that contains the TOI that have the same PC as the TOI.
pC	All threads in the control group of the Process of Interest (POI). This is the same as **gC**.
pS	All threads in the process that participate in the same share group as the TOI.
pW	All worker threads in the POI.
pL	All threads in the POI whose PC is the same as the TOI.
tC tS tW tL	Just the TOI. The **t** specifier overrides the group specifier. So, for example, **tW** and **t** both name the current thread.

Stepping commands behave differently if the group being stepped is a process group rather than a thread group. For example, **aC** *and* **aS** *perform the same action, but* **aL** *is different.*

If you don't add a PID or TID to your arena specifier, TotalView does it for you, taking the PID and TID from the current or default focus.

The following are some additional examples. These examples add PIDs and TIDs numbers to the raw specifier combinations listed in the previous table:

pW3	All worker threads in process 3.
pW3.<	All worker threads in process 3. The focus of this specifier is the same as the focus in the previous example.
gW3	All worker threads in the control group that contains process 3. The difference between this and **pW3** is that **pW3** restricts the focus to one of the processes in the control group.
gL3.2	All threads in the same *share* group as process 3 that are executing at the same PC as thread 2 in process 3. The reason this is a share group and not a control group is that different share groups can reside only in one control group.
/3	Specifies processes and threads in process 3. The arena width, POI, and TOI are inherited from the existing P/T set, so the exact meaning of this specifier depends on the previous context. While the slash is unnecessary because no group is indicated, it is syntactically correct.
g3.2/3	The **3.2** group ID is the name of the lockstep group for thread 3.2. This group includes all threads in the process 3 share group that are executing at the same PC as thread 2.
p3/3	Sets the process to process 3. The Group of Interest (GOI) is set to group 3. If group 3 is a process group, most commands ignore the group setting. If group 3 is a thread group, most commands act on all threads in process 3 that are also in group 3. When you set the process using an explicit group, you might not be including all the threads you expect to be included. This is because commands must look at the TOI, POI, and GOI.

It is redundant to specify a thread width with an explicit group ID *as this width means that the focus is on one thread.*

In the following examples, the first argument to the **dfocus** command defines a temporary P/T set that the CLI command (the last term) operates on. The **dstatus** command lists information about processes and threads. These examples assume that the global focus was **d1.<** initially.

dfocus g dstatus
Displays the status of all threads in the control group.

dfocus gW dstatus
Displays the status of all worker threads in the control group.

dfocus p dstatus
Displays the status of all worker threads in the current focus process. The width here, as in the previous example, is process, and the (default) group is the control group. The intersection of this width and the default group creates a focus that is the same as in the previous example.

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

The following example shows how the prompt changes as you change the focus. In particular, notice how the prompt changes when you use the **C** and the **D** group specifiers.

```
d1.<> f C
dC1.<
dC1.<> f D
d1.<
d1.<>
```

Two of these lines end with the less-than symbol (<). These lines aren't prompts. Instead, they are the value returned by TotalView when it executes the **dfocus** command.

'All' Does Not Always Mean 'All'

When you use stepping commands, TotalView determines the scope of what runs and what stops by looking at the TOI. This section looks at the differences in behavior when you use the **a** (all) arena specifier. The following table describes what runs when you use this arena:

Specifier	Specifies
aC	All threads.
aS	All threads.
aW	All threads in all workers groups.
aL	All threads. Every thread is a member of a control group and a member of a share group and a member of a lockstep group. Consequently, three of these definitions mean "all threads."

The following are some combinations:

f aC dgo
Runs everything. If you're using the **dgo** command, everything after the **a** is ignored: **a/aPizza/17.2**, **ac**, **aS**, and **aL** do the same thing. TotalView runs everything.

f aC duntil
While everything runs, TotalView must wait until something reaches a goal. It really isn't obvious what this focus is. Since **C** is a process group, you might guess that

all processes run until at least one thread in every participating process arrives at a goal. The reality is that since this goal must reside in the current share group, this command completes as soon as all processes in the TOI share group have at least one thread at the goal. Processes in other control groups run freely until this happens.

The TOI determines the goal. If there are other control groups, they do not participate in the goal.

f aS duntil This command does the same thing as the **f aC duntil** command because the goals for **f aC duntil** and **f aS duntil** are the same, and the processes that are in this scope are identical.

Although more than one share group can exist in a control group, these other share groups do not participate in the goal.

f aL duntil Although everything will run, it is not clear what should occur. **L** is a thread group, so you might expect that the **duntil** command will wait until all threads in all lockstep groups arrive at the goal. Instead, TotalView defines the set of threads that it allows to run to a goal as just those thread in the TOI' lockstep group. Although there are other lockstep groups, these lockstep groups do not participate in the goal. So, while the TOI's lockstep threads are progressing towards their goal, all threads that were previously stopped run freely.

f aW duntil Everything runs. TotalView waits until all members of the TOI workers group arrive at the goal.

There are two broad distinctions between process and thread group behavior:

- When the focus is on a process group, TotalView waits until just one thread from each participating process arrives at the goal. The other threads just run.

  When focus is on a thread group, every participating thread must arrive at the goal.
- When the focus is on a process group, TotalView steps a thread over the goal breakpoint and continues the process if it isn't the right thread.

  When the focus is on a thread group, TotalView holds a thread even if it isn't the right thread. It also continues the rest of the process.

  If your system doesn't support asynchronous thread control, TotalView treats thread specifiers as if they were process specifiers.

With this in mind, **f aL dstep** does not step all threads. Instead, it steps only the threads in the TOI's lockstep group. All other threads run freely until the stepping process for these lockstep threads completes.

Setting Groups

This section presents a series of examples that set and create groups.

You can use the following methods to indicate that thread 3 in process 2 is a worker thread:

`dset WGROUP(2.3) $WGROUP(2)`
: Assigns the group ID of the thread group of worker threads associated with process 2 to the **WGROUP** variable. (Assigning a nonzero value to **WGROUP** indicates that this is a worker group.)

`dset WGROUP(2.3) 1`
: This is a simpler way of doing the same thing as the previous example.

`dfocus 2.3 dworker 1`
: Adds the groups in the indicated focus to a workers group.

```
dset CGROUP(2) $CGROUP(1)
dgroups -add -g $CGROUP(1) 2
dfocus 1 dgroups -add 2
```
: These three commands insert process 2 into the same control group as process 1.

`dgroups -add -g $WGROUP(2) 2.3`
: Adds process 2, thread 3 to the workers group associated with process 2.

`dfocus tW2.3 dgroups -add`
: This is a simpler way of doing the same thing as the previous example.

The following are some additional examples:

`dfocus g1 dgroups -add -new thread`
: Creates a new thread group that contains all the threads in all the processes in the control group associated with process 1.

```
set mygroup [dgroups -add -new thread
$GROUP($SGROUP(2))]
dgroups -remove -g $mygroup 2.3
dfocus g$mygroup/2 dgo
```
: These three commands define a new group that contains all the threads in the process 2 share group except for thread 2.3, and then continue that set of threads. The first command creates a new group that contains all the threads from the share group; the second removes thread 2.3; and the third runs the remaining threads.

Using the g Specifier: An Extended Example

The meaning of the **g** width specifier is sometimes not clear when it is coupled with a group scope specifier. Why have a **g** specifier when you have four other group specifiers? Stated in another way, isn't something like **gL** redundant?

The simplest answer, and the reason you most often use the **g** specifier, is that it forces the group when the default focus indicates something different from what you want it to be.

The following example shows this. The first step sets a breakpoint in a multithreaded OMP program and execute the program until it hits the breakpoint.

```
d1.<> dbreak 35
Loaded OpenMP support library libguidedb_3_8.so :
                  KAP/Pro Toolset 3.8
1
d1.<> dcont
Thread 1.1 has appeared
Created process 1/37258, named "omp_prog"
Thread 1.1 has exited
Thread 1.1 has appeared
Thread 1.2 has appeared
Thread 1.3 has appeared
Thread 1.1 hit breakpoint 1 at line 35 in
".breakpoint_here"
```

The default focus is `d1.<`, which means that the CLI is at its default width, the POI is 1, and the TOI is the lowest numbered nonmanager thread. Because the default width for the **dstatus** command is process, the CLI displays the status of all processes. Typing **dfocus p dstatus** produces the same output.

```
d1.<> dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Stopped     PC=0xd042c944
d1.<> dfocus p dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Stopped     PC=0xd042c944
```

The CLI displays the following when you ask for the status of the lockstep group. (The rest of this example uses the **f** abbreviation for **dfocus**, and **st** for **dstatus**.)

```
d1.<> f L st
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
```

This command tells the CLI to display the status of the threads in thread, which is the 1.1 lockstep group since this thread is the TOI. The **f L focus** command narrows the set so that the display only includes the threads in the process that are at the same PC as the TOI.

By default, the dstatus *command displays information at process width. This means that you don't need to type* f pL dstatus.

The **duntil** command runs thread 1.3 to the same line as thread 1.1. The **dstatus** command then displays the status of all the threads in the process:

```
d1.<> f t1.3 duntil 35
    35@>            write(*,*)"i= ",i,
                      "thread= ",omp_get_thread_num()
d1.<> f p dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
```

As expected, the CLI adds a thread to the lockstep group:

```
d1.<> f L dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.3: 37258.3  Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
```

The next set of commands begins by narrowing the width of the default focus to thread width—notice that the prompt changes—and then displays the contents of the lockstep group:

```
d1.<> f t
t1.<> f L dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
```

Although the lockstep group of the TOI has two threads, the current focus has only one thread, and that thread is, of course, part of the lockstep group. Consequently, the lockstep group *in the current focus* is just the one thread, even though this thread's lockstep group has two threads.

If you ask for a wider width (**p** or **g**) with **L**, the CLI displays more threads from the lockstep group of thread 1.1. as follows:

```
t1.<> f pL dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
t1.<> f gL dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
t1.<>
```

If the TOI *is* 1.1, L *refers to group number* 1.1, *which is the lockstep group of thread* 1.1.

Because this example only contains one process, the **pL** and **gL** specifiers produce the same result when used with the **dstatus** command. If, however, there were additional processes in the group, you only see them when you use the **gL** specifier.

Merging Focuses

When you specify more than one focus for a command, the CLI merges them together. In the following example, the focus indicated by the prompt—this focus is called the *outer* focus—controls the display. This example shows what happens when **dfocus** commands are strung together:

```
t1.<> f d
d1.<
d1.<> f tL dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
d1.<> f tL f p dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
d1.<> f tL f p f D dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
d1.<> f tL f p f D f L dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
d1.<>
```

Stringing multiple focuses together might not produce the most readable result. In this case, it shows how one **dfocus** command can modify what another sees and acts on. The ultimate result is an arena that a command acts on. In these examples, the **dfocus** command tells the **dstatus** command what to display.

Naming Incomplete Arenas

In general, you do not need to completely specify an arena. TotalView provides values for any missing elements. TotalView either uses built-in default values or obtains them from the current focus. The following explains how TotalView fills in missing pieces:

- If you don't use a width, TotalView uses the width from the current focus.
- If you don't use a PID, TotalView uses the PID from the current focus.

- If you set the focus to a list, there is no longer a default arena. This means that you must explicitly name a width and a PID. You can, however, omit the TID. (If you omit the TID, TotalView defaults to the less-than symbol <.)

  You can type a PID without typing a TID. If you omit the TID, TotalView uses the default **<**, where **<** indicates the lowest numbered worker thread in the process. If, however, the arena explicitly names a thread group, **<** means the lowest numbered member of the thread group.

  TotalView doesn't use the TID from the current focus, since the TID is a process-relative value.
- A dot before or after the number specifies a process or a thread. For example, **1.** is clearly a PID, while **.7** is clearly a TID.

  If you type a number without typing a dot, the CLI most often interprets the number as being a PID.
- If the width is **t**, you can omit the dot. For instance, **t7** refers to thread 7.
- If you enter a width and don't specify a PID or TID, TotalView uses the PID and TID from the current focus.

  If you use a letter as a group specifier, TotalView obtains the rest of the arena specifier from the default focus.
- You can use a group ID or tag followed by a **/**. TotalView obtains the rest of the arena from the default focus.

Focus merging can also influence how TotalView fills in missing specifiers. For more information, see "*Merging Focuses*" on page 225.

Naming Lists with Inconsistent Widths

TotalView lets you create lists that contain more than one width specifier. This can be very useful, but it can be confusing. Consider the following:

```
{p2 t7 g3.4}
```

This list is quite explicit: all of process 2, thread 7, and all processes in the same group as process 3, thread 4. However, how should TotalView use this set of processes, groups, and threads?

In most cases, TotalView does what you would expect it to do: it iterates over the list and acts on each arena. If TotalView cannot interpret an inconsistent focus, it prints an error message.

Some commands work differently. Some use each arena's width to determine the number of threads on which it acts. This is exactly what the **dgo** command does. In contrast, the **dwhere** command creates a call graph for process-level arenas, and the **dstep** command runs all threads in the arena while stepping the TOI. TotalView may wait for threads in multiple processes for group-level arenas. The command description in the *TotalView Reference Guide* points out anything that you need to watch out for.

Stepping (Part II): Examples

The following are examples that use the CLI stepping commands:

- **Step a single thread**

  While the thread runs, no other threads run (except kernel manager threads).

  Example: `dfocus t dstep`
- **Step a single thread while the process runs**

  A single thread runs into or through a critical region.

  Example: `dfocus p dstep`
- **Step one thread in each process in the group**

  While one thread in each process in the share group runs to a goal, the rest of the threads run freely.

  Example: `dfocus g dstep`
- **Step all worker threads in the process while nonworker threads run**

  Worker threads run through a parallel region in lockstep.

  Example: `dfocus pW dstep`
- **Step all workers in the share group**

  All processes in the share group participate. The nonworker threads run.

  Example: `dfocus gW dstep`
- **Step all threads that are at the same PC as the TOI**

  TotalView selects threads from one process or the entire share group. This differs from the previous two items in that TotalView uses the set of threads are in lockstep with the TOI rather than using the workers group.

  Example: `dfocus L dstep`

In the following examples, the default focus is set to `d1.<`.

dstep	Steps the TOI while running all other threads in the process.
dfocus W dnext	Runs the TOI and all other worker threads in the process to the next statement. Other threads in the process run freely.
dfocus W duntil 37	Runs all worker threads in the process to line 37.
dfocus L dnext	Runs the TOI and all other stopped threads at the same PC to the next statement. Other threads in the process run freely. Threads that encounter a temporary breakpoint in the course of running to the next statement usually join the lockstep group.
dfocus gW duntil 37	Runs all worker threads in the share group to line 37. Other threads in the control group run freely.
UNW 37	Performs the same action as the previous command: runs all worker threads in the share group to line 37. This example uses the predefined UNW alias instead of the individual commands. That is, UNW is an alias for dfocus gW duntil.

SL	Finds all threads in the share group that are at the same PC as the TOI and steps them all in one statement. This command is the built-in alias for **dfocus gL dstep**.
sl	Finds all threads in the current process that are at the same PC as the TOI, and steps them all in one statement. This command is the built-in alias for **dfocus L dstep**.

Using P/T Set Operators

At times, you do not want all of one type of group or process to be in the focus set. TotalView lets you use the following three operators to manage your P/T sets:

|	Creates a union; that is, all members of the sets.
-	Creates a difference; that is, all members of the first set that are not also members of a second set.
&	Creates an intersection; that is, all members of the first set that are also members of the second set.

For example, the following creates a union of two P/T sets:

```
p3 | L2
```

A set operator only operates on two sets. You can, however, apply these operations repeatedly; for example:

```
p2 | p3 & L2
```

This statement creates a union between **p2** and **p3**, and then creates an intersection between the union and **L2**. As this example suggests, TotalView associates sets from left to right. You can change this order by using parentheses; for example:

```
p2 | (p3 & pL2)
```

Typically, these three operators are used with the following P/T set functions:

breakpoint()	Returns a list of all threads that are stopped at a breakpoint.
error()	Returns a list of all threads stopped due to an error.
existent()	Returns a list of all threads.
held()	Returns a list of all threads that are held.
nonexistent()	Returns a list of all processes that have exited or which, while loaded, have not yet been created.
running()	Returns a list of all running threads.
stopped()	Returns a list of all stopped threads.
unheld()	Returns a list of all threads that are not held.
watchpoint()	Returns a list of all threads that are stopped at a watchpoint.

The argument that all of these operators use is a P/T set. You specify this set in the same way that you specify a P/T set for the **dfocus** command. For example, **watchpoint(L)** returns all threads in the current lockstep group.

The dot operator (.), which indicates the current set, can be helpful when you are editing an existing set.

The following examples clarify how you use these operators and functions. The P/T set **a** (all) is the argument to these operators.

f {breakpoint(a) | watchpoint(a)} dstatus
: Shows information about all threads that stopped at breakpoints and watchpoints. The a argument is the standard P/T set indicator for **all**.

f {stopped(a) - breakpoint(a)} dstatus
: Shows information about all stopped threads that are not stopped at breakpoints.

f {. | breakpoint(a)} dstatus
: Shows information about all threads in the current set, as well as all threads stopped at a breakpoint.

f {g.3 - p6} duntil 577
: Runs thread 3 along with all other processes in the group to line 577. However, it does not run anything in process 6.

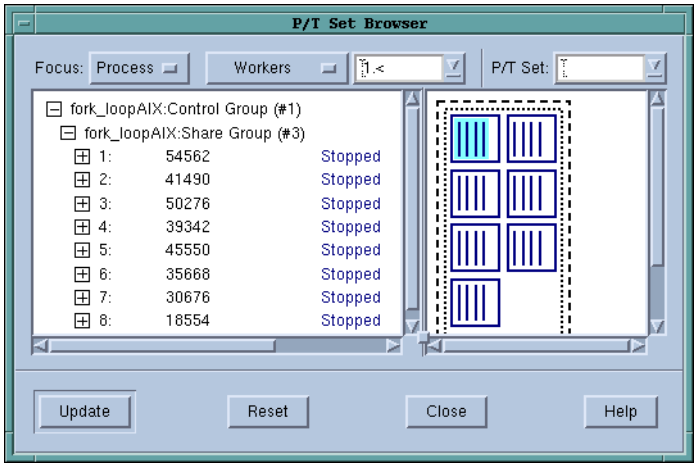
f {($PTSET) & p123}
: Uses just process 123 in the current P/T set.

Using the P/T Set Browser

There are few programs that need all the power that the P/T set syntax provides. When you need this power, the ability to previsualize the contents of a P/T set before you execute a command can be essential when you need to specify a complicated set. This is what the P/T Set Browser does. The browser, which is accessed from the Root Window **Tool** menu, shows the current state of processes and threads, as well as show what is or will be selected when you specify a P/T set. The figure on the next page shows a P/T set browser displaying information about a multiprocess, multithreaded program.

The top part of this window contains the standard P/T set controls. (See "*Using P/T Set Controls*" on page 210 for more information.) The large area on the left is a tree control, where clicking on the plus sign (+) shows more information, and clicking on a minus sign (–) (not shown in this figure) condenses the information. This large area displays a list of all your program's processes and threads. The information is organized in a hierarchy, with the outermost level being your program's control groups. In a control group, information is further organized by share group, where you are shown the processes contained in a share group. Finally, if you click the innermost + symbols, the browser shows information on the threads in a process.

Figure 147: A P/T Set Browser Window



The control and share group numbers displayed in this window are the same as those that are displayed in the Groups Page in the Root Window.

The right-hand side of the P/T Set Browser contains a graphical depiction of your program's threads. In the preceding figure, TotalView highlights some of the threads. These are the threads in the current focus, which in this case is 1.<. As you make changes to the P/T set, the threads highlighted in the right-hand side change, showing you the scope of a P/T set definition. The figure on the next page shows a variety of P/T set examples.

The numbers on this figure indicate the following:

❶ This P/T set differs from the one in the previous figure in that the **Focus** pulldown menu is now set to **All**. TotalView responds by highlighting all threads on the right-hand side.

❷ The **Focus** pulldown menu was changed back to **Process**, but the number of processes was limited to 1, 2, and 3. Before these changes were made, process 3 was told to go. The browser shows those processes as running.

❸ Thread 3.1 was halted.

❹ Thread 2.4 was selected with the mouse. It doesn't matter if it was selected in the left- or right-hand sides, as selecting it causes it to be highlighted in both. After selecting a thread, you can extend the selection by clicking the left-mouse button on another thread while holding down the Shift key. You can select noncontiguous threads by holding down the Control key while clicking your left-mouse button.

If you are seeing this document online, your selection is in grey, while the selection that indicates the P/T set is in blue.

❺ The P/T set information was modified to show a difference expression; in this case, thread 1.3 was eliminated from the set of threads named by **p1.<**.

The elements on the right side are drawn within two boxes. These boxes represent the control and share groups. Clicking on them tells the browser to select that group.

Using the Group Editor

The visual group editor, which is displayed after you select the **Group > Edit Group** command, can simplify the way in which you create named groups.
The top half of this dialog box lets you add, update, and delete named sets. The bottom half contains controls that let you specify which processes and threads become part of the group. These controls were are discussed in the previous section.

When using the controls in the upper part of the dialog box, you need to be careful to give the group a name, before you click on the **Add** button. Details on using the controls in this dialog box are contained in the online Help.

Figure 148: P/T Set Browser Windows

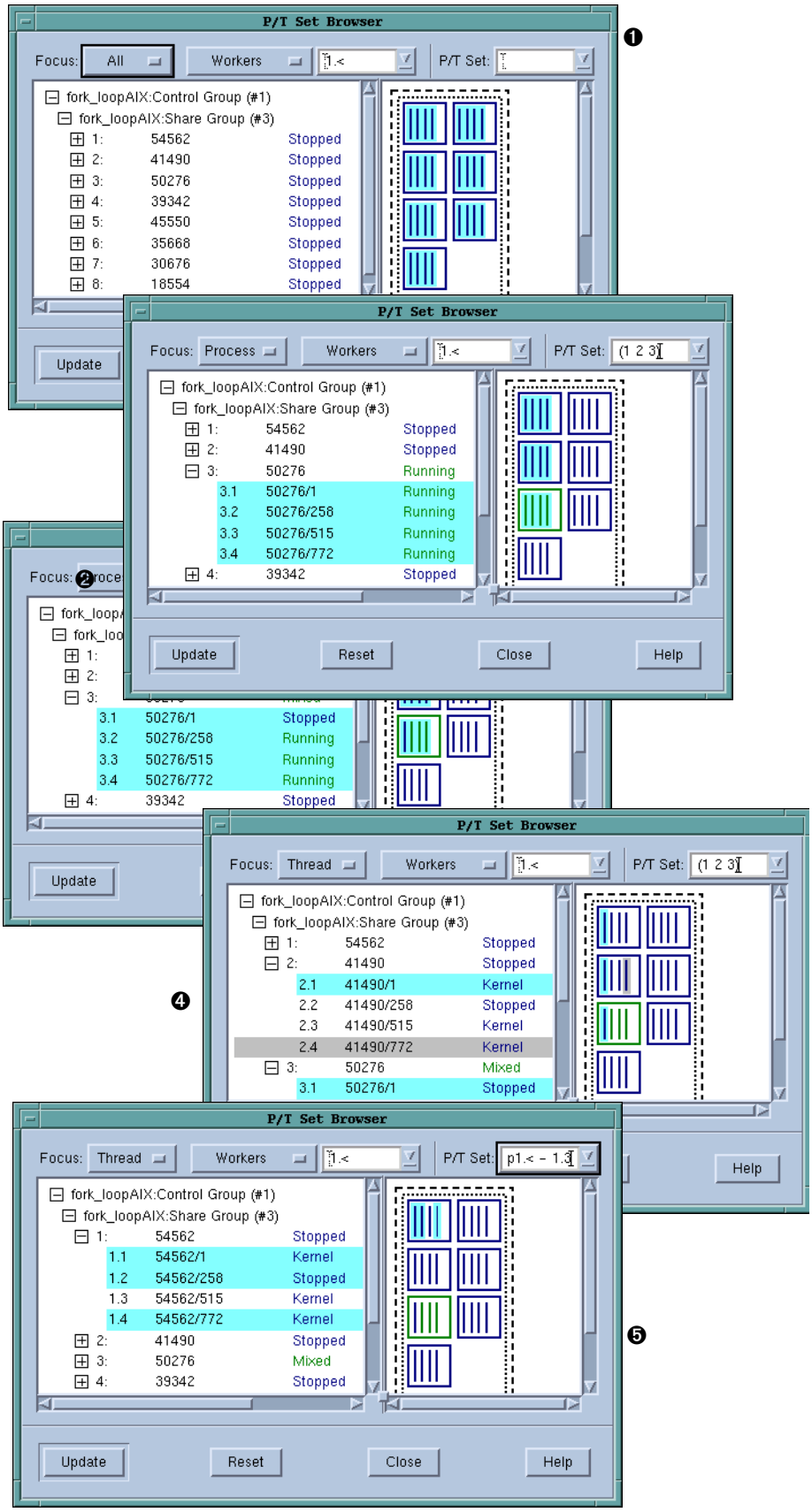
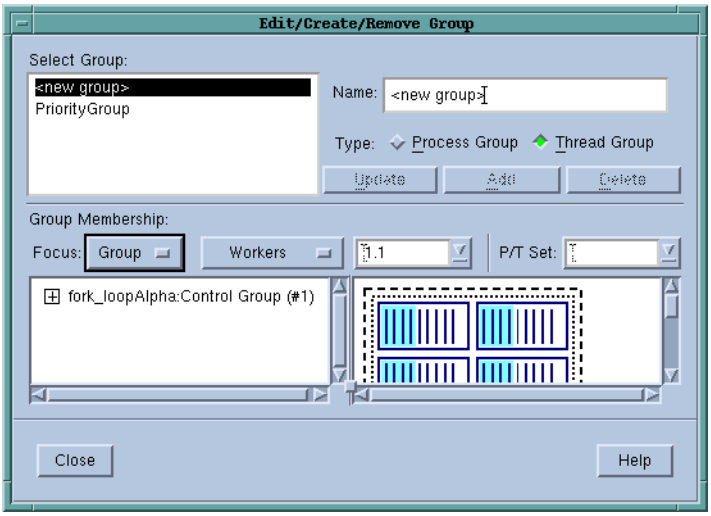


Figure 149: Group > Edit Group Dialog Box



Examining and Changing Data

12

This chapter explains how to examine and change data as you debug your program.

This chapter contains the following sections:

- "*Changing How Data is Displayed*" on page 235
- "*Displaying Variables*" on page 238
- "*Diving in Variable Windows*" on page 249
- "*Viewing a List of Variables*" on page 254
- "*Changing the Values of Variables*" on page 259
- "*Changing a Variable's Data Type*" on page 260
- "*Changing the Address of Variables*" on page 267
- "*Displaying C++ Types*" on page 268
- "*Displaying Fortran Types*" on page 269
- "*Displaying Thread Objects*" on page 274
- "*Scoping and Symbol Names*" on page 274

This chapter does not discuss array data. For that information, see Chapter 13, "*Examining Arrays*," on page 279.

Changing How Data is Displayed

When a debugger displays a variable, it relies on the definitions of the data used by your compiler. The following two sections show how you can change the way TotalView displays this information:

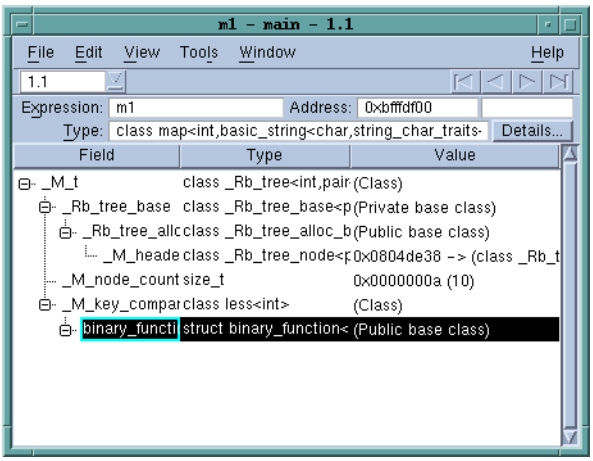
- "*Displaying STL Variables*" on page 235
- "*Changing Size and Precision*" on page 238

Displaying STL Variables

The C++ STL (Standard Template Library) greatly simplifies the way in which you can access data. Since it offers standard and prepackaged ways to organize data, you do not have to be concerned with the mechanics of the access method. The disadvantage to using the STL while debugging is that the information debuggers display is organized according to the com-

piler's view of the data, rather than the STL's logical view. For example, here is how your compiler sees a map compiled using the GNU C++ compiler (gcc):

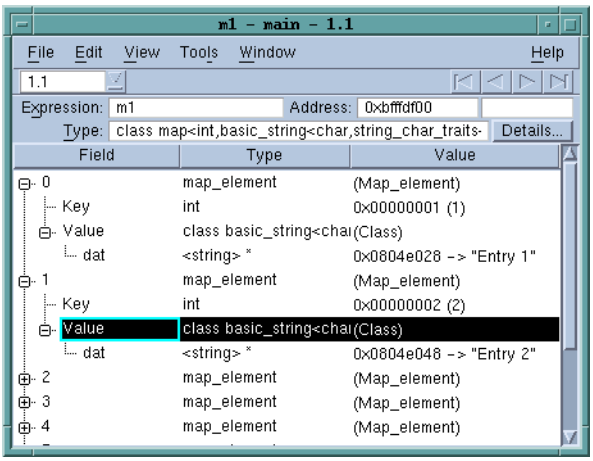
Figure 150: An Untransformed Map



Most of the information is generated by the STL template and, in most cases, is not interesting. In addition, the STL does not aggregate the information in a useful way.

STLView solves these problems by rearranging (that is, *transforming*) the data so that you can easily examine it. For example, here is the transformed map.

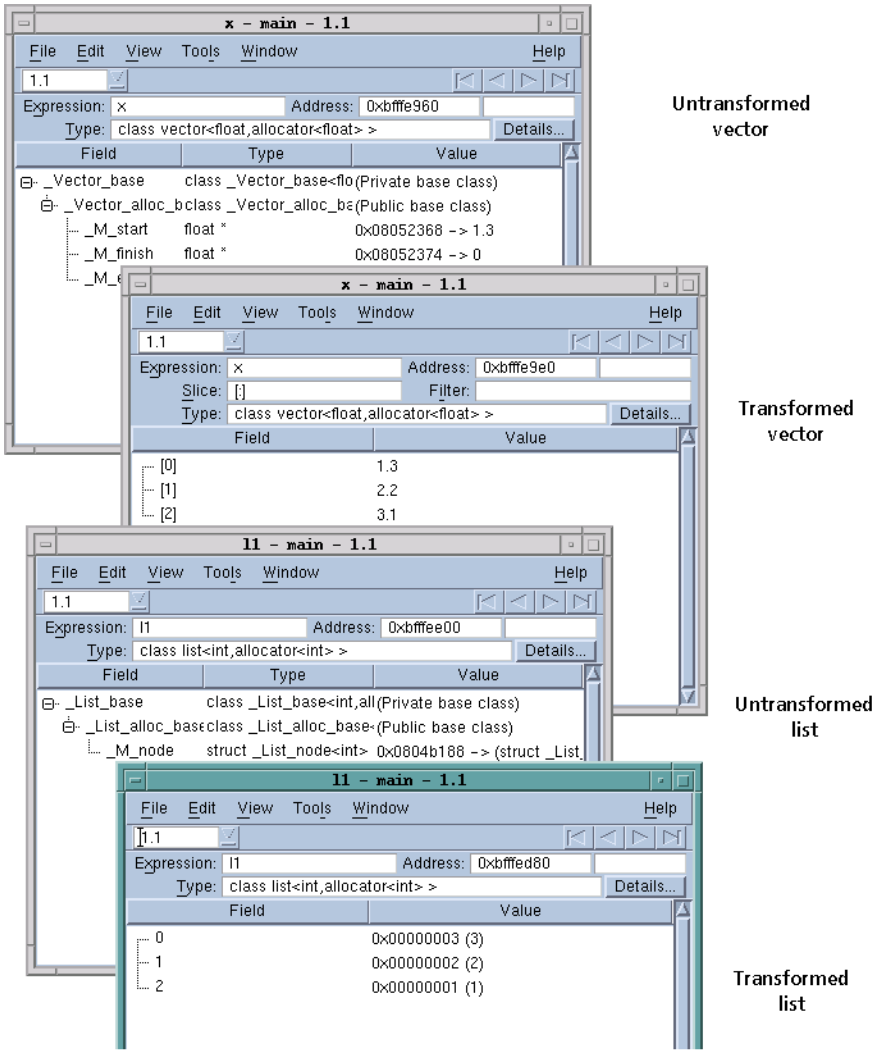
Figure 151: A Transformed Map



Using native and GCC compilers on IBM AIX, IRIX/MIPS, HP Tru64 Alpha, and Sun Solaris, TotalView can transform STL strings, vectors, lists, and maps. TotalView can also transform these STL types if you are using GCC and Intel Version 7 and 8 C++ 32-bit compiler running on the Red Hat x86 platform. The *TotalView Platforms Guide* names the compilers for which TotalView transforms STL data types.

The following figure shows an untransformed and transformed list and vector.

Figure 152: List and Vector Transformations



You can create transformations for other STL *containers. See the "TotalView Reference Guide" for more information.*

By default, TotalView transforms STL types. If you need to look at the untransformed data structures, clear the **View simplified STL containers (and user-defined transformations)** checkbox on the Options Page of the **File > Preference** Dialog Box.

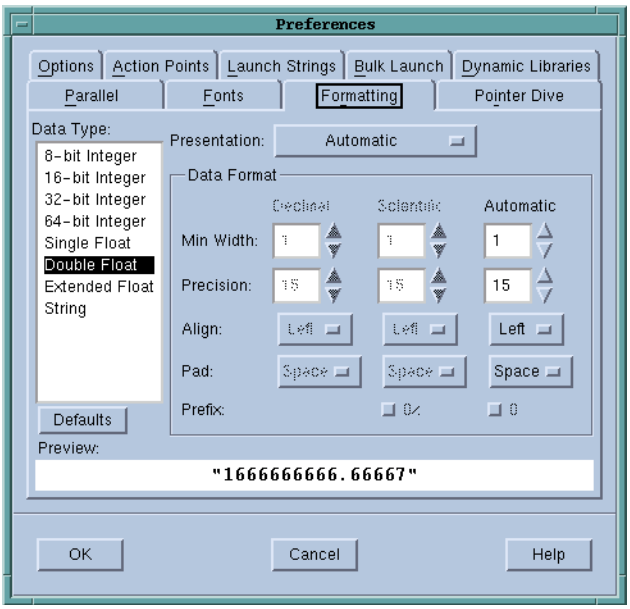
CLI: **dset TV::ttf { true | false }**

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

Changing Size and Precision

If the defaults formats that TotalView uses to display a variable's value doesn't t meet your needs, you can use the Formatting Page of the **File > Preferences** Dialog Box to indicate how precise you want the simple data types to be.

Figure 153: File > Preferences Formatting Page



After selecting one of the data types listed on the left side of the Formatting Page, you can set how many character positions a value uses when TotalView displays it (**Min Width**) and how many numbers to display to the right of the decimal place (**Precision**). You can also tell TotalView how to align the value in the **Min Width** area, and if it should pad numbers with zeros or spaces.

Although the way in which these controls relate and interrelate may appear to be complex, the **Preview** area shows you the result of a change. Play with the controls for a minute or so to see what each control does. You may need to set the **Min Width** value to a larger number than you need it to be to see the results of a change. For example, if the **Min Width** value doesn't allow a number to be justified, it could appear that nothing is happening.

CLI: You can set these properties from within the CLI. To obtain a list of variables that you can set, type "dset TV::data_format*".

Displaying Variables

The Process Window Stack Frame Pane displays variables that are local to the current stack frame. This pane doesn't show the data for nonsimple variables, such as pointers, arrays, and structures. To see this information, you need to dive on the variable. This tells TotalView to display a Variable

Window that contains the variable's data. For example, diving on an array variable tells TotalView to display the entire contents of the array.

Dive on a variable by clicking your middle mouse button on it. If your mouse doesn't have three buttons, you can single- or double-click on an item.

If you dive on simple variables or registers, TotalView still brings up a Variable Window; however, you will see some additional information about the variable or register.

Although a Variable Window is the best way to see all of an array's elements or all elements in a structure, using the **Expression List** Window is easier for variables with one value. Using it also cuts down on the number of windows that are open at any one time. For more information, see "*Viewing a List of Variables*" on page 254.

The following sections discuss how you can display variable information in TotalView:

- "*Displaying Program Variables*" on page 239
- "*Displaying Variables in the Current Block*" on page 241
- "*Viewing a Variable in Different Scopes as Your Program Executes*" on page 242
- "*Scoping Issues*" on page 242
- "*Browsing for Variables*" on page 243
- "*Displaying Local Variables and Registers*" on page 244
- "*Dereferencing Variables Automatically*" on page 245
- "*Displaying Areas of Memory*" on page 246
- "*Displaying Machine Instructions*" on page 248
- "*Rebinding the Variable Window*" on page 248
- "*Closing Variable Windows*" on page 249

Displaying Program Variables

You can display local and global variables by:

- Diving into the variable in the Source or Stack Panes.
- Selecting the **View** > **Lookup Variable** command. When prompted, enter the name of the variable.

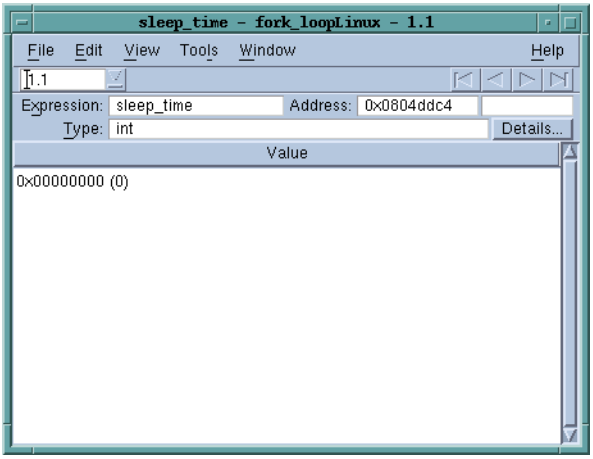
CLI:	**dprint** *variable*

- Using the **Tools** > **Program Browser** command.

After using one of these methods, TotalView displays a Variable Window that contains the information you want. The Variable Window can display simple variables, such as **int**s, sets of like elements such as arrays, or more complicated variables defined as structures and arrays of structures. (A Variable Window containing a global window is shown on the next page.)

If you keep a Variable Window open while a process or thread is running, the information being displayed might not be accurate. TotalView updates the window when the process or thread stops. If TotalView can't find a stack frame for a displayed local variable, the variable's status is **sparse**,

Figure 154: Variable Window for a Global Variable



since the variable no longer exists. The **Status** area can contain other information that alerts you to issues and problems with a variable.

When you debug recursive code, TotalView doesn't automatically refocus a Variable Window onto different instances of a recursive function. If you have a breakpoint in a recursive function, you need to explicitly open a new Variable Window to see the local variable's value in that stack frame.

> CLI: **dwhere, dup, and dprint**
>
> **Use dwhere to locate the stack frame, use dup to move to it, and then use dprint to display the value.**

Select the **View > Compilation Scope > Float** command to tell TotalView that it can refocus a Variable Window on different instances. For more information, see "*Viewing a Variable in Different Scopes as Your Program Executes*" on page 242.

Seeing Structure Information

When TotalView displays a Variable Window, it displays structures in a compact form, concealing the elements within the structure. Click the + button to display these elements, or select the **View > Expand All** command to see all entries. If you want to return the display to a more compact form, you can click the – button to collapse one structure, or select the **View > Collapse All** command to return the window to what it was when you first opened it.

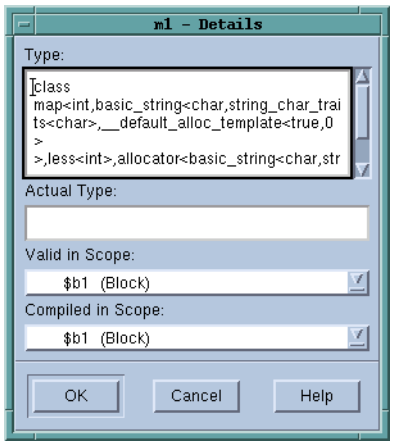
If a substructure contains more than about forty elements, TotalView does not let you expand it in line. That is, it does not place a + symbol in front of the substructure. To see the contents of this substructure, dive on it.

Similarly, if a structure contains an array as an element, TotalView only displays the array within the structure if it has less than about forty elements. To see the contents of an embedded array, dive on it.

Seeing More Information

If TotalView doesn't have enough space to display all the characters in a variable data type, click the **Details** button. TotalView responds by displaying the following dialog box.

Figure 155: The Details Dialog Box



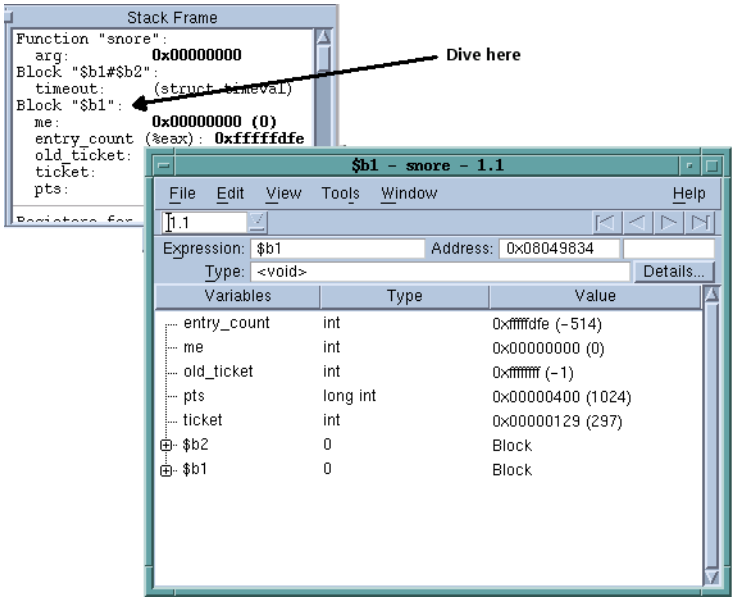
Because the **Type** field in this dialog box is larger than it is in the Variable Window, it is easier to see this information here than in the Variable Window.

Displaying Variables in the Current Block

GUI

In many cases, you may want to see all of the variables in the current block. If you dive on a block label in the Stack Frame Pane, TotalView opens a Variable Window that contains just those variables.

Figure 156: Displaying Scoped Variables



After you dive on a variable in this block window, TotalView displays a Variable Widow for that scoped variable. In this figure, block **$b1** has two nested blocks.

Viewing a Variable in Different Scopes as Your Program Executes

When TotalView displays a Variable Window, it understands the scope in which the variable exists. As your program executes, this scope doesn't change. In other words, if you're looking at variable **my_var** in one routine, and you then execute your program until it is within a second subroutine that also has a **my_var** variable, TotalView does not change the scope so that you are seeing the *in scope* variable.

If you would like TotalView to update a variable's scope as your program executes, select the **View > Compilation Scope > Floating** command. This tells TotalView that, when execution stops, it should look for the variable in the current scope. If it finds the variable, it displays the variable contained within the current scope.

Select the **View > Compilation Scope > Fixed** command to return TotalView to its default behavior, which is not to change the scope.

Selecting floating scope can be very handy when you are debugging recursive routines or have routines with identical names. For example, **i**, **j**, and **k** are popular names for counter variables.

Scoping Issues

When you dive into a variable from the Source Pane, the scope that TotalView uses is that associated with the current frame's PC; for example:

```
1: void f()
2: {
3:    int x;
4: }
5:
6: int main()
7: {
8:    int x;
9:}
```

If the PC is at line 3, which is in **f()**, and you dive on the **x** contained in **main()**, TotalView displays the value for the **x** in **f()**, not the **x** in **main()**. In this example, the difference is clear: TotalView chooses the PC's scope instead of the scope at the place where you dove. If you are working with templated and overloaded code, determining the scope can be impossible, since the compiler does not retain sufficient information. In all cases, you can click the **Details** button within the Variable window to display a **Details** Dialog Box. The **Valid in Scope** field can help you determine which instance of a variable you located.

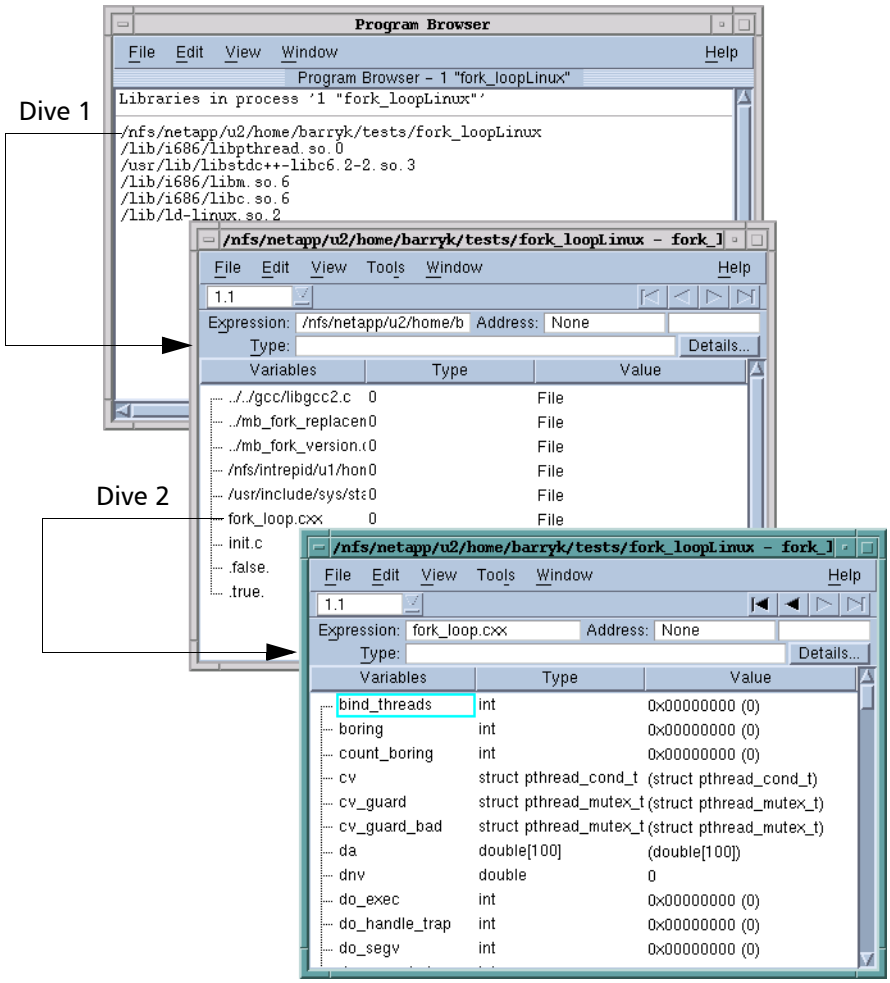
You can, of course, use the **View > Lookup Variable** command to locate the correct instance.

Browsing for Variables

GUI

The Process Window **Tools** > **Program Browser** command displays a window that contains all your executable's components. By clicking on a library or program name, you can access all of the variables contained in it.

Figure 157: Program Browser and Variable Windows (Part 1)



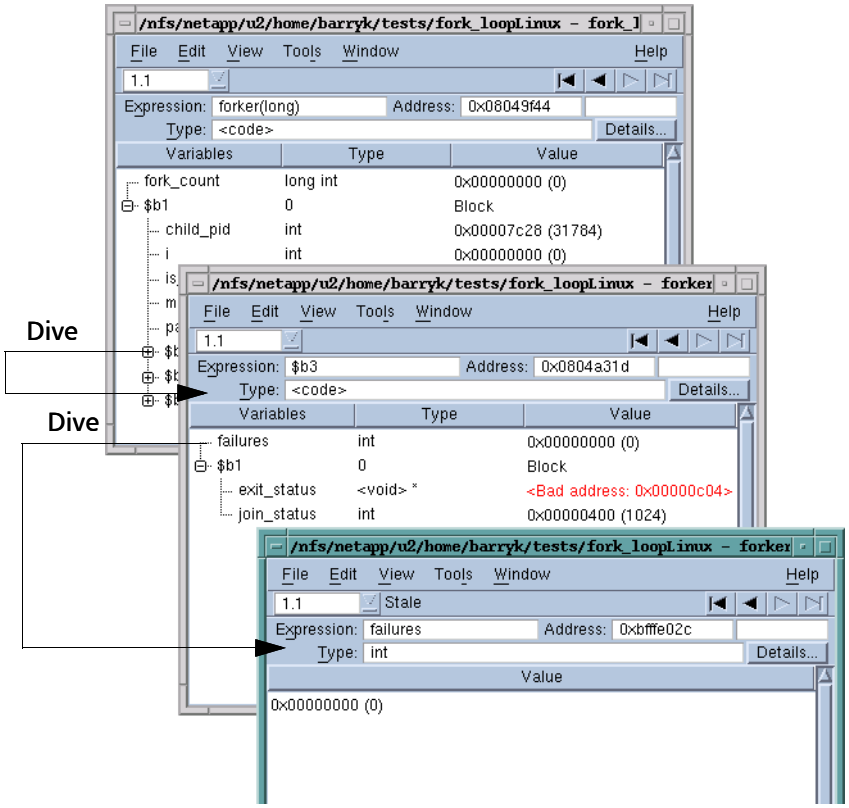
The window at the top of the figure shows programs and libraries that are loaded. If you have loaded more than one program with the **File** > **New Program** command, TotalView only displays information for the currently selected process. After diving on an entry in this window (labelled **Dive 1**), TotalView displays a Variable Window that contains a list of files that make up the program, as well as other related information.

Diving on an entry in this Variable Window (**Dive 2** in this figure) changes the Variable Window so that it contains variables and other information related to the file. A list of functions defined within the program is at the end of this list.

Diving on a function changes the Variable Window again. The window shown at the top of the next figure was created by diving on one of these

functions. The window shown in the center is the result of diving on a block in that subroutine. The bottom window shows a variable.

Figure 158: Program Browser and Variable Window (Part 2)

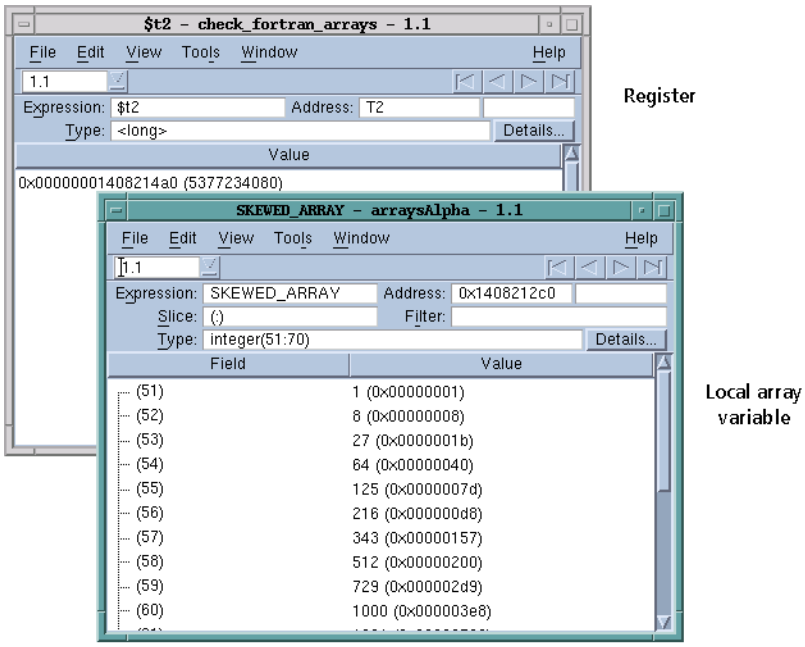


If you dive on a line in a Variable Window, the new contents replace the old contents, and you can use the undive/redive buttons to move back and forth.

Displaying Local Variables and Registers

In the Stack Frame Pane, diving on a function's parameter, local variable, or register tells TotalView to display information in a Variable Window. You can also dive on parameters and local variables in the Source Pane. The displayed Variable Window shows the name, address, data type, and value for the object. (The figure on the next page has an example.)

Figure 159: Diving on Local Variables and Registers



The window at the top of the figure shows the result of diving on a register, while the bottom window shows the results of diving on an array variable.

CLI: dprint *variable*

This command lets you view variables and expressions without having to select or find them.

You can also display local variables by using the **View > Lookup Variable** command. After TotalView displays a dialog box, enter the name of the variable you want to see.

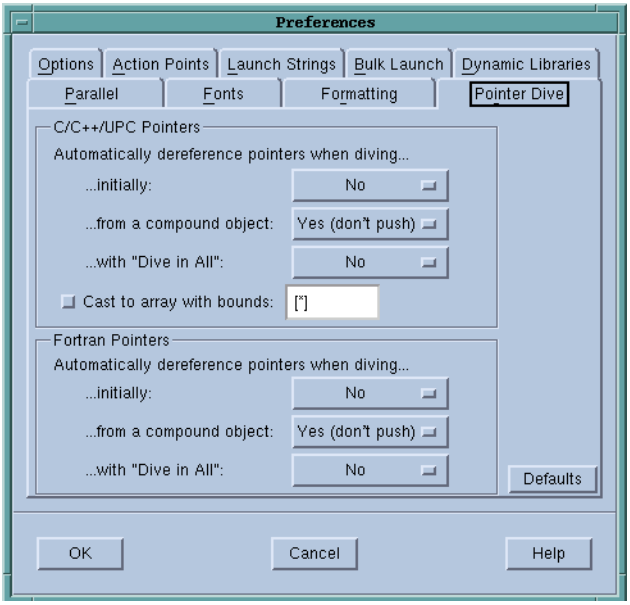
Dereferencing Variables Automatically

In most cases, you want to see what a pointer points to, rather than what the value of its variable is. Using the controls on the **File > Preferences** Pointer Dive Page (which is shown on the next page), you can tell TotalView to automatically dereference pointers.

Dereferencing pointers is especially useful when you want to visualize the data linked together with pointers, since it can present the data as a unified array. Because the data appears as a unified array, you can use the debugger's array manipulation commands and the Visualizer to view the data.

Each pulldown list on the Pointer Dive Page has three settings: **No**, **Yes**, and **Yes (don't push)**. The meaning for **No** is that automatic dereferencing does not occur. The remaining two values tell TotalView to automatically dereference pointers. The difference between the two is based on whether you can use the **Back** command to see the undereferenced pointer's value. If

Figure 160: File > Preferences Pointer Dive Page



you choose **Yes**, you can see the value. If you choose **Yes (don't push)**, you cannot use the **Back** command to see this pointer's value.

CLI: TV::auto_array_cast_bounds
TV::auto_deref_in_all_c
TV::auto_deref_in_all_fortran
TV::auto_deref_initial_c
TV::auto_deref_initial_fortran
TV::auto_deref_nested_c
TV::auto_deref_nested_fortran

Automatic dereferencing can occur in the following situations:

- When TotalView *initially* displays a value.
- When you dive on a value in an aggregate or structure.
- When you use the **Dive in All** command.

Displaying Areas of Memory

You can display areas of memory using hexadecimal, octal, or decimal values. Do this by selecting the **View > Lookup Variable** command, and then entering one of the following in the dialog box that appears:

- **An address**

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

  CLI: dprint *address*

- **A pair of addresses**

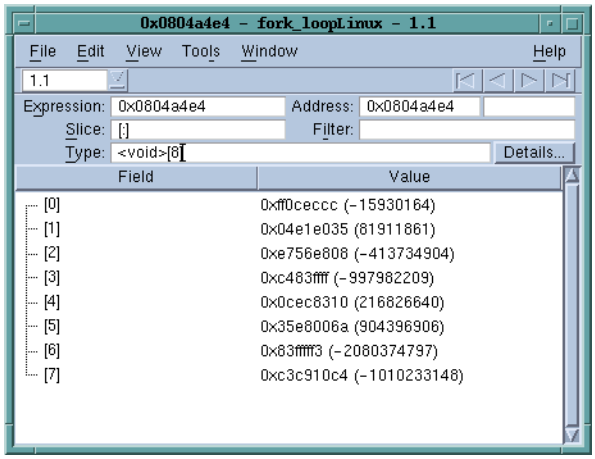
  When you enter a pair of addresses, TotalView displays the data (in word increments) from the first to the last address. To enter a pair of addresses, enter the first address, a comma, and the last address.

  CLI: dprint *address,address*

All octal constants must begin with 0 (*zero*). *Hexadecimal constants must begin with* 0*x*.

The Variable Window for an area of memory displays the address and contents of each word.

Figure 161: *Variable Window for an Area of Memory*



TotalView displays the memory area's starting location at the top of the window's data area. In the window, TotalView displays information in hexadecimal and decimal notation.

If a Variable Window is already being displayed, you can change the type to <void> and add an array specifier. If you do this, the results are similar to what is shown in this figure.

Changing Types to Display Machine Instructions

You can display machine instructions in a Variable Window by changing the text in the Variable Window **Type** field. All you need do is edit the type string to be an array of <code> data types. You also need to add an array specifier to tell TotalView how many instruction to display. For example, the following changes the Variable Window so that it displays three machine instructions:

```
<code>[3]
```

The Variable Window lists the following information about each machine instruction:

Offset+Label	The symbolic address of the location as a hexadecimal offset from a routine name.
Code	The hexadecimal value stored in the location.
Instruction	The instruction and operands stored in the location.

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

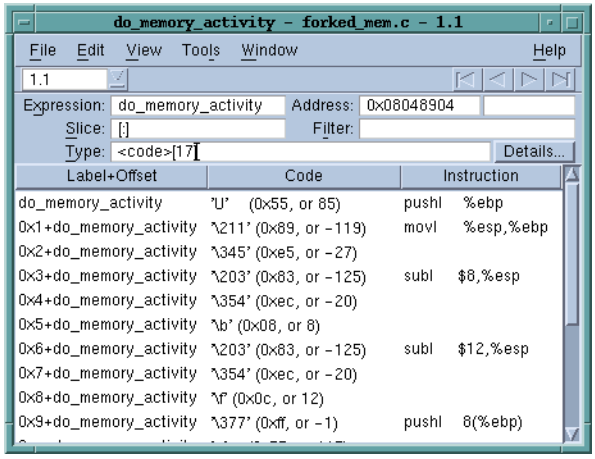
Displaying Machine Instructions

GUI

You can display the machine instructions for entire routines as follows:

- Dive on the address of an assembler instruction in the Source Pane (such as **main+0x10** or **0x60**). A Variable Window displays the instructions for the entire function, and highlights the instruction you dove on.
- Dive on the PC in the Stack Frame Pane. A Variable Window displays the instructions for the entire function that contains the PC, and also highlights the instruction pointed to by the PC.

Figure 162: Variable Window with Machine Instructions



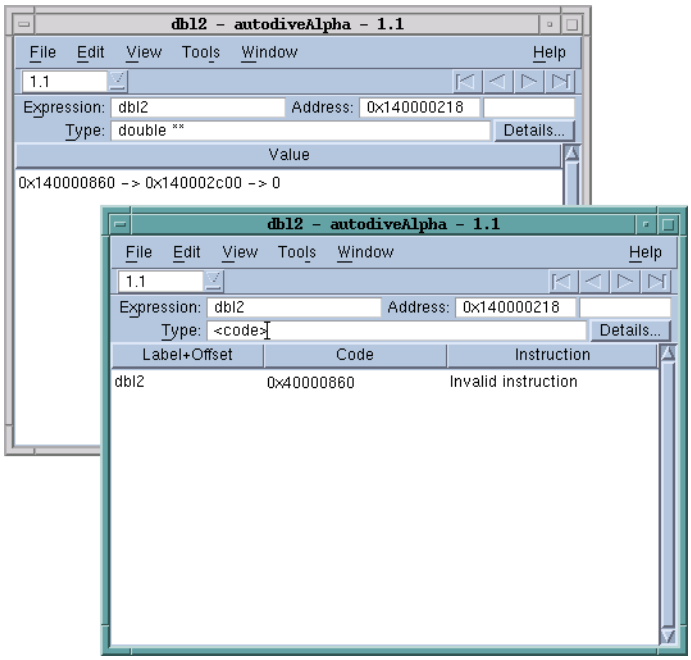
- Cast a variable to type **<code>** or array of **<code>**, as described in *"Changing Types to Display Machine Instructions"* on page 247. (This is shown in the figure on the next page.)

Rebinding the Variable Window

When you restart your program, TotalView must identify the thread in which the variable existed. For example, suppose variable **my_var** was in thread 3.6. When you restart your program, TotalView tries to rebind the thread to a newly created thread. Because the order in which the operating system starts and executes threads can differ, there's no guarantee that the thread 3.6 in the current context is the same thread as what it was previously. Problems can occur. To correct this, use the **Threads** box in the Variable Window toolbar to specify the thread to which you want to bind the variable.

Another way to use the **Threads** box is to change to a different thread to see the variable or expression's value there. For example, suppose variable **my_var** is being displayed in thread 3.4. If you type 3.5 in the Threads box, TotalView updates the information in the **Expression List** Window so that it is what exists in thread 3.5.

Figure 163: Casting to Code



Closing Variable Windows

When you finish analyzing the information in a Variable Window, use the **File > Close** command to close the window. You can also use the **File > Close Similar** command to close all Variable Windows.

Diving in Variable Windows

If the variable being displayed in a Variable Window is a pointer, structure, or array, you can dive on the value. This new dive, which is called a *nested dive*, tells TotalView to replace the information in the Variable Window with information about the selected variable. If this information contains non-scalar data types, you can also dive on these data types. Although a typical data structure doesn't have too many levels, repeatedly diving on data lets you follow pointer chains. That is, diving lets you see the elements of a linked list.

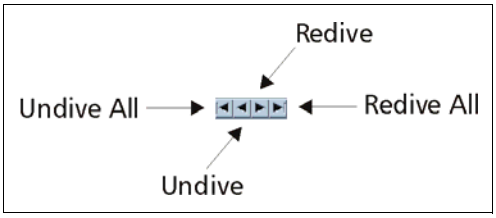
The following topics contain information related to this topic:

- "*Displaying an Array of Structure's Elements*" on page 251
- "*Changing What the Variable Window Displays*" on page 253

TotalView lets you see a member of an array of structures as a single array across all the structures. See "*Displaying an Array of Structure's Elements*" on page 251 for more information.

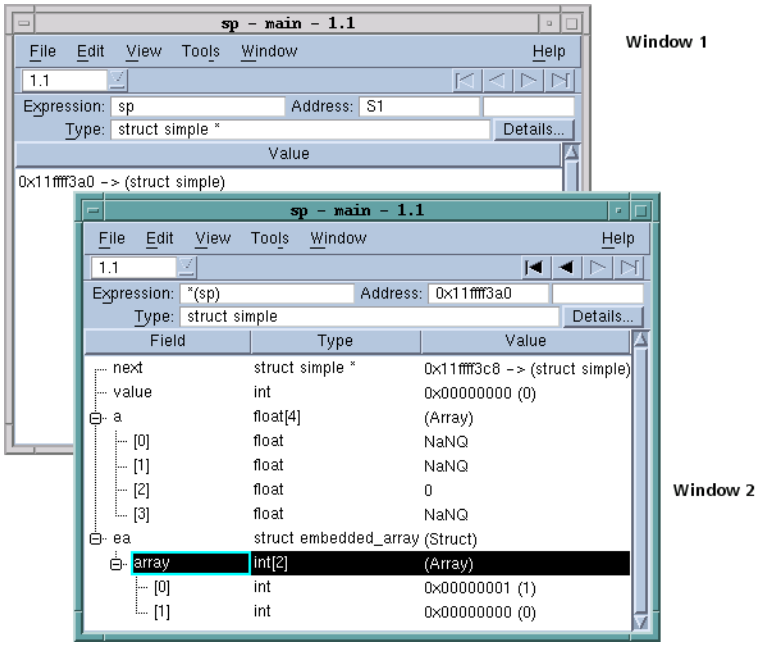
TotalView remembers your dives. This means that you can use the undive/redive buttons to view where you already dove.

Figure 164: Undive/Redive Buttons



The following figure shows a Variable Window after diving into a pointer variable named **sp** with a type of **simple***. The first Variable Window, which is called the *base window*, displays the value of **sp**. (This is **Window 1** in the following figure.)

Figure 165: Nested Dives



The nested dive window, (**Window 2** in this figure) shows the structure referenced by the **simple*** pointer.

You can manipulate Variable Windows and nested dive windows by using the undive/redive buttons, as follows:

- To undive from a nested dive, click the undive arrow button. The previous contents of the Variable Window appear.
- To undive from all your dive operations, click the undive all arrow button.
- To redive after you undive, click the redive arrow button. TotalView restores a previously executed dive operation.
- To redive from all your undive operations, click on the **Redive All** arrow button.

- If you dive on a variable that already has a Variable Window open, the Variable Window pops to the top of the window display.
- If you select the **Window > Duplicate** command, a new Variable Window appears, which is a duplicate of the current Variable Window.

Displaying an Array of Structure's Elements

GUI

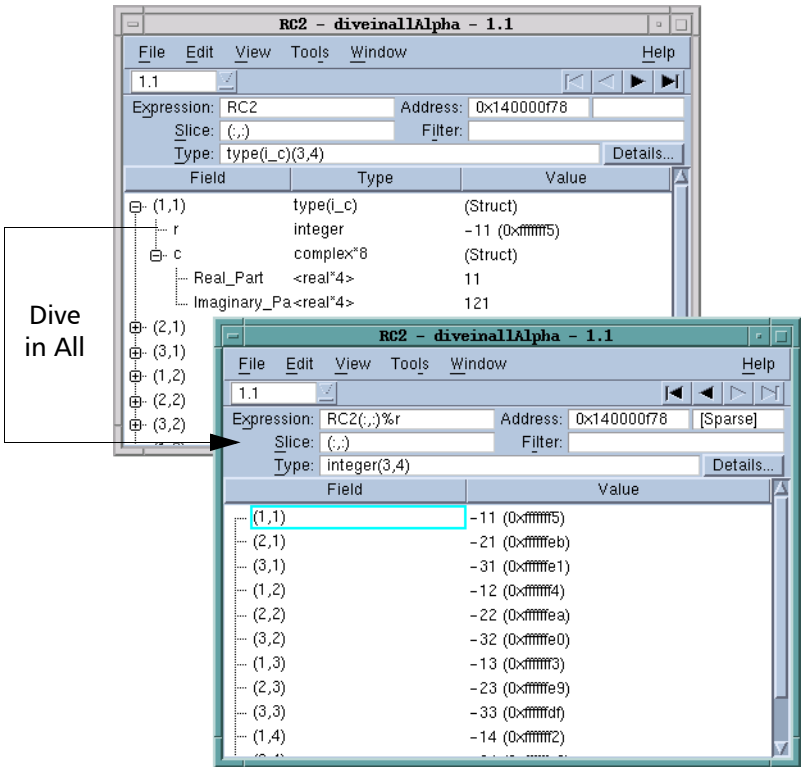
The **View > Dive In All** command, which is also available when you right-click on a field, lets you display an element in an array of structures as if it were a simple array. For example, suppose you have the following Fortran definition:

```
type i_c
   integer r
   complex c
end type i_C

type(i_c), target :: rc2(3,4)
```

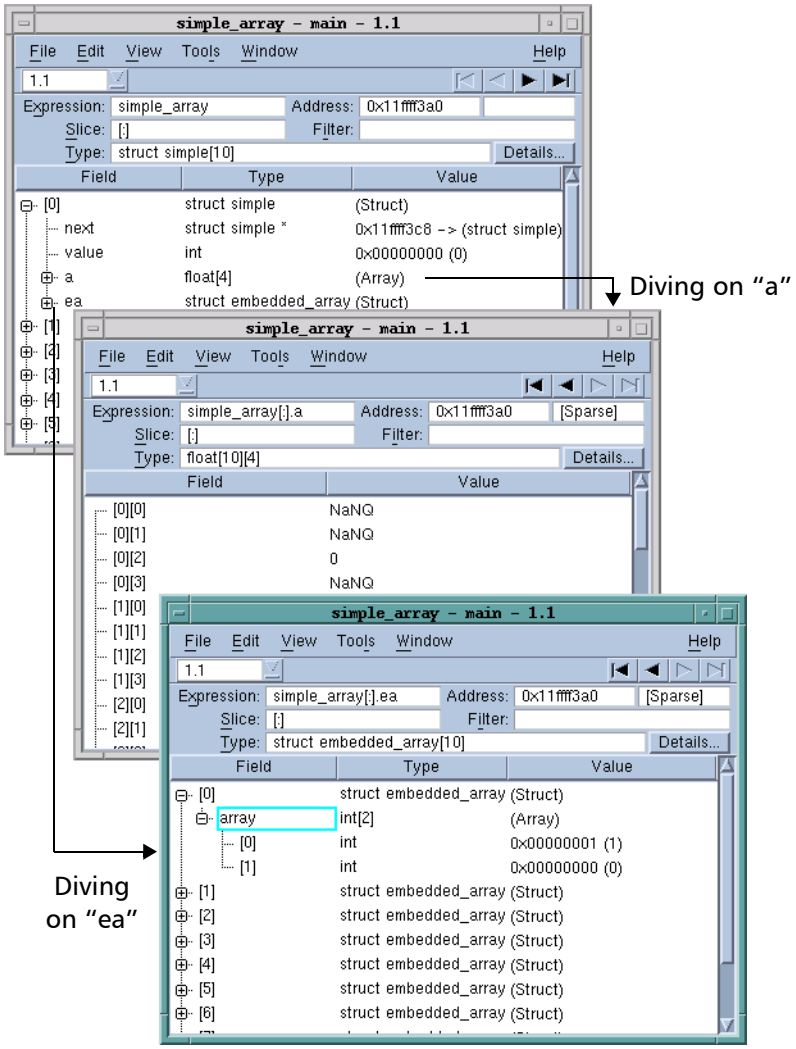
After selecting an **r** element, select the **View > Dive In All** command. TotalView displays all of the **r** elements of the **rc2** array as if it were a single array.

Figure 166: Displaying a Fortran Structure



The **View > Dive in All** command can also display the elements of a C array of structures as arrays. This figure shows a unified array of structures and a multidimensional array in a structure.

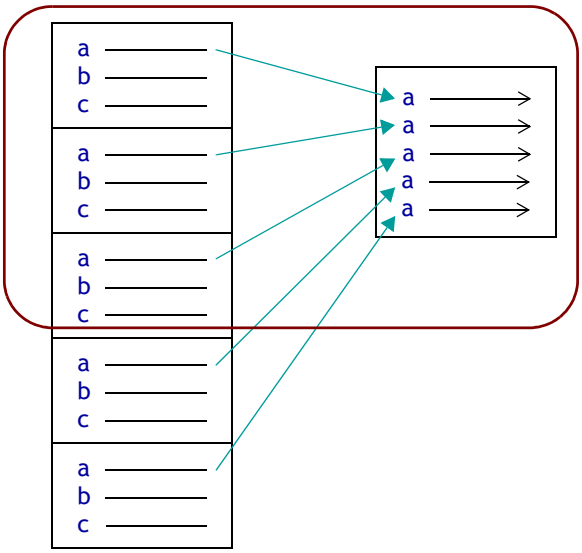
Figure 167: Displaying C Structures and Arrays



As the array manipulation commands (described in Chapter 8) generally work on what's displayed and not what is stored in memory, TotalView commands that refine and display array information work on this virtual array. For example, you can visualize the array, obtain statistics about it, filter elements in it, and so on.

The following figure is a high-level look at what occurred.

Figure 168: Dive in All



In this figure, the rounded rectangle represents a Variable Window. On the left is an array of five structures. After you select the **Dive in All** command with element **a** selected, TotalView *replaces* the contents of your Variable Window with an array that contains all of these **a** elements.

Changing What the Variable Window Displays

When TotalView displays a Variable Window, the **Expression** field contains either a variable or an expression. Technically, a variable is also an expression. For example, **my_var.an_element** is actually an addressing expression. Similarly, **my_var.an_element[10]** and **my_var[10].an_element** are also expressions, since both TotalView and your program must figure out where the data associated with the element resides.

The expression in the **Expression** field is dynamic. That is, you can tell TotalView to evaluate what you enter before trying to obtain a memory address. For example, if you enter **my_var.an_element[i]**, TotalView evaluates the value of **i** before it redisplays your information. A more complicated example is **my_var.an_element[i+1]**. In this example, TotalView must use its internal expression evaluation system to create a value before it retrieves data values.

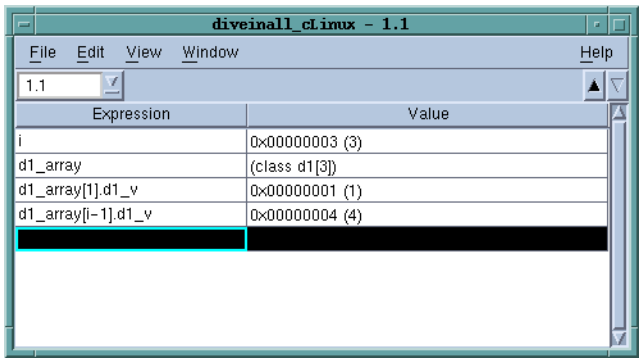
You can replace the variable expression with something completely different, such as **i+1**, and TotalView simply displays the value produced by evaluating the expression.

"*Evaluating Expressions*" on page 319 is a general discussion of the evaluation system and typing expressions in an eval point in the **Tools > Evaluate** Window. In contrast, the expressions you can type in the **Expression List** Window are restricted with the principal restriction being that what you type cannot have side effects. For example, you cannot use an expression that contains a function call or an operator that changes memory, such as ++ or --.

Viewing a List of Variables

As you debug your program, you may want to monitor a variable's value as your program executes. For many types of information, the **Expression List** Window offers a more compact display than the Variable Window for displaying scalar variables. Here's the window:

Figure 169: The Tools > Expression List Window



The topics discussing the **Expression List** Window are:

- "*Entering Variables and Expressions*" on page 254
- "*Entering Expressions into the Expression Column*" on page 255
- "*Using the Expression List with Multiprocess/Multithreaded Programs*" on page 256
- "*Reevaluating, Reopening, Rebinding, and Restarting*" on page 257
- "*Seeing More Information*" on page 257
- "*Sorting, Reordering, and Editing*" on page 258

Entering Variables and Expressions

You can place information in the first column of the **Expression List** window in the following ways:

- Type information into a blank cell in the **Expression** column. When you do this, the context for what you are typing is the current PC in the process and thread indicated in the **Threads** box. If you type **my_var** in the window shown in the previous section, you would type the value of **my_var** in process 1, thread 1.
- Right-click on a line in the Process Window Source or Stack Frame Panes. From the displayed context menu, select **Add to Expression List**. The following figure shows the context menu that TotalView displays in the Source Pane:

Figure 170: A Context Menu

- Right-click on something in a Variable Window. Select **Add to Expression List** from the displayed context menu. You can also use the **View > Add to Expression List** command.

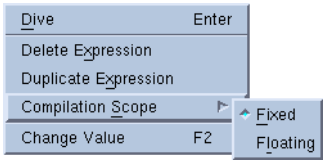
You can bring up this window directly by using the **Tools > Expression List** command.

When you enter information in the **Tools > Expression List** Window, where you place the cursor and what you select make a difference. If you click on a variable or select a row in the Variable Window, TotalView adds that variable to the Expression List Window. If you instead select text, TotalView adds that text. What's the difference? The Expression List figure in the previous section shows three variations of **d1_array**, and each was obtained in a different way, as follows:

- The first entry was added by just selecting part of what was displayed in the Source Pane.
- The second entry was added by selecting a row in the Variable Window.
- The third entry was added by clicking at a random point in the variable's text in the Source Pane.

You can tell TotalView to look for a variable in the scope that exists when your program stops executing, rather than keeping it locked to the scope from which it was added to the **Tools > Expression List** Window. Do this by right-clicking an item, then selecting **Compilation Scope > Floating** from the context menu.

Figure 171: Expression List Window Context Menu

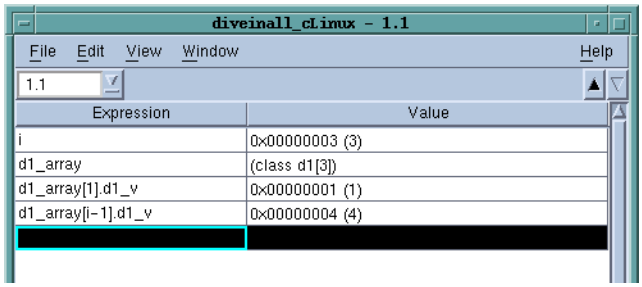


For more information, see "*Viewing a Variable in Different Scopes as Your Program Executes*" on page 242.

Entering Expressions into the Expression Column

The simple answer is just about anything except function calls. (See "*Entering Variables and Expressions*" on page 254 for more information.) A variable is, after all, a type of expression. The following Expression List Window shows four different types of expressions, as follows

Figure 172: The Tools > Expression List Window



The expressions in this window are:

i
A variable with one value. The **Value** column shows its value.

d1_array
An aggregate variable; that is, an array, a structure, a class, and so on. Its value cannot be displayed in one line. Consequently, TotalView just gives you some information about the variable. To see more information, dive on the variable. After diving, TotalView displays the variable in a Variable Window.

When you place an aggregate variable in the **Expression** column, you need to dive on it to get more information.

d1_array[1].d1_v
An element in an array of structures. If TotalView can resolve what you enter in the **Expression** column into a single value, it displays a value in the **Value** column. If TotalView can't, it displays information in the same way that it displays information in the **d1_array** example.

d1_array[i-1].d1_v
An element in an array of structures. This expression differs from the previous example in that the array index is an expression. Whenever execution stops in the current thread, TotalView reevaluates the **i-1** expression. This means that TotalView might display the value of a different array item every time execution stops.

The expressions you enter cannot include function calls.

Using the Expression List with Multiprocess/ Multithreaded Programs

You can change the thread in which TotalView evaluates your expressions by typing a new thread value in the **Threads** box at the top of the window. A second method is to select a value by using the drop-down list in the **Threads** box.

When you use an **Add to Expression List** command, TotalView checks whether an Expression List window is already open for the current thread. If one is open, TotalView adds the variable to the bottom of the list. If an Expression List Window isn't associated with the thread, TotalView duplicates an existing window, changes the thread of the duplicated window, and then adds the variable to all open **Tools > Expression List** Windows. That is, you have two **Tools > Expression List** Windows. Each has the same list of expressions. However, the results of the expression evaluation differ because TotalView is evaluating them in different threads.

In all cases, the list of expressions in all **Tools > Expression List** Windows is the same. What differs is the thread in which TotalView evaluates the window's expressions.

Similarly, if TotalView is displaying two or more **Tools > Expression List** Windows, and you send a variable from yet another thread, TotalView adds the variable to all of them, duplicates one of them, and then changes the thread of the duplicated window.

Reevaluating, Reopening, Rebinding, and Restarting

This section explains what happens in the **Tools** > **Expression List** Window as TotalView performs various operations.

Reevaluating Contents: TotalView reevaluates the value of everything in the **Tools** > **Expression List** Window **Expression** column whenever your thread stops executing. More precisely, if a thread stops executing, TotalView reevaluates the contents of all **Tools** > **Expression List** Windows associated with the thread. In this way, you can see how the values of these expressions change as your program executes.

You can use the **Window** > **Update All** command to update values in all other **Tools** > **Expression List** Windows.

Reopening Windows: If you close all open **Tools** > **Expression List** Windows and then reopen one, TotalView remembers the expressions you add. That is, if the window contains five variables when you close it, it has the same five variables when you open it. The thread TotalView uses to evaluate the window's contents is the Process Window from which you invoked the **Tools** > **Expressions List** command.

Rebinding Windows: The values displayed in an **Expression List** Window are the result of evaluating the expression in the thread indicated in the **Threads** box at the top of the window. To change the thread in which TotalView evaluates these expressions, you can either type a new thread value in the **Threads** box or select a thread from the pulldown list in the **Threads** box. (Changing the thread to evaluate expressions in that thread's context is called *rebinding*.)

Restarting a Program: When you restart your program, TotalView attempts to rebind the expressions in an **Tools** > **Expression List** Window to the *correct* thread. Unfortunately, it is not possible to select the right thread with 100% accuracy. For example, the order in which your operating system creates threads can differ each time you run your program. Or, program logic can cause threads to be opened in a different order.

You may need to manually change the thread by using the **Threads** box at the top of the window.

Seeing More Information

When you first open the **Tools** > **Expression List** Window, it contains two columns, but TotalView can display other columns. If you right-click on a column heading line, TotalView displays a context menu that indicates all possible columns. Clicking on a heading name listed in the context menu changes if from displayed to hidden or vice versa.

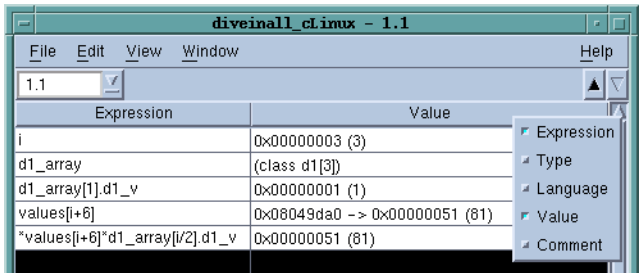


Figure 173: The Tools > Expression List Window Showing Column Selector

Even when you add additional columns, the **Expression List** Window might not show you what you need to know about a variable. If you dive on a row (or select **Dive** from a context menu), TotalView opens a Variable Window for what you just dove on.

You can combine the **Expression List** Window and diving to bookmark your data. For example, you can enter the names of structures and arrays. When you want to see information about them, dive on the name. In this way, you don't have to clutter up your screen with the Variable Windows that you don't need to refer to often.

Sorting, Reordering, and Editing

This section describes operations you can perform on **Tools** > **Expression List** Window data.

Sorting Contents: You can sort the contents of the **Tools** > **Expression List** Window by clicking on the column header. After you click on the heading, TotalView adds an indicator that shows that the column was sorted and the way in which it was sorted. In the figure in the previous topic, the **Value** column is sorted in ascending order.

Reordering Row Display: The up and down arrows (▲▼) on the right side of the **Tools** > **Expression List** Window toolbar let you change the order in which TotalView displays rows. For example, clicking the down arrow moves the currently selected row (indicated by the highlight) one row lower in the display.

Editing Expressions: You can change an expression by clicking in it, and then typing new characters and deleting others. Select **Edit** > **Reset Defaults** to remove all edits you make. When you edit an expression, TotalView uses the scope that existed when you created the variable.

Changing Data Type: You can edit an expression's data type by displaying the **Type** column and making your changes. Select **Edit** > **Reset Defaults** to remove all edits you make.

Changing an Expression's Value: ,You can change an expression's value if that value is stored in memory by editing the contents of the **Value** column.

About Other Commands: You can also use the following commands when working with expressions:

Edit > **Delete Expression**
: Deletes the selected row. This command is also on a context (right-click) menu. If you have more than one **Expression List** Window open, deleting a row from one window deletes the row from all open windows.

Edit > **Delete All Expressions**
: Deletes all of the **Expression List** Window rows. If you have more than one **Expression List** Window open, deleting all expressions from one window deletes all expressions in all windows.

View > **Dive**
: Displays the expression or variable in a Variable Window. Although this command is also on a context

menu, you can just double-click or middle-click on the variable's name instead.

Edit > Duplicate Expression

Duplicates the selected column. You would duplicate a column to see a similar variable or expression. For example, if **myvar_looks_at[i]** is in the **Expression** column, duplicating it and then modifying the new row is an easy way to see **myvar_looks_at[i]** and **myvar_looks_at[i+j-k]** at the same time.

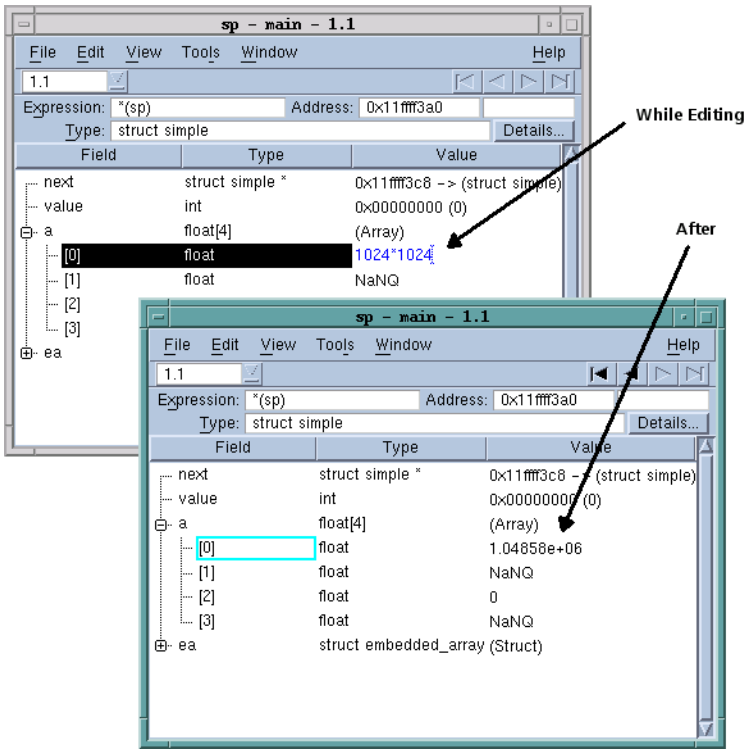
This command is also on a context menu.

Changing the Values of Variables

You can change the value of any variable or the contents of any memory location displayed in a Variable Window, **Expression List** Window, or Stack Frame Pane by selecting the value and typing the new value. In addition to typing a value, you can also type an expression. For example, you can enter **1024*1024** as shown in the following figure. You can include logical operators in all TotalView expressions.

```
CLI:    set my_var [expr 1024*1024]
        dassign int8_array(3) $my_var
```

Figure 174: Using an Expression to Change a Value



In most cases, you can edit a variable's value. If you right-click on a variable and the **Change Value** command isn't faded, you can edit the displayed value.

TotalView does not let you directly change the value of bit fields; you can use the **Tools > Evaluate** Window to assign a value to a bit field. See "*Evaluating Expressions*" on page 319.

> CLI: **Tcl lets you use operators such as & and | to manipulate bit fields on Tcl values.**

Changing a Variable's Data Type

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

The following sections discuss the different aspects of data types:

- "*Displaying C Data Types*" on page 260
- "*Viewing Pointers to Arrays*" on page 261
- "*Viewing Arrays*" on page 261
- "*Viewing typedef Types*" on page 262
- "*Viewing Structures*" on page 262
- "*Viewing Unions*" on page 263
- "*Viewing Built-In Types*" on page 263

You can change the way TotalView displays data in the Variable Window and the **Expression List** Window by editing the data type. This is known as *casting*. TotalView assigns types to all data types, and in most cases, they are identical to their programming language counterparts.

When a C variable is displayed in TotalView, the data types are identical to C type representations, except for pointers to arrays. TotalView uses a simpler syntax for pointers to arrays. (See "*Viewing Pointers to Arrays*" on page 261.) Similarly, when Fortran is displayed in TotalView, the types are identical to Fortran type representations for most data types including **INTEGER**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, **LOGICAL**, and **CHARACTER**.

If the window contains a structure with a list of fields, you can edit the data types of the listed fields.

When you edit a data type, TotalView changes how it displays the variable in the current window. Other windows listing the variable do not change.

Displaying C Data Types

The syntax for displaying data is identical to C language cast syntax for all data types except pointers to arrays. That is, you use C Language cast syntax for **int**, **short**, **unsigned**, **float**, **double**, **union**, and all named **struct** types. In addition, TotalView has a built-in type called **<string>**. Unless you tell it otherwise, TotalView maps **char** arrays to this type.

Read TotalView types from right to left. For example, **<string>*[20]*** is a pointer to an array of 20 pointers to **<string>**.

The following table shows some common TotalView data types:

Data Type String	Description
int	Integer
int*	Pointer to an integer
int[10]	Array of 10 integers
<string>	Null-terminated character string
<string>**	Pointer to a pointer to a null-terminated character string
<string>*[20]*	Pointer to an array of 20 pointers to null-terminated strings

You can enter C Language cast syntax in the **Type** field.

TotalView also lets you cast a variable into an array. In the GUI, add an array specifier to the **Type** declaration. For example, adding **[3]** to a variable declared as an **int** changes it into an array of three **int**s.

When TotalView displays some complex arrays and structures, it displays the compound object or array types in the Variable Window.

Editing a compound object or array data type can produce undesirable results. TotalView tries to give you what you ask for, so if you get it wrong, the results are unpredictable. Fortunately, the remedy is quite simple: close the Variable Window and start over again.

The following sections discuss the following more complex data types.

- "*Viewing Pointers to Arrays*" on page 261
- "*Viewing Arrays*" on page 261
- "*Viewing typedef Types*" on page 262
- "*Viewing Structures*" on page 262
- "*Viewing Unions*" on page 263

Viewing 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**. The C language declaration for this is:

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

Here is how you would cast the **vbl** variable to this type:

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

The TotalView cast for **vbl** is:

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

Viewing Arrays

TotalView Array type names can include a lower and upper bound separated by a colon (:).

See Chapter 13, "Examining Arrays," on page 279 for more information on arrays.

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 ten integers is declared in C and then in Fortran:

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

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

TotalView also lets you cast a variable to an array. In the GUI, just add an array specifier to the **Type** declaration. For example, adding **(3)** to a variable declared as an **integer** changes it to an array of three **integer**s.

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 in the dimension). Consider the following Fortran array declaration:

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

Since both dimensions of this Fortran array use the default lower bound, which is **1**, TotalView displays the data type of the array by using only the extent of each dimension, as follows:

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

If an array declaration doesn't use the default lower bound, TotalView displays both the lower bound and upper bound for each dimension of the array. For example, in Fortran, you declare an array of integers with the first dimension ranging from –1 to 5 and the second dimension ranging from 2 to 10, as follows:

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

TotalView displays this the same way.

When editing an array's dimension, you can enter just the extent (if using the default lower bound), or you can enter the lower and upper bounds separated by a colon.

TotalView also lets you display a subsection of an array, or filter a scalar array for values that match a filter expression. See "*Displaying Array Slices*" on page 279 and "*Filtering Array Data Overview*" on page 283 for more information.

Viewing typedef Types

TotalView recognizes the names defined with **typedef**, and displays these user-defined types; for example:

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

TotalView displays the type for **p_vbl** as **dptr_t**.

Viewing Structures

TotalView lets you use the **struct** keyword as part of a type string. In most cases, this is optional.

This behavior depends upon which compiler you are using. In most cases, you'll see what is described here.

If you have a structure and another data type with the same name, however, you must include the **struct** keyword so that TotalView can distinguish between the two data types.

If you use a **typedef** statement to name a structure, TotalView uses the **typedef** name as the type string. Otherwise, the debugger uses the structure tag for the **struct**.

For example, consider the following structure definition:

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

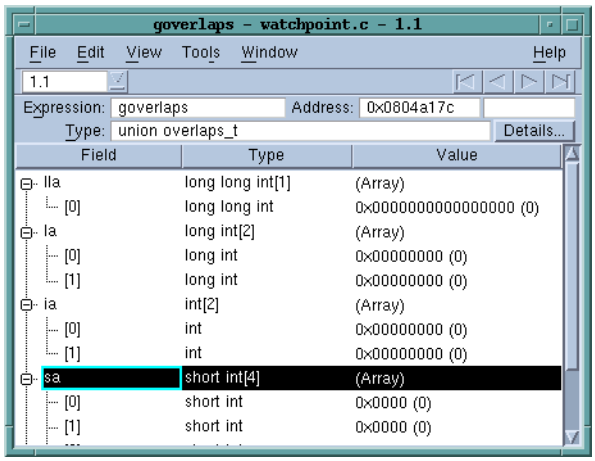
TotalView displays **mystruc_type** as the type for **struct mystruc_struct**.

Viewing Unions

TotalView displays a union in the same way that it displays a structure. Even though the fields of a union are overlaid in storage, TotalView displays the fields on separate lines.

CLI: **dprint** *variable*

Figure 175: Displaying a Union



Viewing Built-In Types

TotalView provides a number of predefined types. These types are enclosed in angle brackets (<>) to avoid conflict with types contained in a programming language. You can use these built-in types anywhere you can use the ones defined in your programming language. These types are also useful in debugging executables with no debugging symbol table information. The following table describes the built-in types:

Type String	Language	Size	Description
<address>	C	void*	Void pointer (address).
<char>	C	char	Character.
<character>	Fortran	character	Character.

Type String	Language	Size	Description
<code>	C	architecture-dependent	Machine instructions. The size used is the number of bytes required to hold the shortest instruction for your computer.
<complex>	Fortran	complex	Single-precision floating-point complex number. The **complex** types contain a real part and an imaginary part, which are both of type **real**.
<complex*8>	Fortran	complex*8	A **real*4**-precision floating-point complex number. The **complex*8** types contain a real part and an imaginary part, which are both of type **real*4**.
<complex*16>	Fortran	complex*16	A **real*8**-precision floating-point complex number. The **complex*16** types contain a real part and an imaginary part, which are both of type **real*8**.
<double>	C	double	Double-precision floating-point number.
<double precision>	Fortran	double precision	Double-precision floating-point number.
<extended>	C	long double	Extended-precision floating-point number. Extended-precision numbers must be supported by the target architecture.
<float>	C	float	Single-precision floating-point number.
<int>	C	int	Integer.
<integer>	Fortran	integer	Integer.
<integer*1>	Fortran	integer*1	One-byte integer.
<integer*2>	Fortran	integer*2	Two-byte integer.
<integer*4>	Fortran	integer*4	Four-byte integer.
<integer*8>	Fortran	integer*8	Eight-byte integer.
<logical>	Fortran	logical	Logical.
<logical*1>	Fortran	logical*1	One-byte logical.
<logical*2>	Fortran	logical*2	Two-byte logical.
<logical*4>	Fortran	logical*4	Four-byte logical.
<logical*8>	Fortran	logical*8	Eight-byte logical.
<long>	C	long	Long integer.
<long long>	C	long long	Long long integer.

Type String	Language	Size	Description
<real>	Fortran	real	Single-precision floating-point number. When using a value such as real, be careful that the actual data type used by your computer is not real*4 or real*8, since different results can occur.
<real*4>	Fortran	real*4	Four-byte floating-point number.
<real*8>	Fortran	real*8	Eight-byte floating-point number.
<real*16>	Fortran	real*16	Sixteen-byte floating-point number.
<short>	C	short	Short integer.
<string>	C	char	Array of characters.
<void>	C	long	Area of memory.

Viewing Opaque Data (<opaque> Data Type)

An opaque type is a data type that isn't fully specified, is hidden, or whose declaration is deferred. For example, the following C declaration defines the data type for **p** as a pointer to **struct foo**, which is not yet defined:

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

When TotalView encounters this kind of information, it might indicate that **foo**'s data type is **<opaque>**; for example:

```
struct foo <opaque>
```

Viewing Character Arrays (<string> Data Type)

If you declare a character array as **char vbl**[*n*], TotalView automatically changes the type to **<string>**[*n*]; that is, a null-terminated, quoted string with a maximum length of *n*. This means that TotalView displays an array as a quoted string of *n* characters, terminated by a null character. Similarly, TotalView changes **char*** declarations to **<string>*** (a pointer to a null-terminated string).

Since most C character arrays represent strings, the TotalView **<string>** type can be very convenient. But if this isn't what you want, you can edit the **<string>** and change it back to a **char** (or **char**[*n*]), to display the variable as you declared it.

Viewing Areas of Memory (<void> Data Type)

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

If you dive on registers or display an area of memory, TotalView lists the contents as a **<void>** data type. If you display an array of **<void>** vari-

ables, the index for each object in the array is the address, not an integer. This address can be useful when you want to display large areas of memory.

If you want, you can change a <void> to another type. Similarly, you can change any type to a <void> to see the variable in decimal and hexadecimal formats.

Viewing Instructions (<code> Data Type)

TotalView uses the <code> data type to display the contents of a location as machine instructions. To look at disassembled code stored at a location, dive on the location and change the type to <code>. To specify a block of locations, use <code>[*n*], where *n* is the number of locations being displayed.

Type-Casting Examples

This section contains three type-casting examples:

- Displaying Declared Arrays
- Displaying Allocated Arrays
- Displaying the argv Array

Displaying Declared Arrays

TotalView displays arrays the same way it displays local and global variables. In the Stack Frame or Source Pane, dive on the declared array. A Variable Window displays the elements of the array.

CLI: dprint array-name

Displaying Allocated Arrays

The C Language uses pointers for dynamically allocated arrays; for example:

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

Since TotalView doesn't know that p actually points to an array of integers, you need to do several things to display the array:

1. Dive on the variable p 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 the argv Array

Typically, argv is the second argument passed to main(), and it is either a char **argv or char *argv[ ]. Suppose argv points to an array of three pointers to character strings. Here is how you can edit its type to display an array of three pointers, as follows:

1. Select the type string for argv.

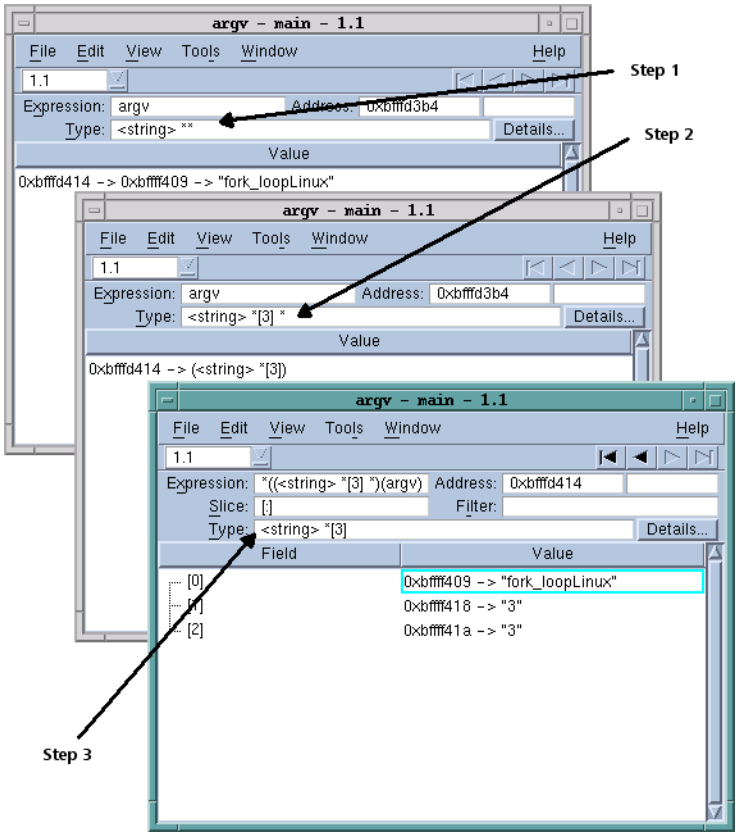
CLI: dprint argv

2 Edit the type string by using the field editor commands. Change it to:
`<string>*[3]*`

CLI:	dprint (<string>*\[3\]*)argv

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

Figure 176: Editing the argv Argument



Changing the Address of Variables

You can edit the address of a variable in a Variable Window by editing the value shown in the **Address** field. When you edit this address, the Variable Window shows the contents of the new location.

You can also enter an address expression, such as **0x10b8 – 0x80** in this area.

Displaying C++ Types

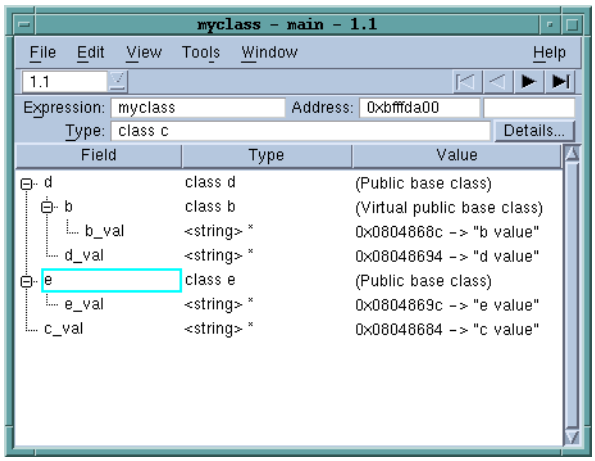
Viewing Classes

TotalView displays C++ classes and accepts **class** as a keyword. When you debug C++, TotalView also accepts the *unadorned* name of a **class**, **struct**, **union**, or **enum** in the type field. TotalView displays nested classes that use inheritance, showing derivation by indentation.

Some C++ compilers do not write accessibility information. In these cases, TotalView cannot display this information.

For example, the following figure displays an object of a **class c**.

Figure 177: Displaying C++ Classes That Use Inheritance



Its definition is as follows:

```
class b {
   char * b_val;
public:
   b() {b_val = “b value“;}
};

class d : virtual public b {
   char * d_val;
public:
   d() {d_val = “d value“;}
};

class e {
   char * e_val;
 public:
   e() {e_val = “e value“;}
};

class c : public d, public e {
   char * c_val;
 public:
   c() {c_val = “c value“;}
};
```

TotalView tries to display the correct data when you change the type of a Variable Window while moving up or down the derivation hierarchy. Unfortunately, many compilers do not contain the information that TotalView needs so you might need to cast your class.

Displaying Fortran Types

TotalView lets you display FORTRAN 77 and Fortran 90 data types.

The topics in this section describe the various types and how TotalView handles them:

- "*Displaying Fortran Common Blocks*" on page 269
- "*Displaying Fortran Module Data*" on page 269
- "*Debugging Fortran 90 Modules*" on page 271
- "*Viewing Fortran 90 User-Defined Types*" on page 272
- "*Viewing Fortran 90 Deferred Shape Array Types*" on page 272
- "*Viewing Fortran 90 Pointer Types*" on page 273
- "*Displaying Fortran Parameters*" on page 273

Displaying Fortran Common Blocks

For each common block defined in the scope of a subroutine or function, TotalView creates an entry in that function's common block list. The Stack Frame Pane displays the name of each common block for a function. The names of common block members have function scope, not global scope.

CLI: **dprint** *variable-name*

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 the figure on the next page.

Window 1 in this figure shows a common block list in a Stack Frame Pane. After several dives, **Window 2** contains the results of diving on the common block.

If you dive on a common block member name, TotalView searches all common blocks in the function's scope for a matching member name, and displays the member in a Variable Window.

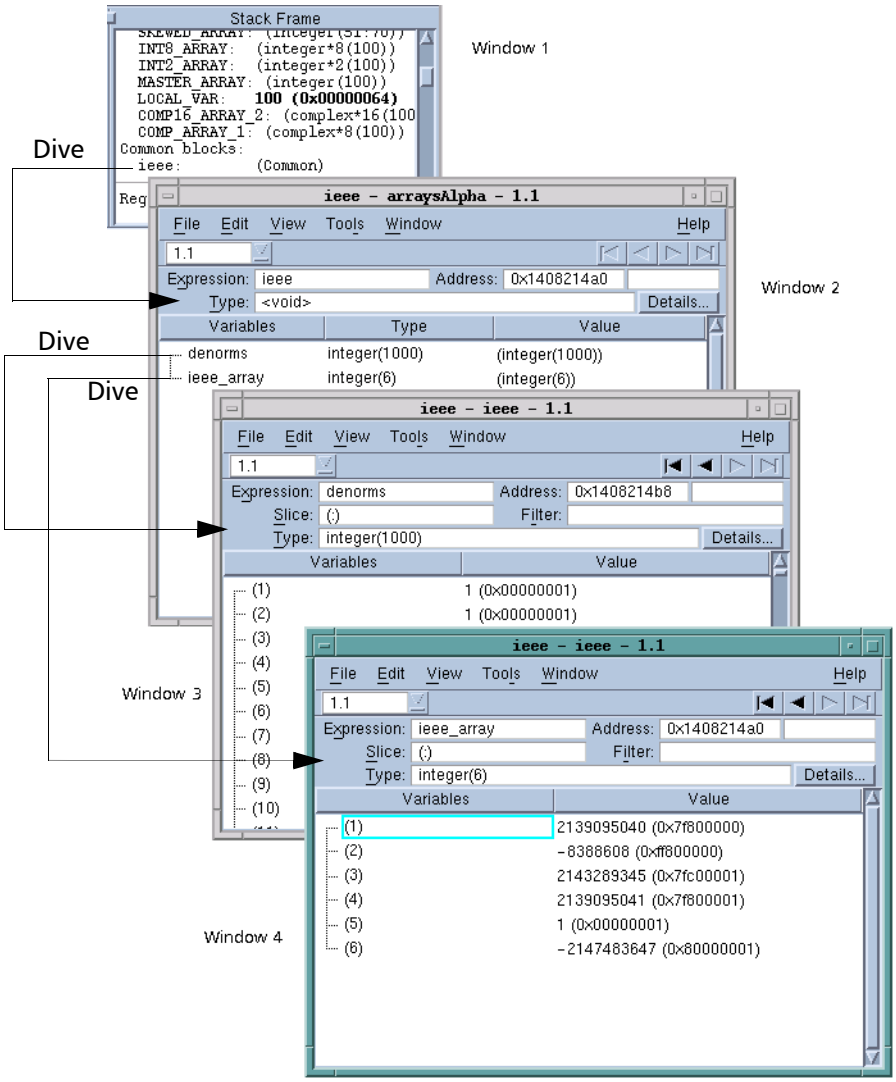
Displaying Fortran Module Data

TotalView tries to locate all data associated with a Fortran module and display it all at once. For functions and subroutines defined in a module, TotalView adds the full module data definition to the list of modules displayed in the Stack Frame Pane.

TotalView only displays a module if it contains data. Also, the amount of information that your compiler gives TotalView can restrict what's displayed.

Although a function may use a module, TotalView doesn't always know if the module was used or what the true names of the variables in the module are. If this happens, either of the following occurs:

Figure 178: Diving on a Common Block List in the Stack Frame Pane



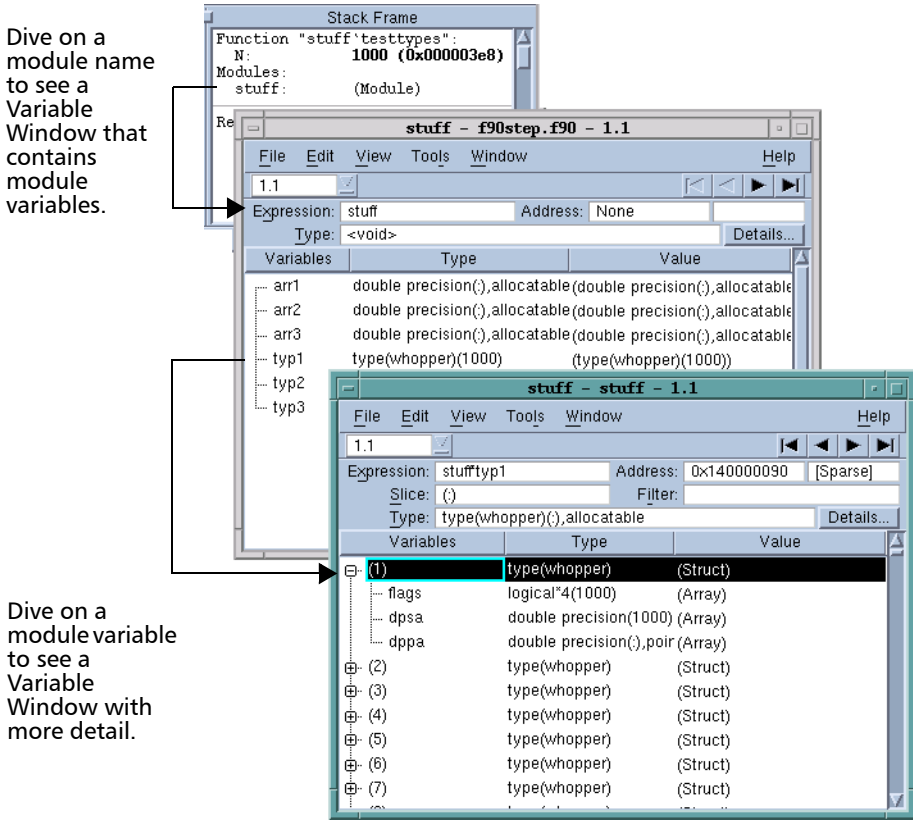
- Module variables appear as local variables of the subroutine.
- A module appears on the list of modules in the Stack Frame Pane that contains (with renaming) only the variables used by the subroutine.

CLI: **dprint** *variable-name*

Alternatively, you can view a list of all the known modules by using the **Tools > Fortran Modules** command. Because Fortran modules display in a Variable Window, you can dive on an entry to display the actual module data, as shown in the figure on the next page.

If you are using the SUNPro *compiler, TotalView can only display module data if you force it to read the debug information for a file that contains the module definition or a module function. For more information, see "Finding the Source Code for Functions" on page* 181.

Figure 179: Fortran Modules Window



Debugging Fortran 90 Modules

Fortran 90 lets you place functions, subroutines, and variables inside modules. You can then include these modules elsewhere by using a **USE** command. When you do this, the names in the module become available in the *using* compilation unit, unless you either exclude them with a **USE ONLY** statement or rename them. This means that you don't need to explicitly qualify the name of a module function or variable from the Fortran source code.

When debugging this kind of information, you need to know the location of the function being called. Consequently, TotalView uses the following syntax when it displays a function contained in a module:

modulename`functionname

You can also use this syntax in the **File > New Program** and **View > Lookup Variable** commands.

Fortran 90 also lets you create a contained function that is only visible in the scope of its parent and siblings. There can be many contained functions in a program, all using the same name. If the compiler gave TotalView the function name for a nested function, TotalView displays it using the following syntax:

parentfunction()`*containedfunction*

CLI: **dprint** *module_name`variable_name*

Viewing Fortran 90 User-Defined Types

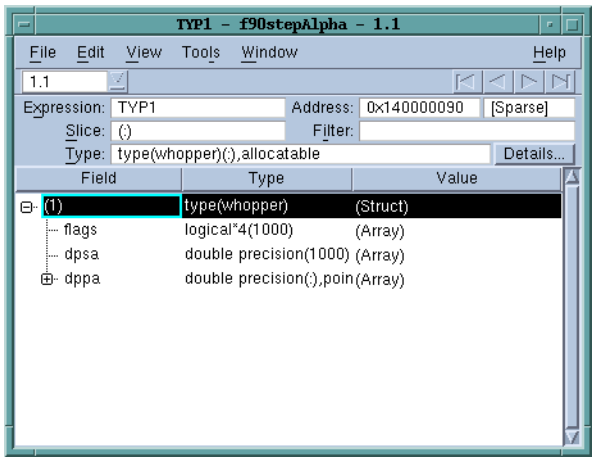
A Fortran 90 user-defined type is similar to a C structure. TotalView displays a user-defined type as **type**(*name*), which is the same syntax used in Fortran 90 to create a user-defined type. For example, the following code fragment defines a variable **typ2** of **type**(*whopper*):

```
TYPE WHOPPER
   LOGICAL, DIMENSION(ISIZE) :: FLAGS
   DOUBLE PRECISION, DIMENSION(ISIZE) :: DPSA
   DOUBLE PRECISION, DIMENSION(:), POINTER :: DPPA
END TYPE WHOPPER

TYPE(WHOPPER), DIMENSION(:), ALLOCATABLE :: TYP2
```

TotalView displays this type, as the following figure shows:

Figure 180: Fortran 90 User-Defined Type



Viewing Fortran 90 Deferred Shape Array Types

Fortran 90 lets you define deferred shape arrays and pointers. The actual bounds of a deferred shape array are not determined until the array is allocated, the pointer is assigned, or, in the case of an assumed shape argument to a subroutine, the subroutine is called. TotalView displays the type of deferred shape arrays as **type(:)**.

When TotalView displays the data for a deferred shape array, it displays the type used in the definition of the variable and the actual type that this instance of the variable has. The actual type is not editable, since you can achieve the same effect by editing the definition's type. The following example shows the type of a deferred shape rank 2 array of **real** data with runtime lower bounds of **–1** and **2**, and upper bounds of **5** and **10**:

```
        Type: real(:,:)
Actual Type: real(-1:5,2:10)
       Slice: (:,:)
```

Viewing Fortran 90 Pointer Types

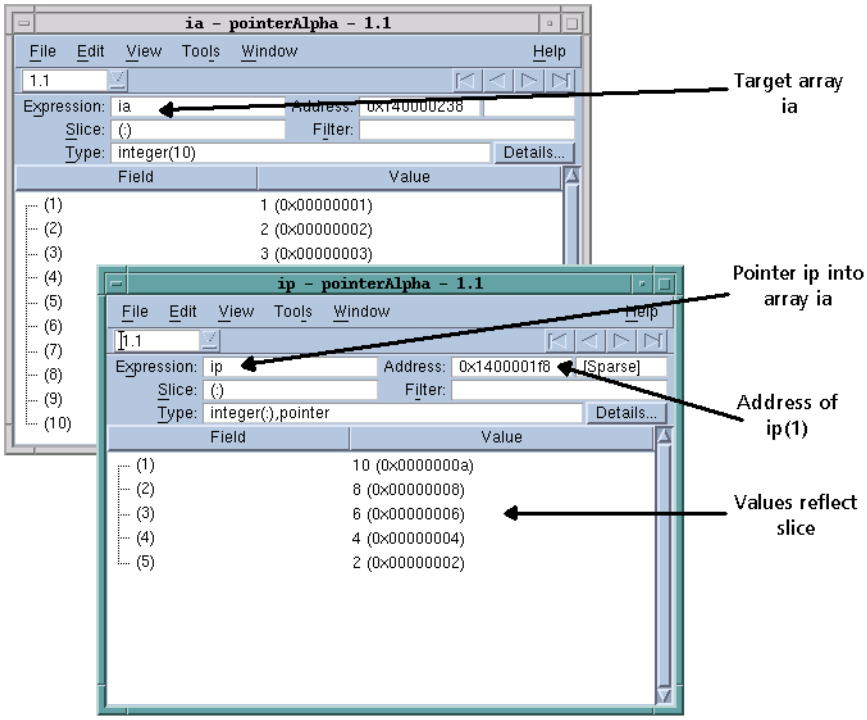
A Fortran 90 pointer type lets you point to scalar or array types.

TotalView implicitly handles slicing operations that set up a pointer or assumed shape subroutine argument so that indices and values it displays in a Variable Window are the same as in the Fortran code; for example:

```
integer, dimension(10), target :: ia
integer, dimension(:), pointer :: ip
do i = 1,10
   ia(i) = i
end do
ip => ia(10:1:-2)
```

After diving through the **ip** pointer, TotalView displays the windows shown in the following figure:

Figure 181: Fortran 90 Pointer Value



The address displayed is not that of the array's base. Since the array's stride is negative, array elements that follow are at lower absolute addresses. Consequently, the address displayed is that of the array element that has the lowest index. This might not be the first displayed element if you used a slice to display the array with reversed indices.

Displaying Fortran Parameters

A Fortran **PARAMETER** defines a named constant. Most compilers do not generate information that TotalView can use to determine the value of a **PARAMETER**. This means that you must make a few changes to your program if you want to see this type of information.

If you're using Fortran 90, you can define variables in a module that you initialize to the value of these **PARAMETER** constants; for example:

```
INCLUDE 'PARAMS.INC'

MODULE CONSTS
   SAVE
   INTEGER PI_C = PI
   ...
END MODULE CONSTS
```

The **PARAMS.INC** file contains your parameter definitions. You then use these parameters to initialize variables in a module. After you compile and link this module into your program, the values of these parameter variables are visible.

If you're using FORTRAN 77, you can achieve the same results if you make the assignments in a common block and then include the block in **main()**. You can also use a block data subroutine to access this information.

Displaying Thread Objects

On HP Alpha Tru64 UNIX and IBM AIX systems, TotalView can display information about mutexes and conditional variables. In addition, TotalView can display information on read/write locks and data keys on IBM AIX. You can obtain this information by selecting the **Tools > Thread Objects** command. After selecting this command, TotalView displays a window that contains either two tabs (HP Alpha) or four tabs (IBM). The figure on the next page shows some AIX examples.

Diving on any line in these windows displays a Variable Window that contains additional information about the item.

Here are some things you should know:

- If you're displaying data keys, many applications initially set keys to **0** (the NULL pointer value). TotalView doesn't display a key's information, however, until a thread sets a non-NULL value to the key.
- If you select a thread ID in a data key window, you can dive on it using the **View > Dive Thread** and **View > Dive Thread New** commands to display a Process Window for that thread ID.

The online Help contains information on the contents of these windows.

Scoping and Symbol Names

TotalView assigns a unique name to every element in your program based on the scope in which the element exists. A *scope* defines the part of a program that knows about a symbol. For example, the scope of a variable that is defined at the beginning of a subroutine is all the statements in the subroutine. The variable's scope does not extend outside of this subroutine. A program consists of multiple *scopes*. Of course, a block contained in the

Figure 182: Thread Objects Page on an IBM AIX Computer

subroutine could have its own definition of the same variable. This would *hide* the definition in the enclosing scope.

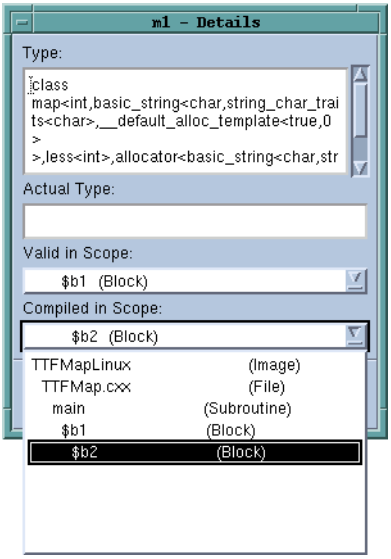
All scopes are defined by your program's structure. Except for the simplest of programs, scopes are embedded in other scopes. The only exception is the outermost scope, which is the one that contains **main()**, which is not embedded. Every element in a program is associated with a scope.

To see the scope in which a variable is valid, click the **Details** button in the Variable Window. The dialog box shown on the next page appears.

The **Valid in Scope** list indicates the scope in which the variable resides. That is, when this scope is active, the variable is defined. The **Compiled in Scope** list can differ if you modify the variable with an expression. It indicates where variables in this expression have meaning.

When you tell the CLI or the GUI to execute a command, TotalView consults the program's symbol table to discover which object you are referring to—this process is known as *symbol lookup*. Symbol lookup is performed with respect to a particular context, and each context uniquely identifies the scope to which a symbol name refers.

Figure 183: Details Dialog Box



For additional information, see "*Scoping Issues*" on page 242.

Qualifying Symbol Names

The way you describe a scope is similar to the way you specify a file. The scopes in a program form a tree, with the outermost scope (which is your program) as the root. At the next level are executable files and dynamic libraries; further down are compilation units (source files), procedures, modules, and other scoping units (for example, blocks) supported by the programming language. Qualifying a symbol is equivalent to describing the path to a file in UNIX file systems.

A symbol is fully scoped when you name all levels of its tree. The following example shows how to scope a symbol and also indicates parts that are optional:

[#*executable-or-lib*#][*file*#][*procedure-or-line*#]*symbol*

The pound sign (**#**) separates elements of the fully qualified name.

Because of the number of different types of elements that can appear in your program, a complete description of what can appear and their possible order is complicated and unreadable. In contrast, after you see a name in the Stack Frame Pane, it is easy to read a variable's scoped name.

TotalView interprets most programs and components as follows:

- You do not need to qualify file names with a full path, and you do not need to use all levels in a symbol's scoping tree. TotalView conventions here are similar to the way UNIX displays file names.
- If a qualified symbol begins with #, the name that follows indicates the name of the executable or shared library (just as an absolute file path begins with a directory immediately in the root directory). If you omit the executable or library component, the qualified symbol doesn't begin with #.

- The source file's name can appear after the possibly omitted executable or shared library.
- Because programming languages typically do not let you name blocks, that portion of the qualifier is specified using the symbols **$b** followed by a number that indicates which block. For example, the first unnamed block is named **$b1**, the second is **$b2**, and so on.

You can omit any part of the scope specification that TotalView doesn't need to uniquely identify the symbol. Thus, **foo#x** identifies the symbol **x** in the procedure **foo**. In contrast, **#foo#x** identifies either procedure **x** in executable **foo** or variable **x** in a scope from that executable.

Similarly, **#foo#bar#x** could identify variable **x** in procedure **bar** in executable **foo**. If **bar** were not unique in that executable, the name would be ambiguous unless you further qualified it by providing a file name. Ambiguities can also occur if a file-level variable (common in C programs) has the same name as variables declared in functions in that file. For instance, **bar.c#x** refers to a file-level variable, but the name can be ambiguous when there are different definitions of **x** embedded in functions that occur in the same file. In this case, you need to enter **bar.c#b1#x** to identify the scope that corresponds to the outer level of the file (that is, the scope that contains line 1 of the file).

Examining Arrays 13

This chapter explains how to examine and change array data as you debug your program. Since arrays also appear in the Variable Window, you need to be familiar with the information in Chapter 12, "*Examining and Changing Data*," on page 235.

The topics in this chapter are:

- "*Examining and Analyzing Arrays*" on page 279
- "*Displaying a Variable in all Processes or Threads*" on page 290
- "*Visualizing Array Data*" on page 292

Examining and Analyzing Arrays

TotalView can quickly display very large arrays in Variable Windows. An array can be the elements that you define in your program, or it can be an area of memory that you cast into an array.

If an array extends beyond the memory section in which it resides, the initial portion of the array is formatted correctly. If memory isn't allocated for an array element, TotalView displays **Bad Address** in the element's subscript.

Topics in this section are:

- "*Displaying Array Slices*" on page 279
- "*Filtering Array Data Overview*" on page 283
- "*Sorting Array Data*" on page 288
- "*Obtaining Array Statistics*" on page 289

Displaying Array Slices

TotalView lets you display array subsections by editing the **Slice** field in an array's Variable Window. (An array subsection is called a *slice*.) The **Slice** field contains placeholders for all array dimensions. For example, the following is a C declaration for a three-dimensional array:

```
integer an_array[10][20][5]
```

Because this is a three-dimensional array, the initial slice definition is [:][:][:]. This lets you know that the array has three dimensions and that TotalView is displaying all array elements.

The following is a deferred shape array definition for a two-dimensional array variable:

```
integer, dimension (:,:) :: another_array
```

The TotalView slice definition is (:,:).

TotalView displays as many colons (:) as there are array dimensions. For example, the slice definition for a one-dimensional array (a vector) is [:] for C arrays and (:) for Fortran arrays.

CLI: dprint an_array\[n:m,p:q\]
dprint an_array(n:m,p:q)

Using Slices and Strides

A slice has the following form:

lower_bound:*upper_bound*[:*stride*]

The *stride*, which is optional, tells TotalView to skip over elements and not display them. Adding a *stride* to a slice tells TotalView to display every *stride* element of the array, starting at the *lower_bound* and continuing through the *upper_bound*, inclusive.

For example, a slice of [0:9:9] used on a ten-element C array tells TotalView to display the first element and last element, which is the ninth element beyond the lower bound.

If the stride is negative and the lower bound is greater than the upper bound, TotalView displays a dimension with its indices reversed. That is, TotalView treats the slice as if it was defined as follows:

[*upperbound* : *lowerbound* : *stride*]

CLI: dprint *an_array*(*n*:*m*:*p*,*q*:*r*:*s*)

For example, the following definition tells TotalView to display an array beginning at its last value and moving to its first:

```
[::-1]
```

This syntax differs from Fortran 90 syntax in that Fortran 90 requires that you explicitly enter the upper and lower bounds when you're reversing the order for displaying array elements.

Because the default value for the stride is 1, you can omit the stride (and the colon that precedes it) from your definition. For example, the following two definitions display array elements 0 through 9:

```
[0:9:1]
[0:9]
```

If the lower and upper bounds are the same, just use a single number. For example, the following two definitions tell TotalView to display array element 9:

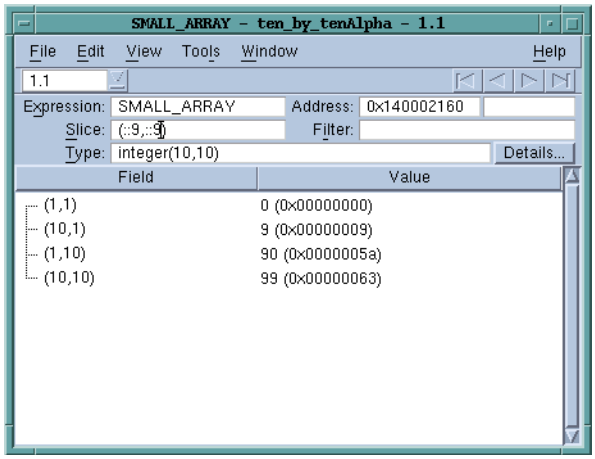
```
[9:9:1]
[9]
```

The lower_bound, upper_bound, and stride must be constants. They cannot be expressions.

Example 1 A slice declaration of **[::2]** for a C or C++ array (with a default lower bound of **0**) tells TotalView to display elements with even indices of the array; that is, 0, 2, 4, and so on. However, if this were defined for a Fortran array (where the default lower bound is **1**), TotalView displays elements with odd indices of the array; that is, 1, 3, 5, and so on.

Example 2 The following figure displays a stride of **(::9,::9)**. This definition displays the four corners of a ten-element by ten-element Fortran array.

Figure 184: Stride Displaying the Four Corners of an Array



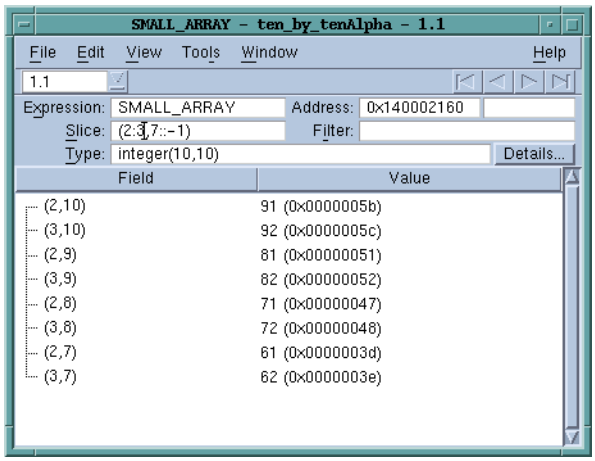
Example 3 You can use a stride to invert the order *and* skip elements. For example, the following slice begins with the upper bound of the array and displays every other element until it reaches the lower bound of the array:

```
(::-2)
```

Using **(::–2)** with a Fortran **integer(10)** array tells TotalView to display the elements 10, 8, 6, 4, and 2.

Example 4 You can simultaneously invert the array's order and limit its extent to display a small section of a large array. The following figure shows how to specify a **(2:3,7::–1)** slice with an **integer*4(–1:5,2:10)** Fortran array.

Figure 185: Fortran Array with Inverse Order and Limited Extent



After you enter this slice value, TotalView only shows elements in rows 2 and 3 of the array, beginning with column 10 and ending with column 7.

Using Slices in the Lookup Variable Command

When you use the **View > Lookup Variable** command to display a Variable Window, you can include a slice expression as part of the variable name. Specifically, if you type an array name followed by a set of slice descriptions in the **View > Lookup Variable** command dialog box, TotalView initializes the **Slice** field in the Variable Window to this slice description.

If you add subscripts to an array name in the **View > Lookup Variable** command dialog box, TotalView interprets these subscripts as a slice description rather than as a request to display an individual value of the array. Therefore, you can display different values of the array by changing the slice expression.

For example, suppose that you have a ten-element by ten-element Fortran array named **small_array**, and you want to display element **(5,5)**. Using the **View > Lookup Variable** command, type **small_array(5,5)**. This sets the initial slice to **(5:5,5:5)**, as shown in **Window 1** of the following figure.

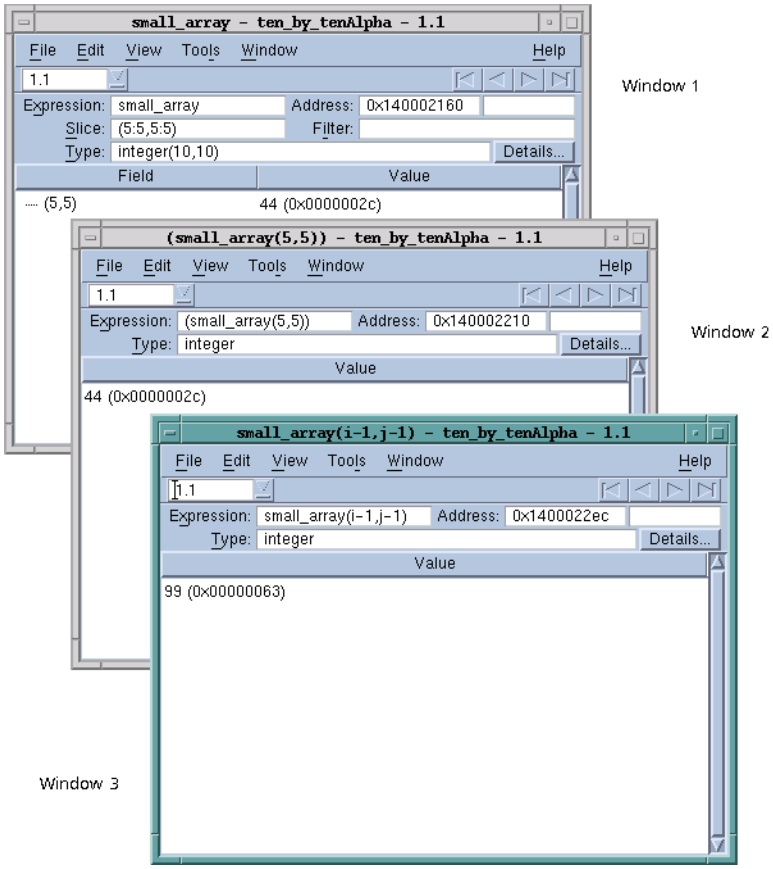
CLI: dprint small_array(5,5)

You can tell TotalView to display one of the array's values by enclosing the array name and subscripts (that is, the information normally included in a slice expression) within parentheses, such as **(small_array(5,5))**.

CLI: dprint (small_array(5,5))

In this case, the Variable Window just displays the type and value of the element and doesn't show its array index. This is shown in **Window 2** of the following figure.

Figure 186: Variable Window for small_array



Window 3 was created by using the **View > Lookup Variable** command with a value of **small_array(i-1,j-1)**. Here, TotalView evaluated the values of **i** and **j** before displaying the window.

Filtering Array Data Overview

GUI

You can restrict what TotalView displays in a Variable Window by adding a filter to the window. You can filter arrays of type character, integer, or floating point. Your filtering options are:

- Arithmetic comparison to a constant value
- Equal or not equal comparison to IEEE NaNs, Infs, and Denorms
- Within a range of values, inclusive or exclusive
- General expressions

When an element of an array matches the filter expression, TotalView includes the element in the Variable Window display.

The following topics describe filtering options:

- "*Filtering Array Data*" on page 284
- "*Filtering by Comparison*" on page 284
- "*Filtering for* IEEE *Values*" on page 285

- "*Filtering a Range of Values*" on page 286
- "*Creating Array Filter Expressions*" on page 287
- "*Using Filter Comparisons*" on page 288

Filtering Array Data

The procedure for filtering an array is simple: select the **Filter** field, enter the array filter expression, and then press Enter.

TotalView updates the Variable Window to exclude elements that do not match the filter expression. TotalView only displays an element if its value matches the filter expression and the slice operation.

If necessary, TotalView converts the array element before evaluating the filter expression. The following conversion rules apply:

- If the filter operand or array element type is floating point, TotalView converts the operand to a double-precision floating-point value. TotalView truncates extended-precision values to double precision. Converting integer or unsigned integer values to double-precision values might result in a loss of precision. TotalView converts unsigned integer values to nonnegative double-precision values.
- If the filter operand or the array element is an unsigned integer, TotalView converts the operand to an unsigned 64-bit integer.
- If both the filter operand and array element are of type integer, TotalView converts the values to type 64-bit integer.

TotalView conversion operations modify a copy of the array's elements—conversions never alter the actual array elements.

To stop filtering an array, delete the contents of the **Filter** field in the Variable Window and press Enter. TotalView then updates the Variable Window so that it includes all elements.

Filtering by Comparison

The simplest filters are ones whose formats are as follows:

operator value

where *operator* is either a C/C++ or Fortran-style comparison operator, and *value* is a signed or unsigned integer constant or a floating-point number. For example, the filter for displaying all values greater than 100 is:

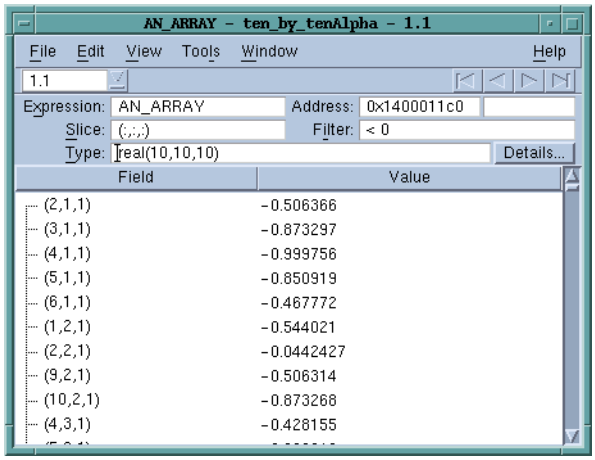
```
> 100
```

The following table lists the comparison operators:

Comparison	C/C++ Operator	Fortran Operator
Equal	==	.eq.
Not equal	!=	.ne.
Less than	<	.lt.
Less than or equal	<=	.le.
Greater than	>	.gt.
Greater than or equal	>=	.ge.

The following figure shows an array whose filter is **< 0**. This tells TotalView to only display array elements whose value is less than 0 (zero).

Figure 187: Array Data Filtering by Comparison



If the *value* you're using in the comparison is an integer constant, TotalView performs a signed comparison. If you add the letter **u** or **U** to the constant, TotalView performs an unsigned comparison.

Filtering for IEEE Values

You can filter IEEE NaN, Infinity, or denormalized floating-point values by specifying a filter in the following form:

operator ieee-tag

The only comparison operators you can use are *equal* and *not equal*.

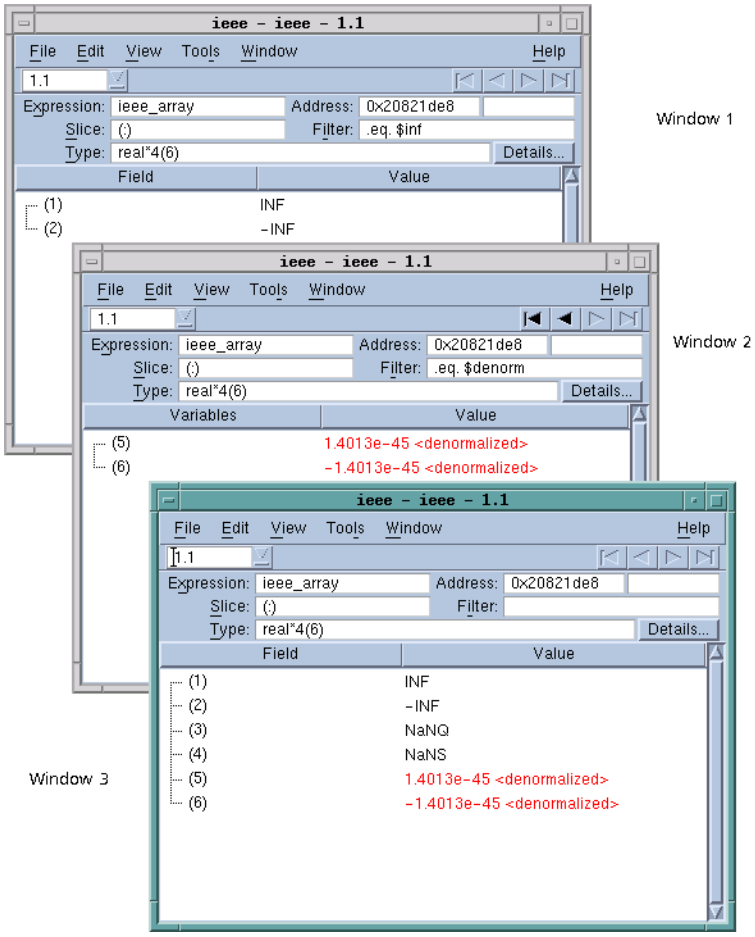
The *ieee-tag* represents an encoding of IEEE floating-point values, as the following table describes:

IEEE Tag Value	Meaning
$nan	NaN (Not a number), either quiet or signaling
$nanq	Quiet NaN
$nans	Signaling NaN
$inf	Infinity, either positive or negative
$pinf	Positive Infinity
$ninf	Negative Infinity
$denorm	Denormalized number, either positive or negative
$pdenorm	Positive denormalized number
$ndenorm	Negative denormalized number

The following figure shows an example of filtering an array for IEEE values. **Window 3** in this figure shows how TotalView displays the unfiltered array. Notice the NaNQ, and NaNS, INF, and –INF values. The other two windows

show filtered displays: **Window 1** shows only infinite values; **Window 2** shows only the values of denormalized numbers.

Figure 188: Array Data Filtering for IEEE *Values*



Filtering a Range of Values

You can also filter array values by specifying a range, as follows:

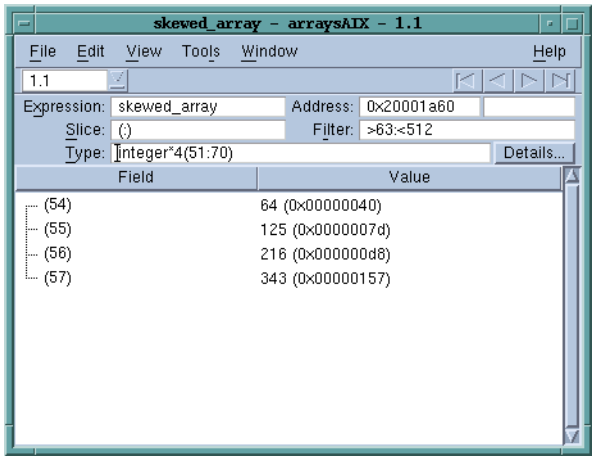
[>] *low-value* : [<] *high-value*

where *low-value* specifies the lowest value to include, and *high-value* specifies the highest value to include, separated by a colon. The high and low values are inclusive unless you use less-than (<) and greater-than (>) symbols. If you specify a > before *low-value*, the low value is exclusive. Similarly, a < before *high-value* makes it exclusive.

The values of *low-value* and *high-value* must be constants of type integer, unsigned integer, or floating point. The data type of *low-value* must be the same as the type of *high-value*, and *low-value* must be less than *high-value*. If *low-value* and *high-value* are integer constants, you can append the letter u or U to the value to force an unsigned comparison. The following figure

shows a filter that tells TotalView that to only display values greater than 63, but less than 512.

Figure 189: Array Data Filtering by Range of Values



Creating Array Filter Expressions

The filtering capabilities described in the previous sections are those that you use most often. In some circumstances, you may need to create a more general expression. When you create a filter expression, you're creating a Fortran or C Boolean expression that TotalView evaluates for every element in the array or the array slice. For example, the following expression displays all array elements whose contents are greater than 0 and less than 50, or greater than 100 and less than 150:

```
($value > 0 && $value < 50) ||
         ($value > 100 && $value < 150)
```

Here's the Fortran equivalent:

```
($value .gt. 0 && $value .lt. 50) .or.
         ($value .gt. 100 .and. $value .lt.150)
```

The **$value** variable is a special TotalView variable that represents the current array element. You can use this value when creating expressions.

Notice how the **and** and **or** operators are used in these expressions. The way in which TotalView computes the results of an expression is identical to the way it computes values at an eval point. For more information, see "*Defining Eval Points and Conditional Breakpoints*" on page 306.

You cannot use any of the IEEE *tag values described in* "*Filtering for* IEEE *Values*" *on page* 285 *in array filter expressions.*

Using Filter Comparisons

TotalView provides several different ways to filter array information. For example, the following two filters display the same array items:

```
> 100
$value > 100
```

The following filters display the same array items:

```
>0:<100
$value > 0 && $value < 100
```

The only difference is that the first method is easier to type than the second, so you're more likely to use the second method when you're creating more complicated expressions.

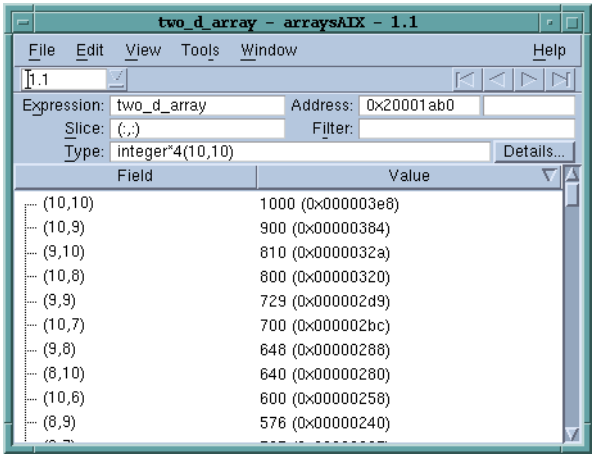
Sorting Array Data

TotalView lets you sort the displayed array data into ascending or descending order. (It does not sort the actual data.) To sort (or remove the sort), click the **Value** label.

- The first time you click, TotalView sorts the array's values into ascending order.
- The next time you click on the header, TotalView reverses the order, sorting the array's values into descending order.
- If you click again on the header, TotalView returns the array to its unsorted order.

Here is an example that sorts an array into descending order:

Figure 190: Sorted Variable Window



When you sort an array's values, you are just rearranging the information that's displayed in the Variable Window. Sorting does not change the order in which values are stored in memory. If you alter what TotalView is displaying by using a filter or a slice, TotalView just sorts the values that could be displayed; it doesn't sort all of the array.

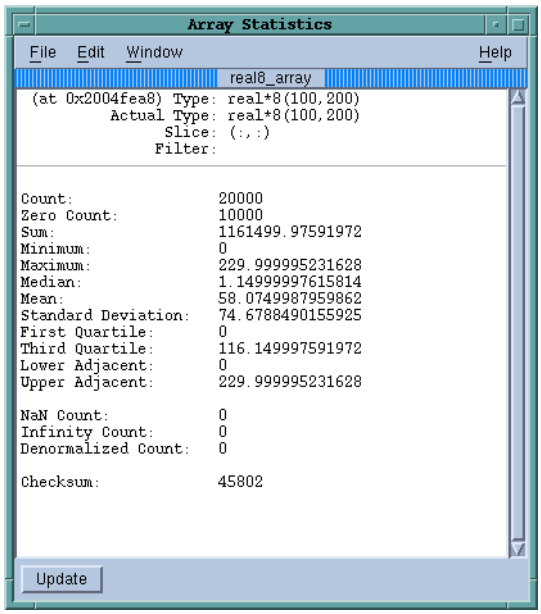
If you are displaying a laminated array—see "*Displaying a Variable in all Processes or Threads*" on page 290 for more information—you can sort your information by process or thread.

Obtaining Array Statistics

GUI

The **Tools** > **Statistics** command displays a window that contains information about your array. The following figure shows an example.

Figure 191: Array Statistics Window



If you have added a filter or a slice, these statistics describe only the information currently being displayed; they do not describe the entire unfiltered array. For example, if 90% of an array's values are less than 0 and you filter the array to show only values greater than 0, the median value is positive even though the array's real median value is less than 0.

TotalView displays the following statistics:

- **Checksum**

  A checksum value for the array elements.
- **Count**

  The total number of displayed array values. If you're displaying a floating-point array, this number doesn't include NaN or Infinity values.
- **Denormalized Count**

  A count of the number of denormalized values found in a floating-point array. This includes both negative and positive denormalized values as defined in the IEEE floating-point standard. Unlike other floating-point statistics, these elements participate in the statistical calculations.
- **Infinity Count**

  A count of the number of infinity values found in a floating-point array. This includes both negative and positive infinity as defined in the IEEE floating-point standard. These elements don't participate in statistical calculations.
- **Lower Adjacent**

  This value provides an estimate of the lower limit of the distribution. Values below this limit are called *outliers*. The lower adjacent value is the first

quartile value minus the value of 1.5 times the difference between the first and third quartiles.

- **Maximum**

  The largest array value.
- **Mean**

  The average value of array elements.
- **Median**

  The middle value. Half of the array's values are less than the median, and half are greater than the median.
- **Minimum**

  The smallest array value.
- **NaN Count**

  A count of the number of NaN (not a number) values found in a floating-point array. This includes both signaling and quiet NaNs as defined in the IEEE floating-point standard. These elements don't participate in statistical calculations.
- **Quartiles, First and Third**

  Either the 25th or 75th percentile values. The first quartile value means that 25% of the array's values are less than this value and 75% are greater than this value. In contrast, the third quartile value means that 75% of the array's values are less than this value and 25% are greater.
- **Standard Deviation**

  The standard deviation for the array's values.
- **Sum**

  The sum of all the displayed array's values.
- **Upper Adjacent**

  This value provides an estimate of the upper limit of the distribution. Values above this limit are called *outliers*. The upper adjacent value is the third quartile value plus the value of 1.5 times the difference between the first and third quartiles.
- **Zero Count**

  The number of elements whose value is 0.

Displaying a Variable in all Processes or Threads

When you're debugging a parallel program that is running many instances of the same executable, you usually need to view or update the value of a variable in all of the processes or threads at once.

Before displaying a variable's value in all threads or processes, you must display an instance of the variable in a Variable Window. After TotalView displays this window, use one of the following commands:

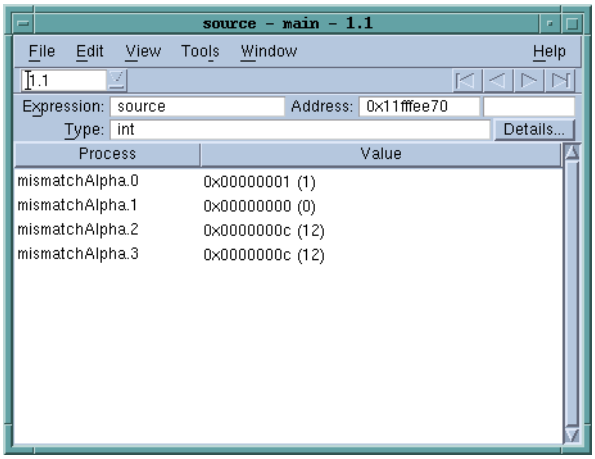
- View > Laminate > Process, displays the value of the variable in all processes.
- View > Laminate > Thread, displays the value of a variable in all threads within a single process.

- **View** > **Laminate** > **None**, delaminates the window if you no longer want it laminated..

You cannot simultaneously laminate across processes and threads in the same Variable Window.

After using one of these commands, the Variable Window switches to laminated mode, and displays the value of the variable in each process or thread. The following figure shows a simple, scalar variable in each of the processes in an OpenMP program.

Figure 192: Laminated Scalar Variable



When looking for a matching stack frame, TotalView matches frames starting from the top frame, and considers calls from different memory or stack locations to be different calls. For example, the following definition of **recurse** contains two additional calls to **recurse**. Each of these calls generates a nonmatching call frame.

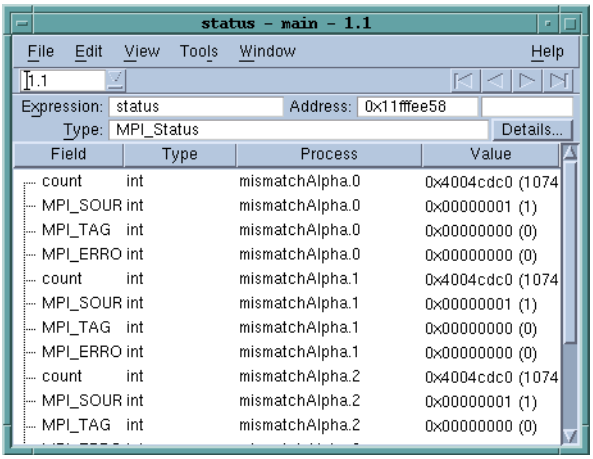
```
void recurse(int i) {
   if (i <= 0)
      return;
   if (i & 1)
      recurse(i - 1);
   else
      recurse(i - 1);
}
```

If the variables are at different addresses in the different processes or threads, the field to the left of the **Address** field displays **Multiple,** and the unique addresses appear with each data item.

TotalView also lets you laminate arrays and structures. When you laminate an array, TotalView displays each element in the array across all processes. You can use a slice to select elements to be displayed in laminated win-

dows. The following figure shows an example of a laminated array and a laminated structure. You can also laminate an array of structures.

Figure 193: Laminated Array and Structure



Diving on a Laminated Pane

GUI

You can dive through pointers in a laminated Variable Window. This dive applies to the associated pointer in each process or thread.

Editing a Laminated Variable

GUI

If you edit a value in a laminated Variable Window, TotalView asks if it should apply this change to all processes or threads or only the one in which you made a change. This is an easy way to update a variable in all processes.

GUI

Visualizing Array Data

The TotalView Visualizer lets you create graphical images of array data. This presentation lets you see your data in one glance and can help you quickly find problems with your data while you are debugging your programs.

You can execute the Visualizer from within TotalView, or you can run it from the command line to visualize data dumped to a file in a previous TotalView session.

For information about running the TotalView Visualizer, see Chapter 7, "*Visualizing Programs and Data*," on page 137.

Visualizing a Laminated Variable Window

GUI

You can export data from a laminated Variable Window to the Visualizer by using the **Tools > Visualize** command. When visualizing laminated data, the process (or thread) index is the first axis of the visualization. This means that you must use one less data dimension than you normally would. If you do not want the process/thread axis to be significant, you can use a normal Variable Window, since all of the data must be in one process.

Setting Action Points 14

This chapter explains how to use action points. TotalView supports four kinds of action points:

- A *breakpoint* stops execution of processes and threads that reach it.
- A *barrier point* synchronizes a set of threads or processes at a location.
- An *eval point* causes a code fragment to execute when it is reached.
- A *watchpoint* lets you monitor a location in memory and stop execution when it changes.

This chapter contains the following sections:

- "*About Action Points*" on page 293
- "*Setting Breakpoints and Barriers*" on page 295
- "*Defining Eval Points and Conditional Breakpoints*" on page 306
- "*Using Watchpoints*" on page 313
- "*Saving Action Points to a File*" on page 319
- "*Evaluating Expressions*" on page 319
- "*Writing Code Fragments*" on page 322

About Action Points

Actions points lets you specify an action for TotalView to perform when a thread or process reaches a source line or machine instruction in your program.The different kinds of action points that you can use:

- **Breakpoints**

  When a thread encounters a breakpoint, it stops at the breakpoint. Other threads in the process also stops. You can indicate that you want other related processes to stop, as well. Breakpoints are the simplest kind of action point.

- **Barrier points**

  Barrier points are similar to simple breakpoints, differing in that you use them to synchronize a group of processes or threads. A barrier point holds each thread or process that reaches it until all threads or processes

reach it. Barrier points work together with the TotalView hold-and-release feature. TotalView supports thread barrier and process barrier points.

- **Eval points**

  An eval point is a breakpoint that has a code fragment associated with it. When a thread or process encounters an eval point, it executes this code. You can use eval points in a variety of ways, including conditional breakpoints, thread-specific breakpoints, countdown breakpoints, and patching code fragments into and out of your program.

- **Watchpoints**

  A watchpoint tells TotalView to either stop the thread so that you can interact with your program (unconditional watchpoint), or evaluate an expression (conditional watchpoint).

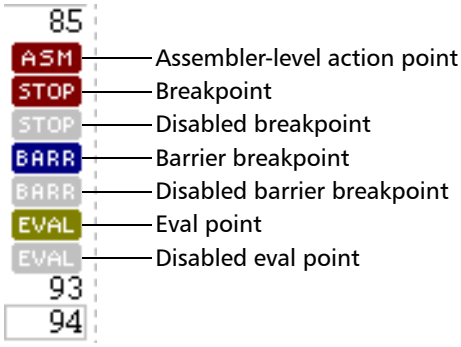
All action points share the following common properties.

- You can independently enable or disable action points. A disabled action isn't deleted; however, when your program reaches a disabled action point, TotalView ignores it.
- You can share action points across multiple processes, or set them in individual processes.
- Action points apply to the process, so in a multithreaded process, the action point applies to all of the threads contained in the process.
- TotalView assigns unique ID numbers to each action point. These IDs appear in several places, including the Root Window, the Action Points Pane of the Process Window, and the **Action Point > Properties** Dialog Box.

The following figure shows the symbol that TotalView displays for an action point:.

CLI: dactions shows information about action points.

Figure 194: Action Point Symbols



The ASM icon is what TotalView displays when you create a breakpoint on an assembler statement.

CLI: All action points display as "@" when you use the dlist command to display your source code. Use the dactions command to see what type of action point is set.

Setting Breakpoints and Barriers

TotalView has several options for setting breakpoints. You can set:

- Source-level breakpoints
- Breakpoints that are shared among all processes in multiprocess programs
- Assembler-level breakpoints

You can also control whether TotalView stops all processes in the control group when a single member reaches a breakpoint.

Topics in this section are:

- "*Setting Source-Level Breakpoints*" on page 295
- "*Setting and Deleting Breakpoints at Locations*" on page 295
- "*Displaying and Controlling Action Points*" on page 297
- "*Setting Machine-Level Breakpoints*" on page 299
- "*Setting Breakpoints for Multiple Processes*" on page 300
- "*Setting Breakpoints When Using the fork()/execve() Functions*" on page 302
- "*Setting Barrier Points*" on page 303

Setting Source-Level Breakpoints

Typically, you set and clear breakpoints before you start a process. To set a source-level breakpoint, select a boxed line number in the Process Window. (A boxed line number indicates that the line is associated with executable code.) A STOP icon lets you know that a breakpoint is set immediately before the source statement.

CLI: @ next to the line number

You can also set a breakpoint while a process is running by selecting a boxed line number in the Process Window.

CLI: Use dbreak whenever the CLI displays a prompt.

GUI

Choosing Source Lines

If you're using C++ templates, TotalView sets a breakpoint in all instantiations of that template. If this isn't what you want, clear the button and then select the **Addresses** button in the Action Point Properties Dialog Box. You can now clear locations where the action point shouldn't be set. (This is shown on the next page.)

Similarly, in a multiprocess program, you might not want to set the breakpoint in all processes. If this is the case, select the **Process** button.

Setting and Deleting Breakpoints at Locations

You can set or delete a breakpoint at a specific function or source-line number without having to first find the function or source line in the Source Pane. Do this by entering a line number or function name in the **Action Point > At Location** Dialog Box. (This dialog box is shown on the next page.)

Figure 195: Setting Breakpoints on Multiple Similar Addresses and on Processes

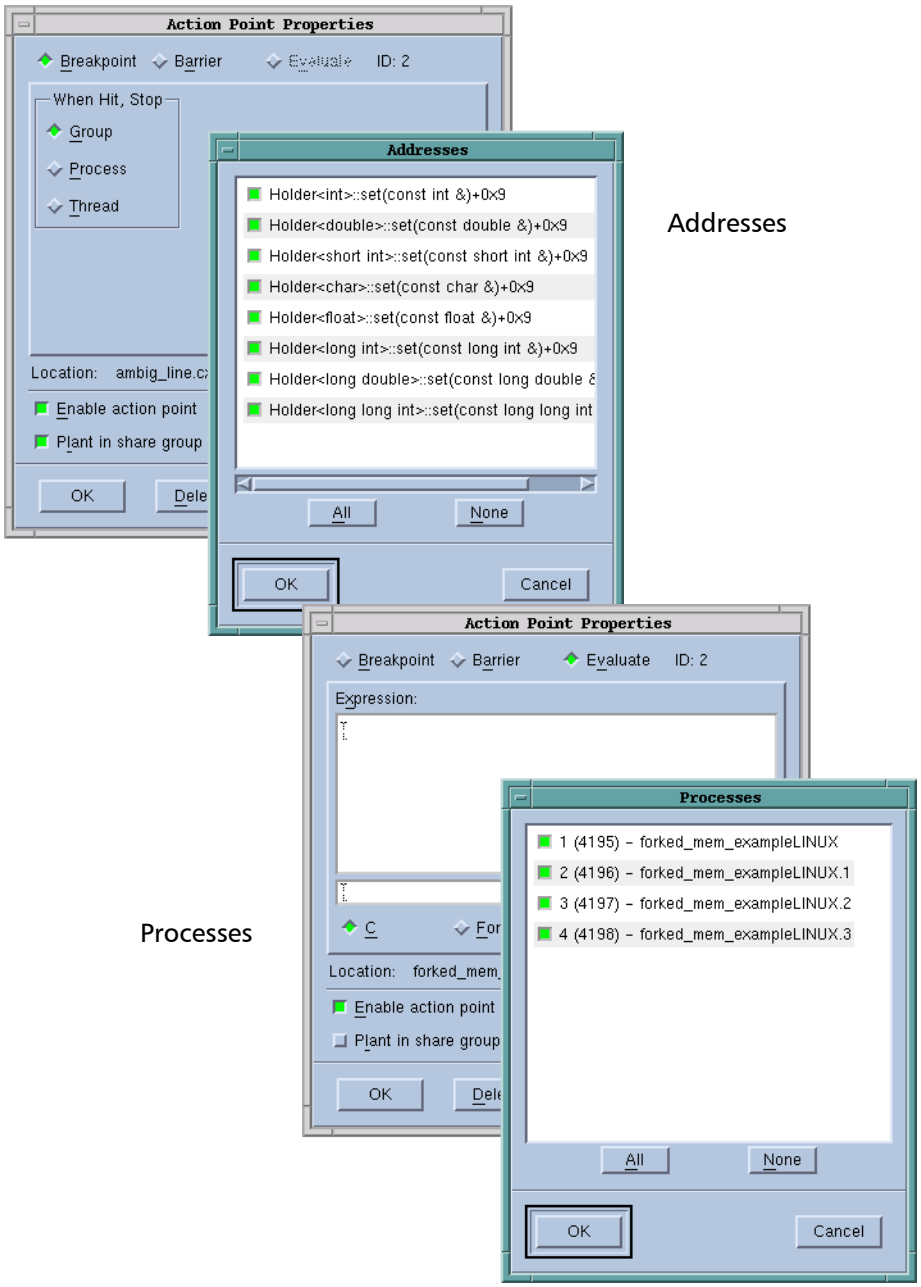
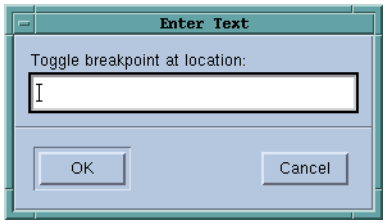


Figure 196: Action Point > At Location Dialog Box



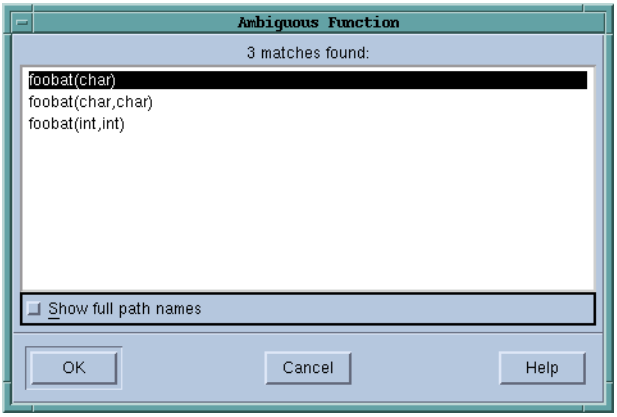
When you're done, TotalView sets a breakpoint at the location. If you enter a function name, TotalView sets the breakpoint at the function's first exe-

cutable line. In either case, if a breakpoint already exists at a location, TotalView deletes it.

CLI:	dbreak sets a breakpoint
	ddelete deletes a breakpoint

If you enter an ambiguous function name using the **Action Point > At Location** command, TotalView displays its **Ambiguous Function** Dialog Box.

Figure 197: Ambiguous Function Dialog Box

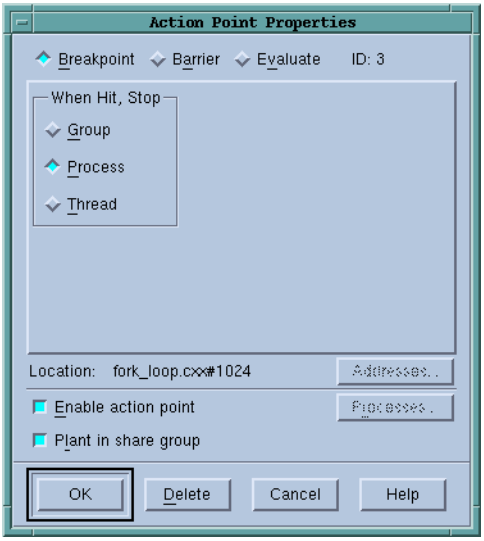


After you select the function at which TotalView will set a breakpoint, press the **OK** button.

Displaying and Controlling Action Points

The **Action Point > Properties** Dialog Box lets you set and control an action point. Controls in this dialog box also lets you change an action point's type to breakpoint, barrier point, or eval point. You can also define what happens to other threads and processes when execution reaches this action point.

Figure 198: Action Point > Properties Dialog Box



The following sections explain how you can control action points by using the Process Window and the **Action Point** > **Properties** Dialog Box.

CLI: dset SHARE_ACTION_POINT
dset STOP_ALL
ddisable *action-point*

Disabling Action Points

TotalView can retain an action point's definition and ignore it while your program is executing. That is, disabling an action point deactivates it without removing it.

CLI: ddisable *action-point*

You can disable an action point by:

- Clearing **Enable action point** in the **Action Point** > **Properties** Dialog Box.
- Selecting the STOP or BARR symbol in the Action Points Pane.
- Using the context (right-click) menu.
- Clicking on the **Action Points** > **Disable** command.

Deleting Action Points

You can permanently remove an action point by selecting the STOP or BARR symbol or selecting the **Delete** button in the **Action Point** > **Properties** Dialog Box.

To delete all breakpoints and barrier points, use the **Action Point** > **Delete All** command.

CLI: ddelete

If you make a significant change to the action point, TotalView disables it rather than delete it when you click the symbol.

Enabling Action Points

You can activate an action point that was previously disabled by selecting a dimmed STOP, BARR, or EVAL symbol in the Source or Action Points Pane, or by selecting **Enable action point** in the **Action Point** > **Properties** Dialog Box.

CLI: denable

Suppressing Action Points

You can tell TotalView to ignore action points by using the **Action Point > Suppress All** command.

CLI: ddisable –a

When you suppress action points, you disable them.After you suppress an action point, TotalView changes the symbol it uses within the Source Panes line number area. In all cases, the icon's color will be lighter. If you have suppressed action points, you cannot update existing action points or create new ones.

You can make previously suppressed action points active and allow the creation of new ones by reselecting the **Action Point > Suppress All** command.

CLI: denable –a

Setting Machine-Level Breakpoints

GUI

To set a machine-level breakpoint, you must first display assembler code. (For information, see "*Viewing the Assembler Version of Your Code*" on page 131.) You can now select an instruction. TotalView replaces some line number with a dotted box (⁝⁝⁝)—this indicates the line is the beginning of a machine instruction. If a line has a line number, this is the line number that appears in the Source Pane. Since instruction sets on some platforms support variable-length instructions, you might see a different number of lines associated with a single line contained in the dotted box. The STOP icon appears, indicating that the breakpoint occurs before the instruction executes.

If you set a breakpoint on the first instruction after a source statement, however, TotalView assumes that you are creating a source-level breakpoint, not an assembler-level breakpoint.

Figure 199: Breakpoint at Assembler Instruction

```
116     0x0804b213:                0xeb  jmp      0x804b215
        0x0804b214:                0x00
117     0x0804b215:                0x89  movl     %ebp,%esp
        0x0804b216:                0xec
 ⁝⁝⁝    0x0804b217:                0x5d  popl     %ebp
 ⁝⁝⁝    0x0804b218:                0xc3  ret
 ⁝⁝⁝    0x0804b219:                0x8d  leal     0(%esi),%esi
        0x0804b21a:                0x76
        0x0804b21b:                0x00
 ⇨  MB_fork_notify_breakpoint_here: 0x55  pushl    %ebp
 ⁝⁝⁝    0x0804b21d:                0x89  movl     %esp,%ebp
        0x0804b21e:                0xe5
124     0x0804b21f:                0xeb  jmp      0x804b221
        0x0804b220:                0x00
125     0x0804b221:                0x89  movl     %ebp,%esp
        0x0804b222:                0xec
 ⁝⁝⁝    0x0804b223:                0x5d  popl     %ebp
 ⁝⁝⁝    0x0804b224:                0xc3  ret
 ⁝⁝⁝    0x0804b225:                0x8d  leal     0(%esi),%esi
        0x0804b226:                0x76
```

If you set machine-level breakpoints on one or more instructions generated from a single source line, and then display source code in the Source Pane, TotalView displays an ASM icon (see Figure 194 on page 294) on the line number. To see the actual breakpoint, you must redisplay assembler instructions.

When a process reaches a breakpoint, TotalView does the following:

- Suspends the process.
- Displays the PC arrow icon ( ) over the stop sign to indicate that the PC is at the breakpoint.

Figure 200: PC Arrow Over a Stop Icon

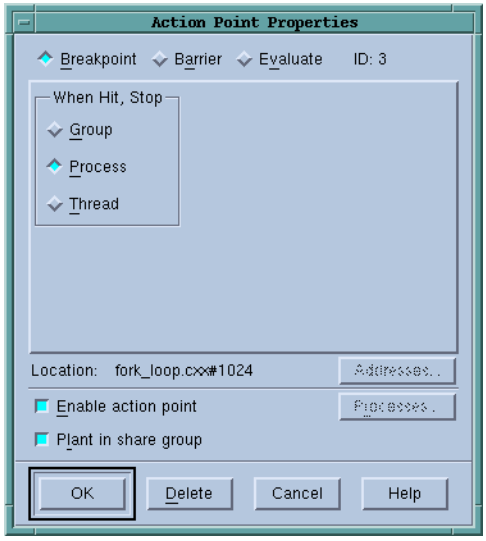
```
 80          do 40 i = 1, 500
 81             denorms(i) = x'00000001'
 82     40   continue
STOP          do 42 i = 500, 1000
 84             denorms(i) = x'80000001'
 86     42   continue
              local_var=100
 87           ieee_array(1) = x'7f800000'              ! infinity
 88           ieee_array(2) = x'ff800000'              ! -infinity
 89           ieee_array(3) = x'7fc00001'              ! NANQ
 90           ieee_array(4) = x'7f800001'              ! NANS
 91           ieee_array(5) = x'00000001'              ! positive denormalized number
 92           ieee_array(6) = x'80000001'              ! negative denormalized number
```

- Displays **At Breakpoint** in the Process Window title bar and other windows.
- Updates the Stack Trace and Stack Frame Panes and all Variable Windows.

Setting Breakpoints for Multiple Processes

In all programs, including multiprocess programs, you can set breakpoints in parent and child processes before you start the program and while the program is executing. Do this using the **Action Point > Properties** Dialog Box.

Figure 201: Action Point > Properties Dialog Box



This dialog box provides the following controls for setting breakpoints:

- **When Hit, Stop**

  When your thread hits a breakpoint, TotalView can also stop the thread's control group or the process in which it is running.

CLI:	dset STOP_ALL
	dbreak –p | –g | –t

- **Plant in share group**

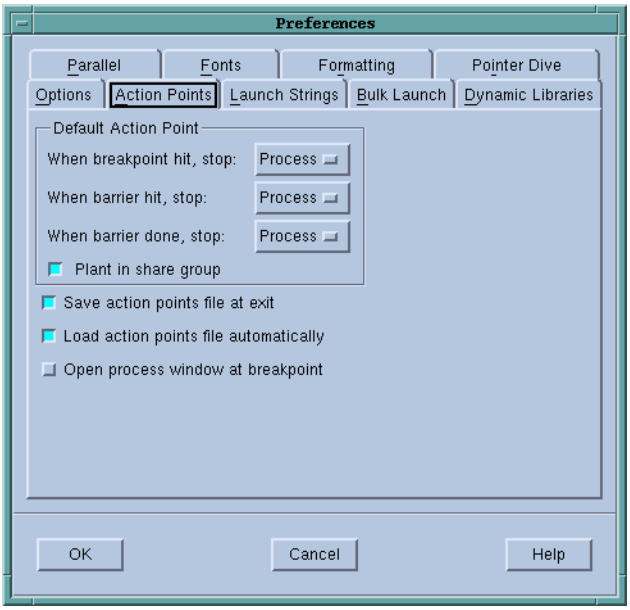
  If you select this check box, TotalView enables the breakpoint in all members of this thread's share group at the same time. If not, you must individually enable and disable breakpoints in each member of the share group.

  CLI: dset SHARE_ACTION_POINT

The **Processes** button lets you indicate which process in a multiprocess program will have enabled breakpoints. If **Plant in share group** is selected, TotalView does not enable this button because you told TotalView to set the breakpoint in all of the processes.

You can preset many of the properties in this dialog box by selecting the **File > Preferences** command. Use the Action Points Page to set action point preferences.

Figure 202: File > Preferences: Action Points Page



You can find additional information about this dialog box in the online Help.

If you select the **Evaluate** button in the **Action Point > Properties** Dialog Box, you can add an expression to the action point. This expression is attached to control and share group members. See "*Writing Code Fragments*" on page 322 for more information.

If you're trying to synchronize your program's threads, you need to set a barrier point. For more information, see "*Setting Barrier Points*" on page 303.

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

You must link with the dbfork library before debugging programs that call the **fork()** and **execve()** functions. See "*Compiling Programs*" on page 35.

Debugging Processes That Call the fork() Function

By default, TotalView places breakpoints in all processes in a share group. (For information on share groups, see "*Organizing Chaos*" on page 22.) When any process in the share group reaches a breakpoint, TotalView stops all processes in the control group. This means that TotalView stops the control group that contains the share group. This control can contain more than one share group.

To override these defaults:

1. Dive into the line number to display the **Action Point** > **Properties** Dialog Box.
2. Clear the **Plant in share group** check box and make sure that the **Group** radio button is selected.

CLI:	dset SHARE_ACTION_POINT false

Debugging Processes that Call the execve() Function

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

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

1. Set the breakpoints and breakpoint options in the parent and the children that do not call the **execve()** function.
2. Start the multiprocess program by displaying the **Group** > **Go** command.

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

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

CLI:	G

3. Answer **Yes**.

   TotalView opens a Process Window for the process. (If you answer **No**, you won't have an opportunity to set breakpoints.)
4. Set breakpoints for the process.

   After you set breakpoints for the first child using this executable, TotalView won't prompt when other children call the **execve()** function. This means that if you do not want to share breakpoints in children that use the same executable, dive into the breakpoints and set the breakpoint options.
5. Select the **Group** > **Go** command.

Example: Multiprocess Breakpoint

The following program excerpt illustrates the places where you can set breakpoints in a multiprocess program:

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

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

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

Setting Barrier Points

A barrier breakpoint is similar to a simple breakpoint, differing only in that it holds processes and threads that reach the barrier point. Other processes and threads continue to run. TotalView holds these processes or threads until all processes or threads defined in the barrier point reach this same place. When the last one reaches a barrier point, TotalView releases all the held processes or threads. In this way, barrier points let you synchronize your program's execution.

CLI: dbarrier

Topics in this section are:

- *"About Barrier Breakpoint States"* on page 303
- *"Setting a Barrier Breakpoint"* on page 304
- *"Creating a Satisfaction Set"* on page 305
- *"Hitting a Barrier Point"* on page 305
- *"Releasing Processes from Barrier Points"* on page 306
- *"Deleting a Barrier Point"* on page 306
- *"Changing Settings and Disabling a Barrier Point"* on page 306

About Barrier Breakpoint States

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

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

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

You can manually release held processes and threads with the **Hold** and **Release** commands found in the **Group**, **Process**, and **Thread** menus. When you manually release a process, the *go* and *step* commands become available again.

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

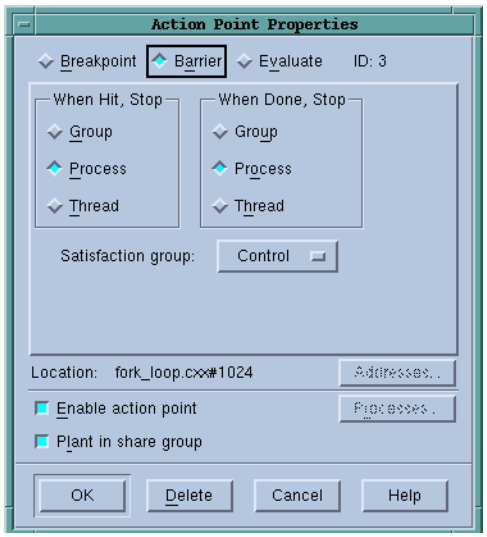
You can reuse the **Hold** command to again toggle the hold state of the process or thread. See "*Holding and Releasing Processes and Threads*" on page 185 for more information.

When a process or a thread is held, TotalView displays an **H** (for a held process) or an **h** (for a held thread) in the process's or thread's entry in the Root Window.

Setting a Barrier Breakpoint

You can set a barrier breakpoint by using the **Action Point > Set Barrier** command or from the **Action Point > Properties** Dialog Box. As an alternative, you can right-click on the line. From the displayed context menu, you can select the **Set Barrier** command.

Figure 203: Action Point > Properties Dialog Box



You most often use barrier points to synchronize a set of threads. When a thread reaches a barrier, it stops, just as it does for a breakpoint. The difference is that TotalView prevents—that is, holds—each thread reaching the barrier from responding to resume commands (for example, *step*, *next*, or *go*) until all threads in the affected set arrive at the barrier. When all threads

reach the barrier, TotalView considers the barrier to be *satisfied* and releases all of the threads being held there. T*hey are just released*; *they are not continued*. That is, they are left stopped at the barrier. If you continue the process, those threads also run.

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

The **When Hit, Stop** radio buttons indicate what other threads TotalView stops when execution reaches the breakpoint, as follows:

Scope	What TotalView does:
Group	Stops all threads in the current thread's control group.
Process	Stops all threads in the current thread's process.
Thread	Stops only this thread.

CLI: dbarrier –stop_when_hit

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

The **When Done, Stop** radio buttons tell TotalView what else it should stop, as follows:

Scope	What TotalView does:
Group	Stops all threads in the current thread's control group.
Process	Stops all threads in the current thread's process.
Thread	Stops only this thread.

CLI: dbarrier –stop_when_done

Creating a Satisfaction Set

For even more control over what TotalView stops, you can select a *satisfaction set*. This set tells TotalView which threads must be held before it can release the group of threads. That is, the barrier is *satisfied* when TotalView has held all of the indicated threads. Use the **Satisfaction group** items to tell TotalView that the satisfaction set consists of all threads in the current thread's **Control**, **Workers**, or **Lockstep** group.

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

Hitting a Barrier Point

If you run one of the processes or threads in a group and it hits a barrier point, you sees an **H** next to the process or thread name in the Root Window and the word **[Held]** in the title bar in the main Process Window. Barrier points are always shared.

CLI: dstatus

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

Releasing Processes from Barrier Points

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

Deleting a Barrier Point

You can delete a barrier point in the following ways:

- Use the **Action Point** > **Properties** Dialog Box.
- Click the BARR icon in the line number area.

CLI: ddelete

Changing Settings and Disabling a Barrier Point

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

TotalView releases all processes and threads that are held and which have threads at the barrier point when you disable the barrier point. You can disable the barrier point in the **Action Point** > **Properties** Dialog Box by selecting **Enable action point** at the bottom of the dialog box.

CLI: ddisable

Defining Eval Points and Conditional Breakpoints

TotalView lets you define *eval points*. These are action points at which you have added a code fragment that TotalView executes. You can write the code fragment in C, Fortran, or assembler.

Assembler support is currently available on the HP Alpha Tru64 UNIX, IBM AIX, *and* SGI IRIX *operating systems. You can enable or disable the debugger's ability to compile eval points.*

When running on many AIX *systems, you can speed up the performance of compiled expressions by using the* –use_aix_fast_trap *command when you start TotalView. For more information, see the TotalView Release Notes.*

Topics in this section are:

- "*Setting Eval Points*" on page 308
- "*Creating Conditional Breakpoint Examples*" on page 308
- "*Patching Programs*" on page 309
- "*About Interpreted Compiled Expressions*" on page 310
- "*Allocating Patch Space for Compiled Expressions*" on page 311

You can do the following when you use eval points:

- Include instructions that stop a process and its relatives. If the code fragment can make a decision whether to stop execution, it is called a *conditional breakpoint*.
- Test potential fixes for your program.
- Set the values of your program's variables.
- Automatically send data to the Visualizer. This can produce animated displays of the changes in your program's data.

You can set an eval point at any source line that generates executable code (marked with a box surrounding a line number) or a line that contains assembler-level instructions. This means that if you can set a breakpoint, you can set an eval point.

At each eval point, TotalView or your program executes the code contained in the eval point before your program executes the code on that line. Although your program can then go on to execute this source line or instruction, it can do the following instead:

- Include a branching instruction (such as **goto** in C or Fortran). The instruction can transfer control to a different point in the target program, which lets you test program patches.
- Execute a TotalView function. These functions let you stop execution, create barriers, and countdown breakpoints. For more information on these statements, see "*Using Built-In Statements*" on page 323.

TotalView evaluates code fragments in the context of the target program. This means that you can refer to program variables and branch to places in your program.

If you call a function from in an eval point and there's a breakpoint within that function, TotalView will stop execution at that point. Similarly, if there's an eval point in the function, TotalView also evaluates that eval point.

For information on what you can include in code fragments, refer to "*Writing Code Fragments*" on page 322.

Eval points only modify the processes being debugged—they do not modify your source program or create a permanent patch in the executable. If you save a program's action points, however, TotalView reapplies the eval point whenever you start a debugging session for that program. For information about how to save your eval points, see"*Saving Action Points to a File*" on page 319.

You should stop a process before setting an eval point in it. This ensures that the eval point is set in a stable context.

Setting Eval Points

This section contains the steps you must follow to create an eval point. These steps are as follows:

1. Display the **Action Point > Properties** Dialog Box. You can do this, for example, by right-clicking a STOP icon and selecting **Properties** or by selecting a line and then invoking the command from the menu bar.
2. Select the **Evaluate** button at the top of the dialog box.
3. Select the button (if it isn't already selected) for the language in which you plan to write the fragment.
4. Type the code fragment. For information on supported C, Fortran, and assembler language constructs, see "*Writing Code Fragments*" on page 322.
5. For multiprocess programs, decide whether to share the eval point among all processes in the program's share group. By default, TotalView selects the **Plant in share group** check box for multiprocess programs, but you can override this by clearing this setting.
6. Select the **OK** button to confirm your changes.

   If the code fragment has an error, TotalView displays an error message. Otherwise, it processes the code, closes the dialog box, and places an EVAL icon on the line number in the Source Pane.

CLI: dbreak –e
dbarrier –e

The variables that you refer to in your eval point can either have a global scope or be local to the block of the line that contains the eval point. If you declare a variable in the eval point, its scope is the block that contains the eval point unless, for example, you declare it in some other scope or declare it to be a static variable.

Creating Conditional Breakpoint Examples

The following are examples that show how you can create conditional breakpoints:

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

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

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

  ```
  $count 10
  ```

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

  ```
  $count my_var * 2
  ```

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

For descriptions of the **$stop** and **$count** statements, see "*Using Built-In Statements*" on page 323.

Patching Programs

You can use expressions in eval points to patch your code if you use the **goto** (C) and **GOTO** (Fortran) statements to jump to a different program location. This lets you do the following:

- Branch around code that you don't want your program to execute.
- Add new statements.

In many cases, correcting an error means that you will do both operations: you use a goto to branch around incorrect lines and add corrections.

Branching Around Code

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

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

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

You can correct this error by setting an eval point on line 3 and entering:

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

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

Adding a Function Call

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

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

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

Correcting Code

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

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

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

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

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

About Interpreted Compiled Expressions

TotalView can either interpret your eval points. It can instead compile them on the HP Alpha Tru64 UNIX, IBM AIX, and SGI IRIX platforms. On HP Alpha Tru64 UNIX and IBM AIX platforms, compiling the expressions in eval points is the default.

You can use the **TV::compile_expressions** CLI variable to enable or disable compiled expressions. See "*Operating Systems*" in the *TotalView Reference Guide* for information about how TotalView handles expressions on specific platforms.

Using any of the following functions forces TotalView to interpret the eval point rather than compile it: **$clid**, **$duid**, **$nid**, **$processduid**, **$systid**, **$tid**, *and* **$visualize**. *In addition*, **$pid** *forces interpretation on* IBM AIX.

About Interpreted Expressions

Interpreted expressions are interpreted by TotalView. Interpreted expressions run slower, possibly much slower, than compiled expressions. With multiprocess programs, interpreted expressions run even more slowly because processes may need to wait for TotalView to execute the expression.

When you debug remote programs, interpreted expressions always run slower because the TotalView process on the host, not the TotalView debugger server (**tvdsvr**) on the client, interprets the expression. For example, an interpreted expression could require that 100 remote processes wait for the TotalView debugger process on the host machine to evaluate one interpreted expression. In contrast, if TotalView compiles the expression, it evaluates them on each remote process.

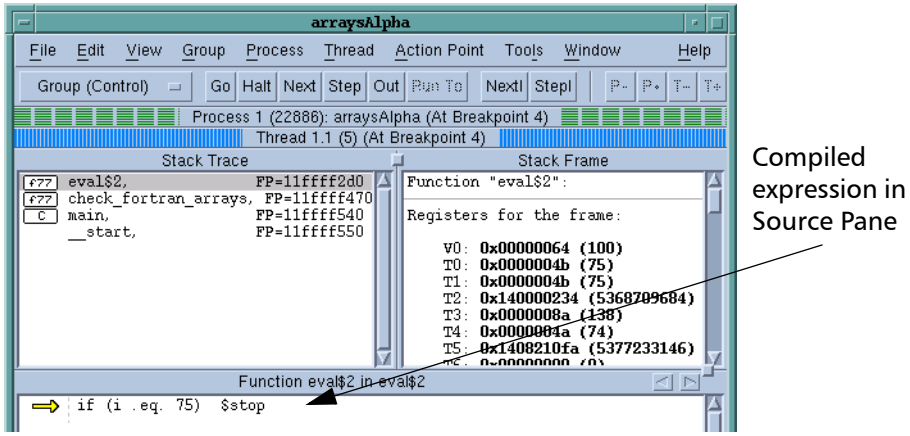
Whenever a thread hits an interpreted eval point, TotalView stops execution. This means that TotalView creates a new set of lockstep groups. Consequently, if goal threads contain interpreted patches, the results are unpredictable.

About Compiled Expressions

TotalView compiles, links, and patches expressions into the target process. Because the target thread executes this code, eval points and conditional breakpoints execute very quickly. (Conditional watchpoints are always interpreted.) Also, this code doesn't need to communicate with the TotalView host process until it needs to.

If the expression executes a **$stop** function, TotalView stops executing the compiled expression. At this time, you can single-step through it and continue executing the expression as you would the rest of your code.

Figure 204: Stopped Execution of Compiled Expressions



If you plan to use many compiled expressions or your expressions are long, you may need to think about allocating patch space.

Allocating Patch Space for Compiled Expressions

TotalView must either allocate or find space in your program to hold the code that it generates for compiled expressions. Since this patch space is part of your program's address space, the location, size, and allocation scheme that TotalView uses might conflict with your program. As a result, you may need to change how TotalView allocates this space.

You can choose one of the following patch space allocation schemes:

- **Dynamic patch space allocation**: Tells TotalView to dynamically find the space for your expression's code.
- **Static patch space allocation**: Tells TotalView to use a statically allocated area of memory.

Allocating Dynamic Patch Space

Dynamic patch space allocation means that TotalView dynamically allocates patch space for code fragments. If you do not specify the size and location for this space, TotalView allocates 1 MB. TotalView creates this space using system calls.

TotalView allocates memory for read, write, and execute access in the addresses shown in the following table:

Platform	Address Range
HP Alpha Tru64 UNIX	`0xFFFFF00000 - 0xFFFFFFFFFF`
IBM AIX (-q32)	`0xEFF00000 - 0xEFFFFFFF`
IBM AIX (-q64)	`0x07f0000000000000 - 0x07ffffffffffffff`
SGI IRIX (–n32)	`0x4FF00000 - 0x4FFFFFFF`
SGI IRIX (–64)	`0x8FF00000 - 0x8FFFFFFF`

You can only allocate dynamic patch space for the computers listed in this table.

If the default address range conflicts with your program, or you would like to change the size of the dynamically allocated patch space, you can change the following:

- *Patch space base address* by using the **–patch_area_base** command-line option.
- *Patch space length* by using the **–patch_area_length** command-line option.

Allocating Static Patch Space

TotalView can statically allocate patch space if you add a specially named array to your program. When TotalView needs to use patch space, it uses the space created for this array.

You can include, for example, a 1 MB statically allocated patch space in your program by adding the **TVDB_patch_base_address** data object in a C module. Because this object must be 8-byte aligned, declare it as an array of doubles; for example:

```
            /* 1 megabyte == size TV expects */
#define PATCH_LEN 0x100000
double TVDB_patch_base_address [PATCH_LEN /
sizeof(double)]
```

If you need to use a static patch space size that differs from the 1 MB default, you must use assembler language. The following table shows sample assembler code for three platforms that support compiled patch points:

Platform	Assembler Code

HP Alpha Tru64 UNIX	`.data`
	`.align 3`
	`.globl TVDB_patch_base_address`
	`.globl TVDB_patch_end_address`
	`TVDB_patch_base_address:`
	`.byte 0x00 : PATCH_SIZE`
	`TVDB_patch_end_address:`

Platform	Assembler Code

IBM AIX	`.csect .data{RW}, 3`
	`.globl TVDB_patch_base_address`
	`.globl TVDB_patch_end_address`
	`TVDB_patch_base_address:`
	`.space PATCH_SIZE`
	`TVDB_patch_end_address:`

SGI IRIX	`.data`
	`.align 3`
	`.globl TVDB_patch_base_address`
	`.globl TVDB_patch_end_address`
	`TVDB_patch_base_address:`
	`.space PATCH_SIZE`
	`TVDB_patch_end_address:`

To use the static patch space assembler code:

1. Use an ASCII editor and place the assembler code into a file named **tvdb_patch_space.s**.
2. Replace the **PATCH_SIZE** tag with the decimal number of bytes you want. This value must be a multiple of 8.
3. Assemble the file into an object file by using a command such as:

   ```
   cc -c tvdb_patch_space.s
   ```

   On SGI IRIX, use **–n32** or **–64** to create the correct object file type.
4. Link the resulting **tvdb_patch_space.o** into your program.

Using Watchpoints

TotalView lets you monitor the changes that occur to memory locations by creating a special type of action point called a *watchpoint*. You most often use watchpoints to find a statement in your program that is writing to places to which it shouldn't be writing. This can occur, for example, when processes share memory and more than one process writes to the same location. It can also occur when your program writes off the end of an array or when your program has a dangling pointer.

Topics in this section are:

- "*Using Watchpoints on Different Architectures*" on page 314
- "*Creating Watchpoints*" on page 315
- "*Watching Memory*" on page 316
- "*Triggering Watchpoints*" on page 316
- "*Using Conditional Watchpoints*" on page 317

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

For example, if location 0x10000 has a value of 0 and your program writes a value of 0 to this location, TotalView doesn't trigger the watchpoint, even

though your program wrote data to the memory location. See "*Triggering Watchpoints*" on page 316 for more details on when watchpoints trigger.

You can also create *conditional watchpoints*. A conditional watchpoint is similar to a conditional breakpoint in that TotalView evaluates the expression when the value in the watched memory location changes. You can use conditional watchpoints for a number of purposes. For example, you can use one to test whether a value changes its sign—that is, it becomes positive or negative—or whether a value moves above or below some threshold value.

Using Watchpoints on Different Architectures

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

Watchpoints are not available on Hewlett Packard (HP) *computers running either Alpha Linux or* HP-UX.

The following list describes constraints that exist on each platform:

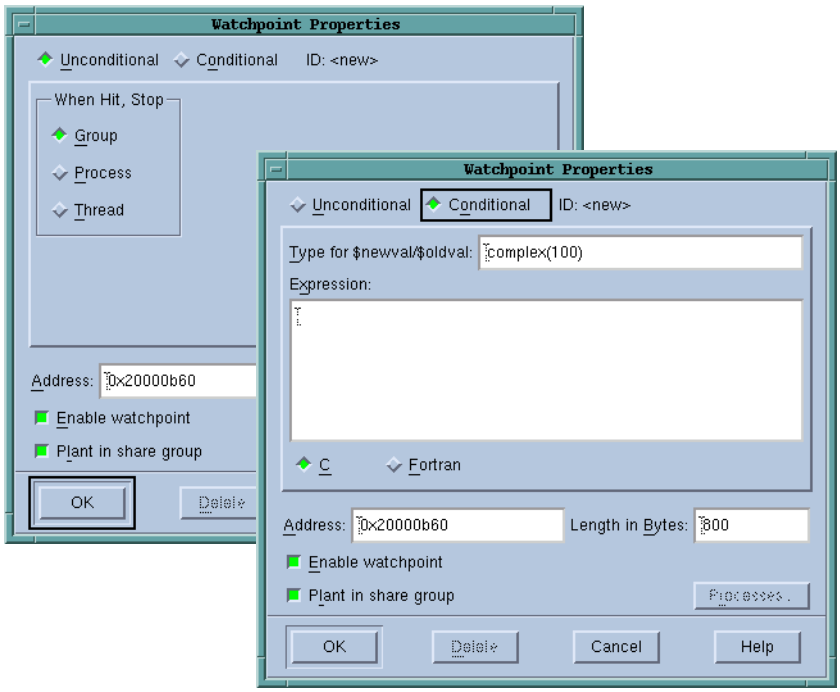
Computer	Constraints
HP Alpha Tru64	Tru64 places no limitations on the number of watchpoints that you can create, and no alignment or size constraints. However, watchpoints can't overlap, and you can't create a watchpoint on an already write-protected page. Watchpoints use a page-protection scheme. Because the page size is 8,192 bytes, watchpoints can degrade performance if your program frequently writes to pages that contains watchpoints
IBM AIX	You can create one watchpoint on AIX 4.3.3.0-2 (AIX 4.3R) or later systems running 64-bit chips. These are Power3 and Power4 systems. (AIX 4.3R is available as APAR IY06844.) A watchpoint cannot be longer than 8 bytes, and you must align it within an 8-byte boundary.
IRIX6 MIPS	Watchpoints are implemented on IRIX 6.2 and later operating systems. These systems let you create approximately 100 watchpoints. There are no alignment or size constraints. However, watchpoints can't overlap.
Linux x86, Linux IA-64, AMD Opteron, HP-UX IA-64	You can create up to four watchpoints and each must be 1, 2, or 4 bytes in length, and a memory address must be aligned for the byte length. That is, you must align a 4-byte watchpoint on a 4-byte address boundary, and you must align 2-byte watchpoint on a 2-byte boundary, and so on.
Solaris SPARC	TotalView supports watchpoints on Solaris 2.6 or later operating systems. These operating system let you create hundreds of watchpoints, and there are no alignment or size constraints. However, watchpoints can't overlap.

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

Creating Watchpoints

Watchpoints are created by using the **Tools > Watchpoint** Dialog Box, which you can only invoke from a Variable Window. (If your platform doesn't support watchpoints, TotalView dims this menu item.)

Figure 205: Tools > Watchpoint Dialog Boxes



Controls in this dialog box let you create unconditional and conditional watchpoints. When you set a watchpoint, you are setting it on the complete contents of the information being displayed in the Variable Window. For example, if the Variable Window displays an array, you can only set a watchpoint on the entire array (or as many bytes that TotalView can watch.) If you only want to watch one array element, dive on the element and then set the watchpoint. Similarly, if the Variable Window displays a structure and you only want to watch one element, dive on the element before you set the watchpoint.

See the online Help for information on the fields in this dialog box.

Displaying Watchpoints

The watchpoint entry, indicated by UDWP (Unconditional Data Watchpoint) and CDWP (Conditional Data Watchpoint), displays the action point ID, the amount of memory being watched, and the location being watched.

If you dive into a watchpoint, TotalView displays the **Watchpoint Properties** Dialog Box.

If you select a watchpoint, TotalView toggles the enabled/disabled state of the watchpoint.

Watching Memory

A watchpoint tracks a memory location—it does not track a variable. This means that a watchpoint might not perform as you would expect it to when watching stack or automatic variables. For example, suppose that you want to watch a variable in a subroutine. When control exits from the subroutine, the memory allocated on the stack for this subroutine is deallocated. At this time, TotalView is watching unallocated stack memory. When the stack memory is reallocated to a new stack frame, TotalView is still watching this same position. This means that TotalView triggers the watchpoint when something changes this newly allocated memory.

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

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

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

In some circumstances, a subroutine may be called from the same location. This means that its local variables might be in the same location. So, you might want to try.

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

Triggering Watchpoints

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

Using Multiple Watchpoints

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

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

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

Copying Previous Data Values

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

Using Conditional Watchpoints

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

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

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

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

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

The following is an expression that uses these values:

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

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

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

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

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

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

Both of these examples require that you set the **Type for $oldval/$newval** field in the **Watchpoint Properties** Dialog Box.

For more information on writing expressions, see "*Writing Code Fragments*" on page 322.

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

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

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

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

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

TotalView always interprets conditional watchpoints; it never compiles them.Because interpreted watchpoints are single-threaded in TotalView, every process or thread that writes to the watched location must wait for other instances of the watchpoint to finish executing. This can adversely affect performance.

Saving Action Points to a File

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

TotalView does not save watchpoints because memory addresses can change radically every time you restart TotalView and your program.

Use the **Action Point** > **Save All** command to save your action points to a file. TotalView places the action points file in the same directory as your program. In contrast, the **Action Point** > **Save As** command lets you name the file to which TotalView saves this information.

CLI: **dactions –save** *filename*

If you're using a preference to automatically save breakpoints, TotalView automatically saves action points to a file. Alternatively, starting TotalView with the **–sb** option (see "*TotalView Command Syntax*" in the *TotalView Reference Guide*) also tells TotalView to save your breakpoints.

At any time, you can restore saved action points if you use the **Action Points** > **Load All** command. After invoking this command, TotalView displays a File Explorer window that you can use to navigate to or name the saved file.

CLI: **dactions –load** *filename*

You control automatic saving and loading by setting preferences. (See **File** > **Preferences** in the online Help for more information.)

CLI: **dset TV::auto_save_breakpoints**

Evaluating Expressions

TotalView lets you open a window to evaluate expressions in the context of a particular process and evaluate them in C, Fortran, or assembler.

Not all platforms let you use assembler constructs. See "Architectures" in the TotalView Reference Guide for details.

You can use the **Tools** > **Evaluate** Dialog Box in many different ways. The following are two examples:

- Expressions can contain loops, so you can use a **for** loop to search an array of structures for an element 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 field.

- Because you can call subroutines, you can test and debug a single routine in your program without building a test program to call it.

To evaluate an expression:

1. Display the **Evaluate** Dialog Box by selecting the **Tools** > **Evaluate** command.

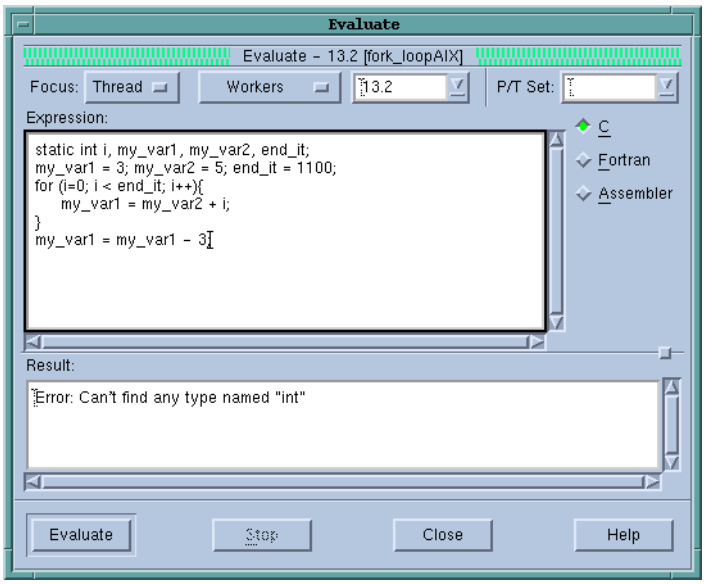
   An **Evaluate** Dialog Box appears. If your program hasn't yet been created, you won't be able to use any of the program's variables or call any of its functions.

2. Select a button for the programming language you're writing the expression in (if it isn't already selected).

3. Move to the **Expression** field and enter a code fragment. For a description of the supported language constructs, see "*Writing Code Fragments*" on page 322.

   The following figure shows a sample expression. The last statement in this example assigns the value of **my_var1-3** back to **my_var1**. Because this is the last statement in the code fragment, the value placed in the **Result** field is the same as if you had just typed **my_var1-3**.

Figure 206: Tools > Evaluate Dialog Box



4. Click the **Evaluate** button.

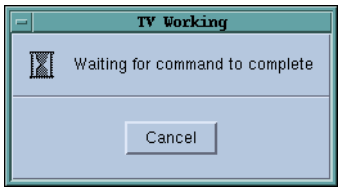
   If TotalView finds an error, it places the cursor on the incorrect line and displays an error message. Otherwise, it interprets (or on some platforms, compiles and executes) the code, and displays the value of the last expression in the **Result** field.

   While the code is being executed, you can't modify anything in the dialog box. TotalView might also display a message box that tells you that it is waiting for the command to complete.

   If you click **Cancel**, TotalView stops execution.

Since TotalView evaluates code fragments in the context of the target process, it evaluates stack variables according to the current program counter.

Figure 207: Waiting to Complete Message Box

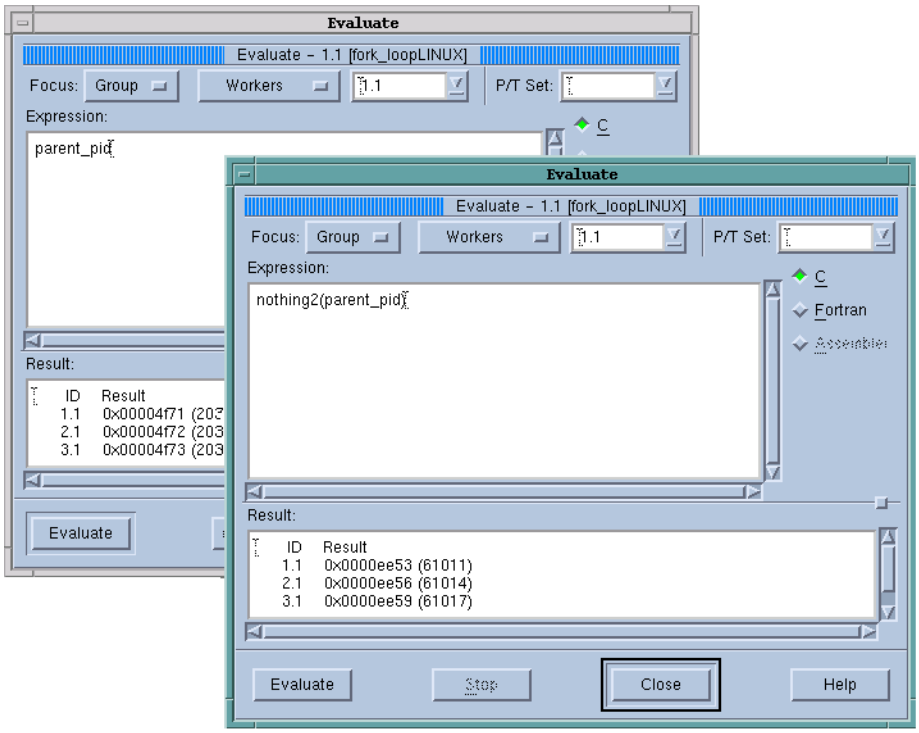


If you declare a variable, its scope is the block that contains the program counter unless, for example, you declare it in some other scope or declare it to be a static variable.

If the fragment reaches a breakpoint (or stops for any other reason), TotalView stops evaluating your expression. Assignment statements in an expression can affect the target process because they can change a variable's value.

The controls at the top of the dialog box let you refine the scope at which TotalView evaluates the information you enter. For example, you can evaluate a function in more than one process. The following figure shows TotalView displaying the value of a variable in multiple processes, and then sending the value as it exists in each process to a function that runs on each of these processes.

Figure 208: Evaluating Information in Multiple Processes



See Chapter 11, "*Using Groups, Processes, and Threads*," on page 205 for information on using the P/T set controls at the top of this window.

Writing Code Fragments

You can use code fragments in eval points and in the **Tools** > **Evaluate** Dialog Box. This section describes the function variables, built-in statements, and language constructs that TotalView supports.

> CLI: Although the CLI does not have an evaluate command, the information in the following sections does apply to the expression argument of the dbreak, dbarrier, and dwatch commands.

Topics in this section are:

- "*Using TotalView Variables*" on page 322
- "*Using Built-In Statements*" on page 323
- "*About Supported C Constructs*" on page 324
- "*Supported Fortran Constructs*" on page 325
- "*Writing Assembler Code*" on page 326

Using TotalView Variables

The TotalView expression system supports built-in variables that let you access special thread and process values. All variables are 32-bit integers, which is an **int** or a **long** on most platforms. The following table describes built-in variables:

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

The built-in variables let you create thread-specific breakpoints from the expression system. For example, the **$tid** variable and the **$stop** built-in function let you create a thread-specific breakpoint, as the following code shows:

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

This tells TotalView to stop the process only when the third thread evaluates the expression.

You can also create complex expressions by using these variables; for example:

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

Using any of the following variables means that the eval point is interpreted instead of compiled: **$clid**, **$duid**, **$nid**, **$processduid**, **$systid**, **$tid**, and **$visualize**. In addition, **$pid** forces interpretation on AIX.

You can't assign a value to a built-in variable or obtain its address.

Using Built-In Statements

TotalView provides a set of built-in statements that you can use when writing code fragments. The statements are available in all languages, and are described in the following table:

Statement	Use
$count *expression* **$countprocess** *expression*	Sets a process-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by *expression*, the process stops. The other processes in the control group continue to execute.
$countall *expression*	Sets a program-group-level countdown breakpoint. All processes in the control group stop when any process in the group executes this statement for the number of times specified by *expression*.
$countthread *expression*	Sets a thread-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by *expression*, the thread stops. Other threads in the process continue to execute. If the target system cannot stop an individual thread, this statement performs the same as **$countprocess**. A thread evaluates *expression* when it executes **$count** for the first time. This expression must evaluate to a positive integer. When TotalView first encounters this variable, it determines a value for *expression*. TotalView does not reevaluate until the expression actually stops the thread. This means that TotalView ignores changes in the value of *expression* until it hits the breakpoint. After the breakpoint occurs, TotalView reevaluates the expression and sets a new value for this statement. The internal counter is stored in the process and shared by all threads in that process.
$hold **$holdprocess**	Holds the current process. If all other processes in the group are already held at this eval point, TotalView releases all of them. If other processes in the group are running, they continue to run.
$holdstopall **$holdprocessstopall**	Like **$hold**, except that any processes in the group which are running are *stopped*. The other processes in the group are not automatically held by this call—they are just stopped.

Statement	Use
$holdthread	Freezes the current thread, leaving other threads running.
$holdthreadstop $holdthreadstopprocess	Like **$holdthread,** except that it *stops* the process. The other processes in the group are left running.
$holdthreadstopall	Like **$holdthreadstop,** except that it stops the entire group.
$stop $stopprocess	Sets a process-level breakpoint. The process that executes this statement stops; other processes in the control group continue to execute.
$stopall	Sets a program-group-level breakpoint. All processes in the control group stop when any thread or process in the group executes this statement.
$stopthread	Sets a thread-level breakpoint. Although the thread that executes this statement stops, all other threads in the process continue to execute. If the target system cannot stop an individual thread, this statement performs the same as to **$stopprocess**.
$visualize(*expression*[,*slice*])	Visualizes the data specified by *expression* and modified by the optional *slice* value. E*xpression* and *slice* must be expressed using the code fragment's language. The *expression* must return a dataset (after modification by *slice*) that can be visualized. *slice* is a quoted string that contains a slice expression. For more information on using **$visualize** in an expression, see "*Visualizing Data Programmatically*" on page 143.

About Supported C Constructs

When writing code fragments in C, you should follow the following guidelines:

- You can use C-style (**/* comment */**) and C++-style (**// comment**) comments; for example:

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

- You can omit semicolons if the result isn't ambiguous.
- You can use dollar signs ($) in identifiers.

Data Types and Declarations

You can use the following C data types and declarations:

- You can use **char**, **short**, **int**, **float**, and **double** data types, and pointers to any primitive type or any named type in the target program.
- You can only use simple declarations. Do not use **struct**, **union**, and array declarations.
- You can refer to variables of any type in the target program.
- Unmodified variable declarations are considered local. References to these declarations override references to similarly named global variables and other variables in the target program.
- (Compiled eval points only.) The **global** declaration makes a variable available to other eval points and expression windows in the target process.

- (Compiled eval points only.) The **extern** declaration references a global variable that was or will be defined elsewhere. If the global variable is not yet defined, TotalView displays a warning.
- Static variables are local and persist even after TotalView evaluates an eval point.
- TotalView only evaluates expressions that initialize static and global variables the first time it evaluates a code fragment. In contrast, it initializes local variables each time it evaluates a code fragment.

Statements

You can use the following the C language statements.

- You can use assignment, **break**, **continue**, **if**/**else** structures, **for**, **goto**, and **while** statements.
- You can use the **goto** statement to define and branch to symbolic labels. These labels are local to the window. You can also refer to a line number in the program. This line number is the number displayed in the Source Pane. For example, the following **goto** statement branches to source line number 432 of the target program:

  ```
  goto 432;
  ```
- Although you can use function calls, you can't pass structures.
- You can use type casting.

You can use all operators, with the following limitations:

- TotalView doesn't support the **?:** conditional operator.
- You can use the **sizeof** operator, but you can't use it for data types.
- The (*type*) operator can't cast data to fixed-dimension arrays by using C cast syntax.

Supported Fortran Constructs

When writing code fragments in Fortran, you need to follow these guidelines:

- Enter only one statement on a line. You can't continue a statement onto more than one line.
- You can use **GOTO**, **GO TO**, **ENDIF**, and **END IF** statements; Although **ELSEIF** statements aren't allowed, you can use **ELSE IF** statements.
- Syntax is free-form. No column rules apply.
- You can enter comments in three ways:
  - ➤ With a C in column 1
  - ➤ As free-form comments using the **/*** ... ***/** delimiters
  - ➤ Using the **//** characters. Anything typed after **//** characters is also ignored.

  The following example shows all three types of comments:

  ```
  C I=I+1
  /*
  I=I+1
  J=J+1
  ARRAY1(I,J)= I * J
  */
  k = 4 // This is also a comment
  ```

- The space character is significant and is sometimes required. (Some Fortran 77 compilers ignore all space characters.) For example:

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

Fortran Data Types and Declarations

You can use the following data types and declarations in a Fortran expression:

- You can use the **INTEGER** (assumed to be **long**), **REAL**, **DOUBLE PRECISION**, and **COMPLEX** data types.
- You can't use implied data types.
- You can only use simple declarations. You can't use a **COMMON**, **BLOCK DATA**, **EQUIVALENCE**, **STRUCTURE**, **RECORD**, **UNION**, or array declaration.
- Your can refer to variables of any type in the target program.

Fortran Statements

You can use the Fortran language statements:

- You can use assignment, **CALL** (to subroutines, functions, and all intrinsic functions except **CHARACTER** functions in the target program), **CONTINUE**, **DO**, **GOTO**, **IF** (including block **IF**, **ENDIF**, **ELSE**, and **ELSE IF**), and **RETURN** (but not alternate return) statements.
- A **GOTO** statement can refer to a line number in your program. This line number is the number that appears in the Source Pane. For example, the following **GOTO** statement branches to source line number 432:

  ```
  GOTO $432;
  ```

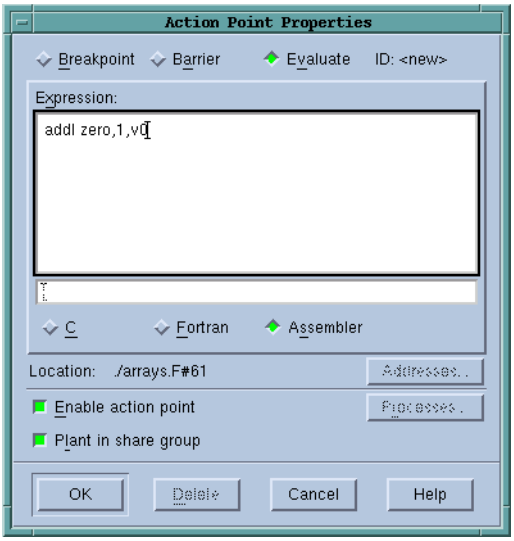
  You must use a dollar sign (**$**) before the line number so that TotalView knows that you're referring to a source line number rather than a statement label.
- The following expression operators are not supported: **CHARACTER** operators and the **.EQV.**, **.NEQV.**, and **.XOR.** logical operators.
- You can't use subroutine function and entry definitions.
- You can't use Fortran 90 array syntax.
- You can't use Fortran 90 pointer assignment (the => operator).
- You can't call Fortran 90 functions that require assumed shape array arguments.

Writing Assembler Code

On HP Alpha Tru64 UNIX, RS/6000 IBM AIX, and SGI IRIX operating systems, TotalView lets you use assembler code in eval points, conditional breakpoints, and in the **Tools > Evaluate** Dialog Box. However, if you want to use assembler constructs, you must enable compiled expressions. See *"About Interpreted Compiled Expressions"* on page 310 for instructions.

To indicate that an expression in the breakpoint or **Evaluate** Dialog Box is an assembler expression, click the **Assembler** button in the **Action Point > Properties** Dialog Box.

Figure 209: Using Assembler Expressions



You write assembler expressions in the target machine's native assembler language and in a TotalView assembler language. However, the operators available to construct expressions in instruction operands, and the set of available pseudo-operators, are the same on all machines, and are described below.

The TotalView assembler accepts instructions using the same mnemonics recognized by the native assembler, and it recognizes the same names for registers that native assemblers recognize.

Some architectures provide extended mnemonics that do not correspond exactly with machine instructions and which represent important, special cases of instructions, or provide for assembling short, commonly used sequences of instructions. The TotalView assembler recognizes mnemonics if:

- They assemble to exactly one instruction.
- The relationship between the operands of the extended mnemonics and the fields in the assembled instruction code is a simple one-to-one correspondence.

Assembler language labels are indicated as *name*: and appear at the beginning of a line. You can place a label alone on a line. The symbols you can use include labels defined in the assembler expression and all program symbols.

The TotalView assembler operators are described in the following table:

Operators	Description
+	Plus
–	Minus (also unary)
*	Multiplication
#	Remainder
/	Division
&	Bitwise AND
^	Bitwise XOR
!	Bitwise OR NOT (also unary minus, bitwise NOT)
|	Bitwise OR
(*expr*)	Grouping
<<	Left shift
>>	Right shift
"*text*"	Text string, 1-4 characters long, is right-justified in a 32-bit word
hi16 (*expr*)	Low 16 bits of operand *expr*
hi32 (*expr*)	High 32 bits of operand *expr*
lo16 (*expr*)	High 16 bits of operand *expr*
lo32 (*expr*)	Low 32 bits of operand *expr*

The TotalView assembler pseudo-operations are as follows:

Pseudo Ops	Description
$debug [0 | 1]	*Internal debugging option.* With no operand, toggle debugging; 0 => turn debugging off 1 => turn debugging on
$hold **$holdprocess**	Hold the process
$holdstopall **$holdprocessstopall**	Hold the process and stop the control group
$holdthread	Hold the thread
$holdthreadstop **$holdthreadstopprocess**	Hold the thread and stop the process
$holdthreadstopall	Hold the thread and stop the control group
$long_branch expr	Branch to location *expr* using a single instruction in an architecture-independent way; using registers is not required
$stop **$stopprocess**	Stop the process
$stopall	Stop the control group
$stopthread	Stop the thread
name=expr	Same as **def** *name,expr*
align *expr* [, *expr*]	Align location counter to an operand 1 alignment; use operand 2 (or 0) as the fill value for skipped bytes
ascii *string*	Same as *string*
asciz *string*	Zero-terminated string

Pseudo Ops	Description
bss *name,size-expr*[,*expr*]	Define *name* to represent *size-expr* bytes of storage in the bss section with alignment optional *expr*; the default alignment depends on the size: if *size-expr* >= 8 then 8 else if *size-expr* >= 4 then 4 else if *size-expr* >= 2 then 2 else 1
byte *expr* [, *expr*] ...	Place *expr* values into a series of bytes
comm *name,expr*	Define *name* to represent *expr* bytes of storage in the bss section; *name* is declared global; alignment is as in bss without an alignment argument
data	Assemble code into data section (data)
def *name,expr*	Define a symbol with *expr* as its value
double *expr* [, *expr*] ...	Place *expr* values into a series of doubles
equiv *name,name*	Make operand 1 an abbreviation for operand 2
fill *expr, expr, expr*	Fill storage with operand 1 objects of size operand 2, filled with value operand 3
float *expr* [, *expr*] ...	Place *expr* values into a series of floating point numbers
global *name*	Declare *name* as global
half *expr* [, *expr*] ...	Place *expr* values into a series of 16-bit words
lcomm *name,expr*[,*expr*]	Identical to bss
lsym *name,expr*	Same as def *name,expr* but allows redefinition of a previously defined name
org *expr* [, *expr*]	Set location counter to operand 1 and set operand 2 (or 0) to fill skipped bytes
quad *expr* [, *expr*] ...	Place *expr* values into a series of 64-bit words
string *string*	Place *string* into storage
text	Assemble code into text section (code)
word *expr* [, *expr*] ...	Place *expr* values into a series of 32-bit words
zero *expr*	Fill *expr* bytes with zeros

Glossary

ACTION POINT: A debugger feature that lets a user request that program execution stop under certain conditions. Action points include breakpoints, watchpoints, eval points, and barriers.

ACTION POINT IDENTIFIER: A unique integer ID associated with an action point.

ADDRESS SPACE: 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.

ADDRESSING EXPRESSION: A set of instructions that tell TotalView where to find information. These expressions are only used within the *type transformation facility* on page 341.

AFFECTED P/T SET: The set of process and threads that are affected by the command. For most commands, this is identical to the target P/T set, but in some cases it might include additional threads. (See "*p/t* (*process/thread*) *set*" on page 338 for more information.)

AGGREGATE DATA: A collection of data elements. For example, a structure or an array is an aggregate.

AGGREGATED OUTPUT: The CLI compresses output from multiple threads when they would be identical except for the P/T identifier.

ARENA: A specifier that indicates the processes, threads, and groups upon which a command executes. Arena specifiers are **p** (process), **t** (thread), **g** (group), **d** (default), and **a** (all).

ARRAY SLICE: A subsection of an array, which is expressed in terms of a *lower bound* on page 336, *upper bound* on page 342, and *stride* on page 340. Displaying a slice of an array can be useful when you are working with very large arrays.

ASYNCHRONOUS: When processes communicate with one another, they send messages. If a process decides that it doesn't want to wait for an answer, it is said to run "asynchronously." For example, in most client/

server programs, one program sends an RPC request to a second program and then waits to receive a response from the second program. This is the normal *synchronous* mode of operation. If, however, the first program sends a message and then continues executing, not waiting for a reply, the first mode of operation is said to be *asynchronous*.

AUTOLAUNCHING: When a process begins executing on a remote computer, TotalView can also launch a **tvdsvr** (TotalView Debugger Server) process on the computer that will send debugging information back to the TotalView process that you are interacting with.

AUTOMATIC PROCESS ACQUISITION: TotalView automatically detects the many processes that parallel and distributed programs run in, and attaches to them automatically so you don't have to attach to them manually. If the process is on a remote computer, automatic process acquisition automatically starts the TotalView Debugger Server (**tvdsvr**).

BARRIER: An action point specifying that processes reaching a particular location in the source code should stop and wait for other processes to catch up.

BASE WINDOW: The original Process Window or Variable Window before you dive into routines or variables. After diving, you can use a **Reset** or **Undive** command to restore this original window.

BLOCKED: A thread state in which the thread is no longer executing because it is waiting for an event to occur. In most cases, the thread is blocked because it is waiting for a mutex or condition state.

BREAKPOINT: A point in a program where execution can be suspended to permit examination and manipulation of data.

CALL FRAME: The memory area that contains the variables belonging to a function, subroutine, or other scope division, such as a block.

CALL STACK: A higher-level view of stack memory, interpreted in terms of source program variables and locations. This is where your program places stack frames.

CHILD PROCESS: A process created by another process (*see "parent process"* on page 337) when that other process calls the **fork()** function.

CLOSED LOOP: See *closed loop* on page 332.

CLUSTER DEBUGGING: The action of debugging a program that is running on a cluster of hosts in a network. Typically, the hosts are of the same type and have the same operating system version.

COMMAND HISTORY LIST: A debugger-maintained list that stores copies of the most recent commands issued by the user.

CONDITION SYNCHRONIZATION: A process that delays thread execution until a condition is satisfied.

CONTEXTUALLY QUALIFIED (SYMBOL): A symbol that is described in terms of its dynamic context, rather than its static scope. This includes process

identifier, thread identifier, frame number, and variable or subprocedure name.

CONTROL GROUP: All the processes that a program creates. These processes can be local or remote. If your program uses processes that it did not create, TotalView places them in separate control groups. For example, a client/server program has two distinct executables that run independently of one another. Each would be in a separate control group. In contrast, processes created by the **fork()** function are in the same control group.

CORE FILE: A file that contains 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 into an invalid address).

CORE-FILE DEBUGGING: A debugging session that examines a core file image. Commands that modify program state are not permitted in this mode.

CROSS-DEBUGGING: A special case of remote debugging where the host platform and the target platform are different types of machines.

CURRENT FRAME: The current portion of stack memory, in the sense that it contains information about the subprocedure invocation that is currently executing.

CURRENT LANGUAGE: The source code language used by the file that contains the current source location.

CURRENT LIST LOCATION: The location governing what source code appears in response to a list command.

DATASET: A set of array elements generated by TotalView and sent to the Visualizer. (See *visualizer process* on page 342.)

DBELOG LIBRARY: A library of routines for creating event points and generating event logs from TotalView. To use event points, you must link your program with both the **dbelog** and **elog** libraries.

DBFORK LIBRARY: A library of special versions of the **fork()** and **execve()** calls used by TotalView to debug multiprocess programs. If you link your program with the TotalView **dbfork** library, TotalView can automatically attach to newly spawned processes.

DEBUGGING INFORMATION: Information relating an executable to the source code from which it was generated.

DEBUGGER PROMPT: A string printed by the CLI that indicates that it is ready to receive another user command.

DEBUGGER SERVER: See *tvdsvr process* on page 341.

DEBUGGER STATE: Information that TotalView or the CLI maintains to interpret and respond to user commands. This includes debugger modes, user-defined commands, and debugger variables.

DEPRECATED: A feature that is still available but might be eliminated in a future release.

DISTRIBUTED DEBUGGING: 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.

DIVING: The action of displaying more information about an item. For example, if you dive into a variable in TotalView, a window appears with more information about the variable.

DOPE VECTOR: This is a run time descriptor that contains all information about an object that requires more information than is available as a single pointer or value. For example, you might declare a Fortran 90 pointer variable that is a pointer to some other object, but which has its own upper bound, as follows:

```
integer, pointer, dimension (:) :: iptr
```

Suppose that you initialize it as follows:

```
iptr => iarray (20:1:-2)
```

iptr is a synonym for every other element in the first twenty elements of **iarray**, and this pointer array is in reverse order. For example, **iptr(1)** maps to **iarray(20)**, **iptr(2)** maps to **iarray(18)**, and so on.

A compiler represents an **iptr** object using a run time descriptor that contains (at least) elements such as a pointer to the first element of the actual data, a stride value, and a count of the number of elements (or equivalently, an upper bound).

DPID: Debugger ID. This is the ID TotalView uses for processes.

EDITING CURSOR: A black line that appears when you select a TotalView GUI field for editing. You use field editor commands to move the editing cursor.

EVAL POINT: A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

EVENT LOG: A file that contains a record of events for each process in a program.

EVENT POINT: A point in the program where TotalView writes an event to the event log for later analysis with TimeScan.

EXECUTABLE: A compiled and linked version of source files

EXPRESSION: An expression consists of symbols (possibly qualified), constants, and operators, arranged in the syntax of the current source language. Not all Fortran 90, C, and C++ operators are supported.

EXTENT: 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).

FIELD EDITOR: A basic text editor that is part of TotalView. The field editor supports a subset of GNU Emacs commands.

FOCUS: The set of groups, processes, and threads upon which a CLI command acts. The current focus is indicated in the CLI prompt (if you're using the default prompt).

FRAME: An area in stack memory that contains the information corresponding to a single invocation of a subprocedure. See *stack frame* on page 339.

FRAME POINTER: See *stack pointer* on page 340.

FULLY QUALIFIED (SYMBOL): A symbol is fully qualified when each level of source code organization is included. For variables, those levels are executable or library, file, procedure or line number, and variable name.

GID: The TotalView group ID.

GRID: A collection of distributed computing resources available over a local or wide area network that appears as if it were one large virtual computing system.

GOI: The group of interest. This is the group that TotalView uses when it is trying to determine what to step, stop, and so on.

GROUP: When TotalView starts processes, it places related processes in families. These families are called "groups."

GROUP OF INTEREST: The primary group that is affected by a command. This is the group that TotalView uses when it is trying to determine what to step, stop, and so on.

HEAP: An area of memory that your program uses when it dynamically allocates blocks of memory. It is also how people describe my car.

HOST COMPUTER: The computer on which the TotalView debugger is running.

IMAGE: All of the programs, libraries, and other components that make up your executable.

INFINITE LOOP: See *loop, infinite* on page 336.

INITIAL PROCESS: The process created as part of a load operation, or that already existed in the runtime environment and was attached by TotalView or the CLI.

INITIALIZATION FILE: An optional file that establishes initial settings for debugger state variables, user-defined commands, and any commands that should be executed whenever TotalView or the CLI is invoked. Must be called .tvdrc.

LHS EXPRESSION: This is a synonym for lvalue.

LOCKSTEP GROUP: All threads that are at the same PC (program counter). This group is a subset of a workers group. A lockstep group only exists for stopped threads. All threads in the lockstep group are also in a workers group. By definition, all members of a lockstep group are in the same work-

ers group. That is, a lockstep group cannot have members in more than one workers group or more than one control group.

LOOP, INFINITE: see *infinite loop* on page 335.

LOWER BOUND: 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.

LVALUE: A symbol name or expression suitable for use on the left-hand side of an assignment statement in the corresponding source language. That is, the expression must be appropriate as the target of an assignment.

MACHINE STATE: Convention for describing the changes in memory, registers, and other machine elements as execution proceeds.

MANAGER THREAD: A thread created by the operating system. In most cases, you do not want to manage or examine manager threads.

MESSAGE QUEUE: A list of messages sent and received by message-passing programs.

MIMD: An acronym for Multiple Instruction, Multiple Data, which describes a type of parallel computing.

MISD: An acronym for Multiple Instruction, Single Data, which describes a type of parallel computing.

MPI: An acronym for "Message Passing Interface."

MPICH: MPI/Chameleon (Message Passing Interface/Chameleon) is a freely available and portable MPI implementation. MPICH was written as a collaboration between Argonne National Lab and Mississippi State University. For more information, see http://www.mcs.anl.gov/mpi.

MPMD PROGRAMS: An acronym for Multiple Program, Multiple Data, which describes programs that involve multiple executables, executed by multiple threads and processes.

MUTEX (MUTUAL EXCLUSION): Techniques for sharing resources so that different users do not conflict and cause unwanted interactions.

NATIVE DEBUGGING: The action of debugging a program that is running on the same machine as TotalView.

NESTED DIVE: TotalView lets you dive into pointers, structures, or arrays in a variable. When you dive into one of these elements, TotalView updates the display so that the new element appears. A nested dive is a *dive* within a dive. You can return to the previous display by selecting the left arrow in the top-right corner of the window.

NODE: A machine on a network. Each machine has a unique network name and address.

OUT-OF-SCOPE: When symbol lookup is performed for a particular symbol name and it isn't found in the current scope or any that contains scopes, the symbol is said to be out-of-scope.

PARALLEL PROGRAM: A program whose execution involves multiple threads and processes.

PARALLEL TASKS: Tasks whose computations are independent of each other, so that all such tasks can be performed simultaneously with correct results.

PARALLELIZABLE PROBLEM: A problem that can be divided into parallel tasks. This type of program might require changes in the code and/or the underlying algorithm.

PARCEL: The number of bytes required to hold the shortest instruction for the target architecture.

PARENT PROCESS: A process that calls the fork() function to spawn other processes (usually called child processes).

PARMACS LIBRARY: A message-passing library for creating distributed programs that was developed by the German National Research Centre for Computer Science.

PARTIALLY QUALIFIED (SYMBOL): A symbol name that includes only some of the levels of source code organization (for example, file name and procedure, but not executable). This is permitted as long as the resulting name can be associated unambiguously with a single entity.

PC: An abbreviation for P*rogram Counter.*

PID: Depending on the context, this is either the process ID or the program ID. In most cases, this is the process ID.

POI: The process of interest. This is the process that TotalView uses when it is trying to determine what to step, stop, and so on.

PROCESS: An executable that is loaded into memory and is running (or capable of running).

PROCESS GROUP: A group of processes associated with a multiprocess program. A process group includes program control groups and share groups.

PROCESS/THREAD IDENTIFIER: A unique integer ID associated with a particular process and thread.

PROCESS OF INTEREST: The primary process that TotalView uses when it is trying to determine what to step, stop, and so on.

PROGRAM CONTROL GROUP: A group of processes that includes the parent process and all related processes. A program control group includes children that were forked (processes that share the same source code as the parent), and children that were forked with a subsequent call to the execve() function (processes that don't share the same source code as the parent). Contrast this with *share group* on page 338.

PROGRAM EVENT: A program occurrence that is being monitored by TotalView or the CLI, such as a breakpoint.

PROGRAM STATE: A higher-level view of the machine state, where addresses, instructions, registers, and such are interpreted in terms of source program variables and statements.

P/T (PROCESS/THREAD) SET: The set of threads drawn from all threads in all processes of the target program.

PVM LIBRARY: 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.

RACE CONDITION: A problem that occurs when threads try to simultaneously access a resource. The result can be a deadlock, data corruption, or a program fault.

REMOTE DEBUGGING: 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.

RESUME COMMANDS: Commands that cause execution to restart from a stopped state: **dstep**, **dgo**, **dcont**, **dwait**.

RHS EXPRESSION: This is a synonym for **rvalue**.

RVALUE: An expression suitable for inclusion on the right-hand side of an assignment statement in the corresponding source language. In other words, an expression that evaluates to a value or collection of values.

SATISFACTION SET: The set of processes and threads that must be held before a barrier can be satisfied.

SATISFIED: A condition that indicates that all processes or threads in a group have reached a barrier. Prior to this event, all executing processes and threads are either running because they have not yet hit the barrier, or are being held at the barrier because not all of the processes or threads have reached it. After the barrier is *satisfied*, the held processes or threads are released, which means they can be run. Prior to this event, they could not run.

SCOPE: The region in your program in which a variable or a function exists or is defined. This region begins with its declaration and extends to the end of the current block.

SERIAL EXECUTION: Execution of a program sequentially, one statement at a time.

SERIAL LINE DEBUGGING: A form of remote debugging where TotalView and the TotalView Debugger Server communicate over a serial line.

SERVICE THREAD: A thread whose purpose is to *service* or manage other threads. For example, queue managers and print spoolers are service threads. There are two kinds of service threads: those created by the operating system or runtime system and those created by your program.

SHARE GROUP: All the processes in a control group that share the same code. In most cases, your program has more than one share group. Share groups, like control groups, can be local or remote.

SHARED LIBRARY: A compiled and linked set of source files that are dynamically loaded by other executables.

SIGNALS: Messages informing processes of asynchronous events, such as serious errors. The action that the process takes in response to the signal depends on the type of signal and whether the program includes a signal handler routine, a routine that traps certain signals and determines appropriate actions to be taken by the program.

SIMD: An acronym for Single Instruction, Multiple Data, which describes a type of parallel computing.

SINGLE STEP: The action of executing a single statement and stopping (as if at a breakpoint).

SISD: An acronym for Single Instruction, Single Data, which describes a type of parallel computing.

SLICE: A subsection of an array, which is expressed in terms of a *lower bound* on page 336, *upper bound* on page 342, and *stride* on page 340. Displaying a slice of an array can be useful when you are working with very large arrays.

SOID: An acronym for symbol object ID. A SOID uniquely identifies all TotalView information. It also represents a handle by which you can access this information.

SOURCE FILE: Program file that contains source language statements. TotalView lets you debug FORTRAN 77, Fortran 90, Fortran 95, C, C++, and assembler files.

SOURCE LOCATION: For each thread, the source code line it executes next. This is a static location, indicating the file and line number; it does not, however, indicate which invocation of the subprocedure is involved.

SPAWNED PROCESS: The process created by a user process executing under debugger control.

SPMD PROGRAMS: An acronym for Single Program, Multiple Data, which describe a type of parallel computing that involves just one executable, executed by multiple threads and processes.

STACK: A portion of computer memory and 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.

STACK FRAME: Whenever your program calls a function, it creates a set of information that includes 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. The information for one function is called a "stack frame" as it is placed on your program's stack.

When your program begins executing, it has only one frame: the one allocated for function main(). As your program calls functions, new frames are allocated. When a function returns to the function from which it is called, the frame is deallocated.

STACK POINTER: A pointer to the area of memory where subprocedure arguments, return addresses, and similar information is stored. This is also called a frame pointer.

STACK TRACE: A sequential list of each currently active routine called by a program, and the frame pointer that points to its stack frame.

STATIC (SYMBOL) SCOPE: A region of a program's source code that has a set of symbols associated with it. A scope can be nested inside another scope.

STEPPING: Advancing program execution by fixed increments, such as by source code statements.

STL: An acronym for Standard Template Library.

STOP SET: A set of threads that TotalView stops after an action point triggers.

STOPPED/HELD STATE: The state of a process whose execution has paused in such a way that another program event (for example, arrival of other threads at the same barrier) is required before it is capable of continuing execution.

STOPPED/RUNNABLE STATE: The state of a process whose execution has been paused (for example, when a breakpoint triggered or due to some user command) but can continue executing as soon as a resume command is issued.

STOPPED STATE: The state of a process that is no longer executing, but will eventually execute again. This is subdivided into stopped/runnable and stopped/held.

STRIDE: The interval between array elements in a slice and the order in which TotalView displays these elements. If the stride is 1, TotalView displays every element between the lower bound and upper bound of the slice. If the stride is 2, TotalView displays every other element. If the stride is –1, TotalView displays every element between the upper bound and lower bound (reverse order).

SYMBOL: Entities within program state, machine state, or debugger state.

SYMBOL LOOKUP: Process whereby TotalView consults its debugging information to discover what entity a symbol name refers to. Search starts with a particular static scope and occurs recursively so that contains scopes are searched in an outward progression.

SYMBOL NAME: The name associated with a symbol known to TotalView (for example, function, variable, data type, and so on).

SYMBOL TABLE: A table of symbolic names used in a program (such as variables or functions) and their memory locations. The symbol table is part of the executable object generated by the compiler (with the –g option) and is used by debuggers to analyze the program.

SYNCHRONIZATION: A mechanism that prevents problems caused by concurrent threads manipulating shared resources. The two most common

mechanisms for synchronizing threads are *mutual exclusion* and *condition synchronization*.

TARGET COMPUTER: The computer on which the process to be debugged is running.

TARGET PROCESS SET: The target set for those occasions when operations can only be applied to entire processes, not to individual threads in a process.

TARGET PROGRAM: The executing program that is the target of debugger operations.

TARGET P/T SET: The set of processes and threads that a CLI command acts on.

TASK: A logically discrete section of computational work. (This is an informal definition.)

THREAD: 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.

THREAD EXECUTION STATE: The convention of describing the operations available for a thread, and the effects of the operation, in terms of a set of predefined states.

THREAD OF INTEREST: The primary thread affected by a command. This is abbreviated as TOI.

TID: The thread ID.

TLA: An acronym for Three-Letter Acronym. So many things from computer hardware and software vendors are referred to by a three-letter acronym that yet another acronym was created to describe these terms.

TOI: The thread of interest. This is the primary thread affected by a command.

TRIGGER SET: The set of threads that can trigger an action point (that is, the threads upon which the action point was defined).

TRIGGERS: The effect during execution when program operations cause an event to occur (such as arriving at a breakpoint).

TTF: See *type transformation facility* on page 341.

TVDSVR PROCESS: The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.

TYPE TRANSFORMATION FACILITY: This is abbreviated as TTF. A TotalView subsystem that allows you to change the way information appears. For example, an STL vector can appear as an array.

UNDISCOVERED SYMBOL: A symbol that is referred to by another symbol. For example, a **typedef** is a reference to the aliased type.

UNDIVING: The action of displaying the previous contents of a window, instead of the contents displayed for the current dive. To undive, you click the **undive** icon in the upper-right corner of the window.

UPPER BOUND: The last element in the dimension of an array or the slice of an array.

USER THREAD: A thread created by your program.

USER INTERRUPT KEY: A keystroke used to interrupt commands, most commonly defined as **^C** (Ctrl+C).

VARIABLE WINDOW: A TotalView window that displays the name, address, data type, and value of a particular variable.

VISUALIZER PROCESS: A process that works with TotalView in a separate window, allowing you to see a graphic representation of program array data.

WATCHPOINT: An action point that tells TotalView to stop execution when the value of a memory location changes.

WORKER THREAD: A thread in a workers group. These are threads created by your program that performs the task for which you've written the program.

WORKERS GROUP: All the worker threads in a control group. Worker threads can reside in more than one share group.

Index

Symbols

scope separator character 276
$clid built-in variable 322
$count built-in function 6, 308, 311, 323
$countall built-in function 323
$countthread built-in function 323
$debug assembler pseudo op 328
$denorm filter 285
$duid built-in variable 322
$hold assembler pseudo op 328
$hold built-in function 323
$holdprocess assembler pseudo op 328
$holdprocess built-in function 323
$holdprocessall built-in function 323
$holdprocessstopall assembler pseudo op 328
$holdstopall assembler pseudo op 328
$holdstopall built-in function 323
$holdthread assembler pseudo op 328
$holdthread built-in function 324
$holdthreadstop assembler pseudo op 328
$holdthreadstop built-in function 324
$holdthreadstopall assembler pseudo op 328
$holdthreadstopall built-in function 324
$holdthreadstopprocess assembler pseudo op 328
$holdthreadstopprocess built-in function 324
$inf filter 285
$long_branch assembler pseudo op 328
$nan filter 285
$nanq filter 285
$nans filter 285
$ndenorm filter 285
$newval built-in function 317
$newval built-in variable 322
$nid built-in variable 322
$ninf filter 285
$oldval built-in function 317
$oldval built-in variable 322
$pdenorm filter 285
$pid built-in variable 322
$pinf filter 285
$processduid built-in variable 322
$stop assembler pseudo op 328
$stop built-in function 6, 311, 318, 324
$stopall assembler pseudo op 328
$stopall built-in function 324
$stopprocess assembler pseudo op 328
$stopprocess built-in function 324
$stopthread assembler pseudo op 328
$stopthread built-in function 324
$systid built-in variable 322
$tid built-in variable 322
$visualize built-in function 143, 144, 324
in animations 143
using casts 144
$visualize function 144
%C server launch replacement characters 68
%D bulk server launch command 69
%D single process server launch command 69
%H bulk server launch command 70
%L bulk server launch command 70
%L single process server launch command 69
%N bulk server launch command 71
%P bulk server launch command 70
%P single process server launch command 69
%R single process server launch command 68
%t1 bulk server launch command 71
%t2 bulk server launch command 71
%V bulk server launch command 70
& intersection operator 228

. (dot) current set indicator 214, 229
. (period), in suffix of process names 189
.rhosts file 72, 84
.totalview subdirectory 38
.tvdrc initialization files 38
.Xdefaults file 39, 59
  autoLoadBreakpoints 59
  deprecated resources 59
/ slash in group specifier 218
/usr/lib/array/arrayd.conf file 70
: (colon), in array type strings 262
: as array separator 280
< first thread indicator (CLI) 213
<address> data type 263
<char> data type 263
<character> data type 263
<code> 248
<code> data type 247, 264, 266
<complex*16> data type 264
<complex*8> data type 264
<complex> data type 264
<double precision> data type 264
<double> data type 264
<extended> data type 264
<float> data type 264
<int> data type 264
<integer*1> data type 264
<integer*2> data type 264
<integer*4> data type 264
<integer*8> data type 264
<integer> data type 264
<logical*1> data type 264
<logical*2> data type 264
<logical*4> data type 264
<logical*8> data type 264
<logical> data type 264
<long long> data type 264
<long> data type 264
<opaque> data type 265
<real*16> data type 265
<real*4> data type 265
<real*8> data type 265
<real> data type 265
<short> data type 265
<string> data type 260, 261, 265
<void> data type 265
> (right angle bracket), indicating nested dives 250
@ action point marker, in CLI 295
– difference operator 228
| union operator 228
` module separator 271

A

–a command-line option 37, 171
  passing arguments to program 37
a width specifier 218
  general discussion 220
absolute addresses, display assembler as 131
acquiring processes 85
Action Point > At Location command 4, 295, 297
Action Point > Delete All command 298
Action Point > Properties command 5, 118, 192, 294, 297, 298, 300, 302, 304, 306, 308
  deleting barrier points 306
Action Point > Properties Dialog Box figure 297, 300, 304
Action Point > Save All command 85, 319
Action Point > Save As command 319
Action Point > Set Barrier command 304
Action Point > Suppress All command 299
action point files 39
action point identifiers 176
  never reused in a session 176
Action Point Symbol figure 294
action points 176
  see also barrier points
  see also eval points
  common properties 294
  definition 5, 293
  deleting 298
  disabling 298
  enabling 298
  evaluation points 5
  ignoring 299
  list of 130
  multiple addresses 295
  saving 319
  suppressing 299
  unsuppressing 299
  watchpoint 10
Action Points Page 116, 130
Action Points Pane 298
adapter_use option 83
Add to Expression List command 10
Add to Expression List context menu command 255
adding command-line arguments 53
adding environment variables 60
adding members to a group 216
adding program arguments 37
address range conflicts 312
addresses
  changing 267
  editing 267
  specifying in variable window 246
  tracking in variable window 239
advancing and holding processes 175
advancing program execution 175
aggregates, in Expression List window 255
aliases
  built-in 173
  group 173
  group, limitations 173
align assembler pseudo op 328
all width specifier 214
allocated arrays, displaying 266
altering groups 231
Ambiguous Function Dialog Box 297
Ambiguous Function Dialog Box figure 181, 297
ambiguous function names 181, 297

Ambiguous Line Dialog Box figure 296
ambiguous locations 297
ambiguous names 182
ambiguous scoping 277
ambiguous source lines 192
angle brackets, in windows 250
animation using $visualize 143
areas of memory, data type 265
arena specifiers 213
  defined 213
  incomplete 225
  inconsistent widths 226
arenas
  and scope 206
  defined 206, 213
  iterating over 213
ARGS variable 171
  modifying 171
ARGS_DEFAULT variable 37, 53, 171
  clearing 172
arguments
  in server launch command 68, 72
  passing to program 37
  replacing 172
  setting 53
Arguments page 53
argv, displaying 266
Array Data Filter by Range of Values figure 287
array data filtering
  by comparison 283
  by range of values 286
  for IEEE values 285
Array Data Filtering by Comparison figure 285
Array Data Filtering for IEEE Values figure 286
array data filtering, *see* arrays, filtering
array of structures 249
  displaying 252
  in Expression List window 255
array pointers 245
array rank 140
array services handle (ash) 88
Array Statistics Window figure 289
arrays
array data filtering 283
bounds 261
casting 262
character 265
checksum statistic 289
colon separators 280
count statistic 289
deferred shape 272, 280
denormalized count statistic 289
display subsection 262
displaying 144, 279
displaying allocated 266
displaying argv 266
displaying contents 133
displaying declared 266
displaying multiple 144
displaying one element 282
displaying slices 279
diving into 249
editing dimension of 262
extent 262
filter conversion rules 284
filter expressions 287
filtering 262, 283, 284, 285
filtering options 283
in C 262
in Fortran 262
infinity count statistic 289
laminating 291
limiting display 282
lower adjacent statistic 289
lower bound of slices 280
lower bounds 261, 262
maximum statistic 290
mean statistic 290
median statistic 290
minimum statistic 290
NaN statistic 290
non-default lower bounds 262
overlapping nonexistent memory 279
pointers to 261
quartiles statistic 290
skipping elements 281
slice 282
slice example 280, 281
slice, initializing 158
slice, printing 159
slice, refining 144
slices with the variable command 282
slicing 8
sorting 288
standard deviation statistic 290
statistics 289
stride 280
stride elements 280
subsections 279
sum statistic 290
type strings for 261
upper adjacent statistic 290
upper bound 261
upper bound of slices 280
visualization 144
visualizing 142
zero count statistic 290
arrow buttons 8
arrow over line number 130
ascii assembler pseudo op 328
asciz assembler pseudo op 328
ash (array services handle) 88
ASM icon 294, 299
assembler
  absolute addresses 131
  and –g compiler option 133
  constructs 326
  displaying 131
  examining 131
  expressions 327
  in code fragment 306
  symbolic addresses 131
Assembler > By Address command 131
Assembler > Symbolically command 131, 132
Assembler command 131
assigning output 170
assigning output to variable 170
assigning p/t set to variable 215
asynchronous processing 16
at breakpoint state 47
At Location command 4, 295, 297
Attach Subsets command 113
Attached Page 86, 124, 126

attached process states 47
attaching
restricting 113
restricting by communicator 114
selective 113
to a task 108
to all 115
to HP MPI job 83
to job 85
to MPICH application 80
to MPICH job 80
to none 115
to PE 85
to poe 86
to processes 42, 43, 85, 107, 113, 127
to PVM task 107
to relatives 44
to RMS processes 87
to SGI MPI job 88
attaching using File > New Program 44
Auto Visualize, in Directory Window 146
auto_array_cast_bounds variable 246
auto_deref_in_all_c variable 246
auto_deref_in_all_fortran variable 246
auto_deref_initial_c variable 246
auto_deref_initial_fortran variable 246
auto_deref_nested_c variable 246
auto_deref_nested_fortran variable 246
auto_save_breakpoints variable 319
autolaunch 63, 64
changing 71
defined 42
disabling 42, 64, 65, 71
launch problems 67
sequence 72
autolaunching 72
autoLoadBreakpoints .Xdefault 59
automatic dereferencing 245
automatic process acquisition 79, 83, 106
averaging data points 151
averaging surface display 151
axis, transposing 148

B

B state 47
backtick separator 271
backward icon 134
barrier points
see also process barrier breakpoint 13, 31, 185, 303, 304
clearing 298
defined 176
defined (again) 303
deleting 306
satisfying 305
states 303
stopped process 306
baud rate, for serial line 74
bit fields 260
block scoping 274
blocking send operations 94
blocks
displaying 241
naming 276
bold data 7
Both command 131, 202
bounds for arrays 261
boxed line number 130, 206, 295
Breakpoint at Assembler Instruction figure 299
breakpoint files 39
breakpoint operator 228
breakpoints
and MPI_Init() 85
apply to all threads 294
automatically copied from master process 79
behavior when reached 300
changing for parallelization 116
clearing 124, 206, 298
conditional 306, 307, 308, 323
copy, master to slave 79
countdown 308, 323
counting down 323
default stopping action 116
defined 176, 293
deleting 298
disabling 298
enabling 298
entering 88
example setting in multiprocess program 303
fork() 302
hitting within eval point 321
ignoring 299
in child process 300
in parent process 300
in spawned process 107
listing 130
machine-level 299
multiple processes 300
not shared in separated children 302
placing 130
reloading 85
removed when detaching 45
removing 124
saving 319
set while a process is running 295
set while running parallel tasks 84
setting 84, 124, 161, 206, 295, 300
shared by default in processes 302
sharing 301, 302
stop all related processes 300
suppressing 299
thread-specific 322
toggling 295
while stepping over 194
bss assembler pseudo op 329
built-in aliases 173
built-in functions
$count 6, 308, 311, 323
$countall 323
$countthread 323
$hold 323
$holdprocess 323
$holdprocessall 323
$holdstopall 323
$holdthread 324
$holdthreadstop 324
$holdthreadstopall 324
$holdthreadstopprocess 324
$stop 6, 311, 318, 324
$stopall 324
$stopprocess 324
$stopthread 324
$visualize 143, 144, 324
forcing interpretation 310
built-in type strings 263
built-in variables 322

$clid 322
$duid 322
$newval 322
$nid 322
$oldval 322
$pid 322
$processduid 322
$systid 322
$tid 322
forcing interpretation 323
Bulk Launch Page 67
Bulk Launch page 67
bulk server launch 63, 65
command 65
connection timeout 66
on HP Alpha 70
on IBM RS/6000 71
on SGI MIPS 69
bulk server launch command
%D 69
%H 70
%L 70
%N 71
%P 70
%t1 71
%t2 71
%V 70
–callback_host 70
–callback_ports 70
–set_pws 70
–verbosity 70
–working_directory 69
bulk_incr_timeout variable 67
bulk_launch_base_timeout variable 67
bulk_launch_enabled variable 65, 67
bulk_launch_incr_timeout variable 67
bulk_launch_string variable 66
bulk_launch_tmpefile1_trailer_ line variable 66
bulk_launch_tmpefile2_trailer_ line variable 66
bulk_launch_tmpfile1_header_ line variable 66
bulk_launch_tmpfile1_header_line variable 66
bulk_launch_tmpfile1_host_ lines variable 66
bulk_launch_tmpfile1_host_line variable 66
bulk_launch_tmpfile1_trailer_line variable 66
bulk_launch_tmpfile2_header_ line variable 66
bulk_launch_tmpfile2_header_line variable 66
bulk_launch_tmpfile2_host_ lines variable 66
bulk_launch_tmpfile2_host_line variable 66
bulk_launch_tmpfile2_trailer_line variable 66
By Address command 131
byte assembler pseudo op 329

C

C casting for Global Arrays 101, 102
C control group specifier 217, 218
C language
array bounds 262
arrays 262
filter expression 287
how data types are displayed 260
in code fragment 306
in evaluation points 324
type strings supported 260
C++
changing class types 269
display classes 268
call stack 130
call tree
updating display 137
Call Tree command 137
–callback command-line option 71
–callback_host bulk server launch command 70
–callback_option single process server launch command 69
–callback_ports bulk server launch command 70
capture command 170, 171
casting 260, 261
examples 266
to type 248
types of variable 260
casting arrays 262
casting Global Arrays 101, 102
CGROUP variable 216, 222
ch_lfshmem device 78
ch_mpl device 78
ch_p4 device 78, 80, 118
ch_shmem device 78, 80
changing autolaunch options 64
changing command-line arguments 53
changing expression in 253
changing groups 231
changing precision 238
changing process thread set 212
changing program state 166
changing remote shell 71
changing size 238
changing threads in Variable window 248
changing values 135
changing variables 259
character arrays 265
chasing pointers 245, 249
checksum array statistic 289
child process names 189
children calling execve(), *see* execve()
classes, displaying 268
Clear All STOP and EVAL command 298
clearing
breakpoints 124, 206, 298, 300
continuation signal 197
evaluation points 124
CLI
and Tcl 165
components 165
in startup file 168
initialization 168
interface 166
introduced 13
invoking program from shell example 168
not a library 165
output 170
relationship to TotalView 166
starting 37, 167
starting from command prompt 167
starting from TotalView GUI 167

starting program using 168
CLI and Tcl relationship 166
CLI commands
assigning output to variable 170
capture 170, 171
dactions 294
dactions –load 85, 319
dactions –save 85, 319
dassign 259
dattach 37, 41, 43, 44, 46, 80, 85, 86, 89, 175
dattach mprun 89
dbarrier 303, 305
dbarrier –e 308
dbarrier –stop_when_hit 118
dbreak 295, 297, 300
dbreak –e 308
dcheckpoint 198
ddelete 94, 297, 298, 306
ddetach 45
ddisable 298, 299, 306
ddlopen 199
ddown 195
default focus 212, 213
denable 298, 299
dfocus 193, 211, 212
dga 101
dgo 81, 84, 85, 88, 116, 190, 191, 226
dgroups –add 216, 222
dhalt 117, 184, 194
dhold 186, 304
dhold –thread 186
dkill 118, 169, 175, 197
dlist 107
dload 37, 41, 42, 68, 168, 169, 175
dnext 117, 191, 195
dnexti 192, 195
dout 196, 207
dprint 98, 99, 181, 203, 239, 240, 245, 246, 247, 263, 266, 269, 270, 271, 280, 282
dptsets 46, 190
drerun 169, 197
redirecting I/O 54
drestart 198
drun 168, 171, 172
redirecting I/O 54
dset 171, 173
dstatus 46, 305
dstep 117, 191, 194, 207, 213, 215, 226
dstepi 191, 194
dunhold 186, 304
dunhold –thread 186
dunset 172
duntil 192, 195, 207, 209
dup 195, 240
dwhere 214, 226, 240
exit 40
read_symbols 202
run when starting TotalView 39
CLI prompt 169
CLI variables
ARGS 171
ARGS, modifying 171
ARGS_DEFAULT 37, 53, 171
clearing 172
auto_array_cast_bounds 246
auto_deref_in_all_c 246
auto_deref_in_all_fortran 246
auto_deref_initial_c 246
auto_deref_initial_fortran 246
auto_deref_nested_c 246
auto_deref_nested_fortran 246
auto_save_breakpoints 319
bulk_incr_timeout 67
bulk_launch_base_timeout 67
bulk_launch_enabled 65, 67
bulk_launch_incr_timeout 67
bulk_launch_string 66
bulk_launch_tmpefile1_trailer_line 66
bulk_launch_tmpefile2_trailer_line 66
bulk_launch_tmpfile1_header_ line 66
bulk_launch_tmpfile1_header_line 66
bulk_launch_tmpfile1_host_ lines 66
bulk_launch_tmpfile1_host_line 66
bulk_launch_tmpfile1_trailer_line 66
bulk_launch_tmpfile2_header_ line 66
bulk_launch_tmpfile2_header_line 66
bulk_launch_tmpfile2_host_line 66
bulk_launch_tmpfile2_host_lines 66
bulk_launch_tmpfile2_trailer_line 66
data format 238
dll_read_all_symbols 201
dll_read_loader_symbols_only 201
dll_read_no_symbols 201
dpvm 105, 106
EXECUTABLE_PATH 42, 44, 50, 52, 105, 158
LINES_PER_SCREEN 171
parallel_attach 116
parallel_stop 115
pop_at_breakpoint 50
pop_on_error 49
process_load_callbacks 40
PROMPT 173
pvm 106
server_launch_enabled 64, 67, 71
server_launch_string 65
server_launch_timeout 65
SHARE_ACTION_POINT 298, 301, 302
signal_handling_mode 49
STOP_ALL 298, 300
suffixes 36
ttf 237
warn_step_throw 49
$clid built-in variable 322
Close command 134, 249
Close command (Visualizer) 146
Close Relatives command 134
Close Similar command 134, 249
Close, in Data Window 146
closed loop, *see* closed loop

closing similar windows 134
closing variable windows 249
closing windows 134
cluster ID 322
code constructs supported
  Assembler 326
  C 324
  Fortran 325
<code> data type 247, 266
code fragments 306, 320, 322
  modifying instruction path 307
  when executed 307
  which programming languages 306
collapsing structures 240
colons as array separators 280
columns
  displaying 257
comm assembler pseudo op 329
command arguments 171
  clearing example 171
  passing defaults 172
  setting 171
command line arguments 53, 169
  passing to TotalView 37
Command Line command 37, 167
command line-options
  launch Visualizer 152
command prompts 172
  default 172
  format 172
  setting 173
  starting the CLI from 167
command scope 275
command-line options
  –a 37, 171
  –remote 38, 65
  –s startup 168
commands 37
  Action Point > At Location 4, 295
  Action Point > Delete All 298
  Action Point > Properties 118, 298, 300, 302, 304, 306
  Action Point > Save All 85, 319
  Action Point > Save As 319
  Action Point > Set Barrier 304
  Action Point > Suppress All 299
  arguments 53
  Auto Visualize (Visualizer) 146
  change Visualizer launch 141
  Clear All STOP and EVAL 298
  CLI, *see* CLI commands
  dmpirun 81, 82
  dpvm 105
  Edit > Copy 136
  Edit > Cut 136
  Edit > Delete 136
  Edit > Delete All Expressions 258
  Edit > Delete Expression 258
  Edit > Duplicate Expression 259
  Edit > Find 4, 180
  Edit > Find Again 180
  Edit > Paste 136
  Edit > Reset Defaults 258
  Edit > Undo 136
  File > Close 134, 249
  File > Close (Visualizer) 146
  File > Close Similar 134, 249
  File > Delete (Visualizer) 146
  File > Directory (Visualizer) 146
  File > Edit Source 183
  File > Exit (Visualizer) 146
  File > New Base Window (Visualizer) 147
  File > New Program 40, 42, 44, 46, 65, 68, 71, 75
  File > Options (Visualizer) 147, 148
  File > Preferences 54
    Formatting page 238
    Launch Strings page 140
    Options page 237
    Pointer Dive page 245
  File > Save Pane 136
  File > Search Path 42, 44, 51, 52, 85, 105
  File > Signals 49
  Group > Attach Subsets 113
  Group > Control > Go 185
  Group > Delete 94, 197
  Group > Edit 216, 231
  Group > Go 85, 116, 190, 191, 302
  Group > Halt 117, 194
  Group > Hold 186
  Group > Next 117
  Group > Release 186
  Group > Restart 197
  Group > Run To 116
  Group > Share > Halt 184
  Group > Step 117
  Group > Workers > Go 190
  group or process 116
  input and output files 53
  interrupting 167
  Load All Symbols in Stack 202
  mpirun 83, 88
  poe 79, 83
  Process > Create 191
  Process > Detach 45
  Process > Go 81, 82, 84, 87, 88, 116, 190, 191, 197
  Process > Halt 117, 184, 194
  Process > Hold 186
  Process > Next 191
  Process > Next Instruction 192
  Process > Out 207
  Process > Run To 192, 207
  Process > Startup 37
  Process > Startup Parameters 53
  Process > Step 191
  Process > Step Instruction 191
  Process Startup Parameters, Environment Page 60
  Process Startup Parameters, Environment page 60
  prun 87
  pvm 104, 105
  remsh 71
  rsh 71, 84
  server launch, arguments 68

Set Signal Handling Mode 105, 106
single-stepping 193
Startup 37
step 4
Thread > Continuation Signal 45, 196
Thread > Go 191
Thread > Hold 186
Thread > Set PC 203
Tools > Call Tree 137
Tools > Command Line 167
Tools > Create Checkpoint 198
Tools > Evaluate 140, 144, 199, 253, 319, 320
Tools > Evaluate, *see* Expression List window
Tools > Global Arrays 101
Tools > Laminate 113
Tools > Manage Shared Libraries 199
Tools > Message Queue 90, 91
Tools > Message Queue Graph 12, 90
Tools > P/T Set Browser 229
Tools > Program Browser 239
Tools > PVM Tasks 107
Tools > Restart 198
Tools > Statistics 289
Tools > Thread Objects 274
Tools > Variable Browser 243
Tools > Visualize 9, 143
Tools > Visualize Distribution 112
Tools > Watchpoint 10, 317
totalview 37, 81, 84, 88
  core files 37, 45
totalviewcli 37, 88
tvdsvr 63
  launching 68
View > Add to Expression List 255
View > Assembler > By Address 131
View > Assembler > Symbolically 131, 132
View > Collapse All 240
View > Display Manager Threads 126
View > Dive 258
View > Dive In All 251, 252
View > Dive in New Window 8
View > Dive Thread 274
View > Dive Thread New 274
View > Expand All 240
View > Graph (Visualizer) 146
View > Laminate > None 291
View > Laminate > Process 290
View > Laminate > Thread 290
View > Lookup Function 180, 182, 183
View > Lookup Variable 239, 245, 246, 271, 282, 283
View > Reset 182, 183
View > Reset (Visualizer) 149, 151
View > Source As > Assembler 131
View > Source As > Both 131, 202
View > Source As > Source 131
View > Surface (Visualizer) 146
View > Variable 98
View >Lookup 107
Visualize 9
visualize 141, 152
Window > Duplicate 134, 251
Window > Memorize 135
Window > Memorize All 135
Window > Update 86, 185
Windows > Update (PVM) 107
common block
  displaying 269
  diving on 269
  members have function scope 269
compiled expressions 310, 311
  allocating patch space for 311
  performance 310
compilers
  mpcc_r 91
  mpxlf_r 91
  mpxlf90_r 91
compiling
  –g compiler option 35, 36
  multiprocess programs 35
  –O option 36
  optimization 36
  programs 3, 35
completion rules for arena specifiers 225
compound objects 261
conditional breakpoints 306, 307, 308, 323
conditional watchpoints, *see* watchpoints
conf file 70
configure command 78
configuring the Visualizer 140
connection for serial line 74
connection timeout 65, 66
  altering 64
connection timeout, bulk server launch 66
contained functions 271
  displaying 271
context menus 123
  Add to Expression 10
continuation signal 197
  clearing 197
Continuation Signal command 45, 196
continuing with a signal 196
continuous execution 167
contour lines 151
contour settings 150
Control Group and Share Groups Examples figure 189
control groups 24, 188
  defined 22
  discussion 189
  overview 216
  specifier for 217
control in parallel environments 175
control in serial environments 175
control registers 203

interpreting 203
controlling program execution 175
conversion rules for filters 284
Copy command 136
copying 136
copying between windows 136
core dump, naming the signal that caused 46
core files
debugging 37
examining 45
in totalview command 37, 45
loading 41
multi-threaded 46
correcting programs 310
count array statistic 289
$count built-in function 323
$countall built-in function 323
countdown breakpoints 308, 323
$countthread built-in function 323
CPU registers 203
cpu_use option 83
Create Checkpoint command 198
creating groups 26, 190
creating new processes 169
creating process
without starting it 191
creating processes 53, 190
and starting them 190
using Step 191
without starting them 191
creating threads 18
creating type transformations 237
crt0.o module 107
Ctrl+C 167
current focus 230
current location of program counter 130
current queue state 90
current set indicator 214, 229
current stack frame 182
current working directory 51, 52
Cut command 136

D

D control group specifier 217
dactions command 294
–load 85, 319
–save 85, 319
daemons 16, 18
dassign command 259
data
editing 7
examining 7
filtering 9
slicing 8
viewing, from Visualizer 142
data assembler pseudo op 329
data filtering, *see* arrays, filtering
data precision, changing display 58
data types
see also TotalView data types
C++ 268
changing 260
changing class types in C++ 269
for visualization 142
int 261
int* 261
int[] 261
opaque data 265
pointers to arrays 261
predefined 263
to visualize 142
data watchpoints, *see* watchpoints
data window (Visualizer) 146
display commands 147
scaling 149
translating 149
zooming 149
data window, *see* Variable Window
data_format variables 238
dataset
defined for Visualizer 142
deleting 145, 146
showing parameters 152
visualizing 144
dattach command 37, 41, 43, 44, 46, 80, 85, 86, 89, 175
mprun command 89
dbarrier command 303, 305
–e 308
–stop_when_hit 118
dbfork library 36, 302
linking with 36
dbreak command 295, 297, 300
–e 308
dcheckpoint command 198
ddelete command 94, 297, 298, 306
ddetach command 45
ddisable command 298, 299, 306
ddlopen command 199
ddown command 195
deadlocks 209
message passing 91
$debug assembler pseudo op 328
–debug, using with MPICH 94
debugger initialization 168
debugger PID 175
debugger server 63
see also, tvdsvr
starting manually 67
Debugger Unique ID (DUID) 322
debugging
core file 37
executable file 37
multiprocess programs 36
not compiled with –g 36
on a remote host 41
OpenMP applications 95
over a serial line 74
PE applications 83
programs that call execve 36
programs that call fork 36
PVM applications 103, 104
QSW RMS 87
SHMEM library code 109
UPC programs 110
debugging Fortran modules 271
debugging session 175
debugging symbols, reading 200
debugging techniques 30, 94, 113
declared arrays, displaying 266

def assembler pseudo op 329
default address range conflicts 312
default control group specifier 217
default focus 223
default process/thread set 212
default programming language 36
default text editor 183
default width specifier 214
deferred shape array
  definition 280
  types 272
deferred symbols
  force loading 201
  reading 200
deferring order for shared libraries 201
Delete All command 298
Delete command 117, 136, 197
Delete command (Visualizer) 146
Delete, in Data Window 146
deleting
  action points 298
  datasets 146
  groups 231
  processes 308
  programs 197
denable command 298, 299
denorm filter 285
denormalized count array statistic 289
DENORMs 283
deprecated X defaults 59
deprecated, defined 59
dereferencing 8
  automatic 245
  controlling 59
  pointers 245
Detach command 45
detaching 114
detaching from processes 45
detaching removes all breakpoints 45
determining scope 205, 242
dfocus command 193, 211, 212
  example 212
dga command 101
dgo command 81, 84, 85, 88, 116, 190, 191, 226
dgroups command
  –add 222
  –add command 216
  –remove 31
dhalt command 117, 184, 194
dhold command 186, 304
  –process 186
  –thread 186
difference operator 228
directories, setting order of search 50
Directory command (Visualizer) 146
directory search path 105
Directory Window, menu commands 145
Directory, in Data Window 146
disabling
  action points 298
  autolaunch 64, 71
  autolaunch feature 65
  visualization 140
disassembled machine code 181
  in variable window 248
discard dive stack 182
discard mode for signals 50
discarding signal problem 50
disconnected processing 16
displaying 133
  areas of memory 246
  argv array 266
  array data 133
  arrays 279
  blocks 241
  columns 257
  common blocks 269
  declared and allocated arrays 266
  exited threads 126
  Fortran data types 269
  Fortran module data 269
  global variables 239, 243
  long variable names 241
  machine instructions 247, 248
  memory 246
  pointer 133
  pointer data 133
  Process window 133
  registers 244
  remote hostnames 125
  stack trace pane 133
  STL variables 235
  structs 262
  subroutines 133
  thread objects 274
  typedefs 262
  unions 263
  variable 133
  Variable Windows 238
distributed debugging
  see also PVM applications
  remote server 63
dive icon 134, 250
Dive In All command 251, 252
Dive in New Window command 8
Dive Thread command 274
Dive Thread New command 274
dividing work up 16
diving 85, 90, 124, 133, 239
  defined 7
  from groups page 190
  in a laminated pane 292
  in a variable window 249
  in source code 182
  into a pointer 133, 249
  into a process 133
  into a stack frame 133
  into a structure 249
  into a thread 133
  into a variable 7, 133
  into an array 249
  into formal parameters 244
  into Fortran common blocks 269
  into function name 182
  into global variables 239, 243
  into local variables 244
  into MPI buffer 93
  into MPI processes 92
  into parameters 244
  into pointer 133
  into processes 43, 133
  into PVM tasks 107
  into registers 244
  into routines 133
  into the PC 248
  into threads 130, 133
  into variables 133
  nested 133

nested dive defined 249
program browser 243
scoping issue 242
using middle mouse button 136
dkill command 118, 169, 175, 197
dlist command 107
dll_read_all_symbols variable 201
dll_read_loader_symbols variable 201
dll_read_loader_symbols_only variable 201
dll_read_no_symbols variable 201
dload command 37, 41, 42, 68, 168, 169, 175
returning process ID 170
dlopen(), using 198
DMPI 91
dmpirun command 81, 82
dnext command 117, 191, 195
dnexti command 192, 195
double assembler pseudo op 329
dout command 196, 207
dpid 175
dprint command 98, 99, 181, 203, 239, 240, 245, 246, 247, 263, 266, 269, 270, 271, 280, 282
dptsets command 46, 190
DPVM
see also PVM
enabling support for 105
must be running before TotalView 105
starting session 105
–dpvm command-line option 105
dpvm shell command 105
dpvm variable 105, 106
drawing options 148
drerun command 169, 197
redirecting I/O 54
drestart command 198
drun command 168, 171, 172
redirecting I/O 54
dset command 171, 173
dstatus command 46, 305
dstep command 191, 194, 207, 213, 215, 226
dstep commands 117
dstepi command 191, 194
DUID 322
of process 322
$duid built-in variable 322
dunhold command 186, 304
–thread 186
dunset command 172
duntil command 192, 195, 207, 209
dup command 195
dup commands 240
Duplicate command 134, 251
dwhere command 214, 226, 240
dynamic call tree 137
Dynamic Libraries page 200
dynamic patch space allocation 311
dynamically linked, stopping after start() 107

E

E state 47
Edit > Copy command 136
Edit > Cut command 136
Edit > Delete All Expressions command 258
Edit > Delete command 136
Edit > Delete Expression command 258
Edit > Duplicate Expression command 259
Edit > Find Again command 180
Edit > Find command 4, 180
Edit > Paste command 136
Edit > Reset Defaults command 258
Edit > Undo command 136
edit mode 124
Edit Source command 183
editing
addresses 267
compound objects or arrays 261
laminated pane 292
source text 183
text 135
type strings 260
EDITOR environment variable 183
editor launch string 183
effects of parallelism on debugger behavior 174
Enable action point 298
Enable Single Debug Server Launch check box 71
Enable Visualizer Launch check box 140
enabling
action points 298
Environment Page 60
Environment page 60
environment variables 60
adding 60
before starting poe 83
EDITOR 183
how to enter 60
LC_LIBRARY_PATH 39
LM_LICENSE_FILE 39
MP_ADAPTER_USE 83
MP_CPU_USE 83
MP_EUIDEVELOP 93
PATH 44, 50, 51
SHLIB_PATH 39
TOTALVIEW 78, 79, 118
TVDSVRLAUNCHCMD 68
equiv assembler pseudo op 329
error operators 228
error state 47
errors, in multiprocess program 50
ESECUTABLE_PATH variable 52
EVAL icon 124
for evaluation points 124
eval points
see evaluation points
Evaluate command 140, 144, 319, 320, 322
evaluating an expression in a watchpoint 314
evaluating expressions 319, 320
evaluating state 176
evaluation points 5, 306
assembler constructs 326
C constructs 324
clearing 124
defined 176, 294
defining 306
examples 308
Fortran constructs 325
hitting breakpoint while evaluating 321
listing 130

lists of 130
machine level 307
patching programs 6
printing from 5
saving 307
setting 124, 161, 308
using $stop 6
where generated 307
event log window 61
event points listing 130
examining
core files 45
data 7
process groups 190
processes 188
source and assembler code 131
stack trace and stack frame 244
status and control registers 203
exception enable modes 203
excluded information, reading 201
exclusion list, shared library 201
EXECUTABLE_PATH variable 42, 44, 50, 105, 158
setting 157
executables
debugging 37
loading 41
specifying name in scope 276
executing
out of function 196
startup file 39
to the completion of a function 196
execution
controlling 175
resuming 185
execution models 11
execve() 36, 44, 189, 302
attaching to processes 43
debugging programs that call 36
setting breakpoints with 302
existent operator 228
exit CLI command 40
Exit command 40
Exit command (Visualizer) 146
exited threads, displaying 126
expanding structures 240
expression evaluation window
compiled and interpreted expressions 310
discussion 319
Expression List window 10, 254
Add to Expression List command 255
aggregates 255
array of structures 255
diving 255
editing contents 258
editing the value 258
editing type field 258
entering variables 254
expressions 255
multiple windows 256
multiprocess/multithreaded behavior 256
rebinding 257
reevaluating 257
reopening 257
reordering rows 258
restarting your program 257
selecting before sending 255
sorting columns 258
expressions 228, 301
can contain loops 319
changing in Variable window 253
compiled 311
evaluating 319
in Expression List window 255
p/t 211
performance of 310
side effects 253
expressions and variables 253
extent of arrays 262

F

figures
Action Point > Properties Dialog Box 297, 300, 304
Action Point Symbol 294
Ambiguous Function Dialog Box 181, 297
Ambiguous Line Dialog Box 296
Array Data Filter by Range of Values 287
Array Data Filtering by Comparison 285
Array Data Filtering for IEEE Values 286
Array Statistics Window 289
Breakpoint at Assembler Instruction Dialog Box 299
Control and Share Groups Example 189
File > New Program Dialog Box 75
File > New Program Dialog Box Page 41
File > Preferences: Action Points Page 301
Five Processes and Their Groups on Two Computers 25
Fortran Array with Inverse Order and Limited Extent 282
Laminated Array and Structure 292
Laminated Scalar Variable 291
PC Arrow Over a Stop Icon 300
Root Widow: Group Page 190
Sorted Variable Window 288
Stopped Execution of Compiled Expressions 311
Stride Displaying the Four Corners of an Array 281
Tools > Evaluate Dialog Box 320, 321
Tools > Watchpoint Dialog Box 315
Two Computers Working on One Problem 17
Undive/Redive Buttons 250
Using Assembler 327
Variable Window for small_array 283
View > Display Exited Threads 126

Waiting to Complete Message Box 321
Zooming and Rotating About an Axis 153
file
for start up 39
File > Close command 134, 249
File > Close command (Visualizer) 146
File > Close Relatives command 134
File > Close Similar command 134, 249
File > Delete command (Visualizer) 146
File > Directory command (Visualizer) 146
File > Edit Source command 183
File > Exit command 40
File > Exit command (Visualizer) 146
File > New Base Window (Visualizer) 147
File > New Program command 37, 40, 42, 44, 46, 65, 68, 71, 75
File > New Program Dialog Box figure 41, 75
File > Options command (Visualizer) 147, 148
File > Preferences
Bulk Launch page 67
Options page 135
File > Preferences command
Action Points page 50, 55, 116
Bulk Launch page 56, 65, 67
Dynamic Libraries Page 200
Dynamic Libraries page 57
Fonts page 58
Formatting page 58, 238
Launch Strings page 56, 64, 140
Options page 49, 237
overview 54
Parallel page 57, 115
Pointer Dive page 59
File > Preferences command: Pointer Dive page 245
File > Preferences: Action Points Page figure 301
File > Save Pane command 136
File > Search Path command 42, 44, 50, 51, 52, 85, 105
search order 50, 51
File > Signals command 49
–file command-line option to Visualizer 141, 152
file extensions 36
files
.rhosts 84
hosts.equiv 84
fill assembler pseudo op 329
filter expression, matching 283
filtering 9
array data 283, 284
array expressions 287
by comparison 284
conversion rules 284
example 285
IEEE values 285
options 283
ranges of values 286
filters
$denorm 285
$inf 285
$nan 285
$nanq 285
$nans 285
$ninf 285
$pdenorm 285
$pinf 285
Find Again command 180
Find command 4, 180
finding
functions 181
source code 181, 182
source code for functions 181
first thread indicator of < 213
Five Processes and Their Groups on Two Computers figure 25
float assembler pseudo op 329
focus
as list 226
changing 212
pushing 212
restoring 212
for loop 319
Force window positions (disables window manager placement modes) check box 135
fork() 36, 189, 302
debugging programs that call 36
setting breakpoints with 302
fork_loop.tvd example program 168
Formatting page 238
Fortran
array bounds 262
arrays 262
common blocks 269
contained functions 271
data types, displaying 269
debugging modules 271
deferred shape array types 272
filter expression 287
in code fragment 306
in evaluation points 325
module data, displaying 269
modules 269, 271
pointer types 273
type strings supported by TotalView 260
user defined types 272
Fortran Array with Inverse Order and Limited Extent figure 282
Fortran casting for Global Arrays 101, 102
Fortran Modules command 270
forward icon 134
four linked processors 19
4142 default port 67
frame pointer 195
function visualization 137
functions
finding 181
locating 180
returning from 196

G

–g compiler option 35, 36, 133
g width specifier 218, 222
cast 101, 102
<Ga> cast 101
<ga> cast 101, 102
gcc UPC compiler 110
generating a symbol table 36
Global Arrays 101
casting 101, 102

diving on type information 101
Intel IA-64 101
global assembler pseudo op 329
global variables
changing 191
displaying 191
diving into 239, 243
Go command 4, 81, 84, 85, 87, 88, 116, 190
GOI defined 205
goto statements 307
Graph command (Visualizer) 146
Graph Data Window 147
graph markers 147
Graph visualization menu 145
graph window, creating 146
Graph, in Directory Window 146
graphs
manipulating, in Visualizer 149
two dimensional 147
group
process 209
thread 209
Group > Attach Subsets command 113
Group > Control > Go command 185
Group > Delete command 94, 117, 197
Group > Edit command 216
Group > Edit Group command 231
Group > Go command 85, 116, 190, 191, 302
Group > Halt command 117, 194
Group > Hold command 186
Group > Next command 117
Group > Release command 186
Group > Restart command 197
Group > Run To command 116
Group > Share > Halt command 184
Group > Step command 117
Group > Workers > Go commands 190
group aliases 173
limitations 173
group commands 116
group indicator
defined 217
group name 218
group number 218
group stepping 208
group syntax 217
group number 218
naming names 218
predefined groups 217
GROUP variable 222
group width specifier 214
groups 104
see also processes
and barriers 13
behavior 208
changing 231
creating 26, 190
defined 22, 23
deleting 231
examining 188
holding processes 186
listing 128
named 231
overview 22
process 209
relationships 215
releasing processes 186
running 115
setting 222
starting 190
stopping 115
thread 209
updating 231
Groups page 13, 128, 190
GUI namespace 172

H

h held indicator 185
–h localhost option for HP MPI 82
half assembler pseudo op 329
Halt command 117, 184, 194
halt commands 184
halting
groups 184
processes 184
threads 184
handler routine 48
handling signals 48, 49, 105, 106
held indicator 185
held operator 228
held processes, defined 303
hexadecimal address, specifying in variable window 246
hi16 assembler operator 328
hi32 assembler operator 328
hierarchy toggle 126
hierarchy toggle button Root Window 126
hold and release 185
$hold assembler pseudo op 328
$hold built-in function 323
Hold command 186
hold state 186
hold state, toggling 304
Hold Threads command 186
holding and advancing processes 175
holding threads 209
$holdprocess assembler pseudo op 328
$holdprocess built-in function 323
$holdprocessall built-in function 323
$holdprocessstopall assembler pseudo op 328
$holdstopall assembler pseudo op 328
$holdstopall built-in function 323
$holdthread assembler pseudo op 328
$holdthread built-in function 324
$holdthreadstop assembler pseudo op 328
$holdthreadstop built-in function 324
$holdthreadstopall assembler pseudo op 328
$holdthreadstopall built-in function 324
$holdthreadstopprocess assembler pseudo op 328
$holdthreadstopprocess built-in function 324
hostname
abbreviated in Root Window 125
for tvdsvr 38
in square brackets 125
hosts.equiv file 84
how TotalView determines share group 190

hung processes 42

I

I state 48
IBM MPI 83
IBM SP machine 78, 79
idle state 48
Ignore mode warning 50
ignoring action points 299
implicitly defined process/thread set 212
incomplete arena specifier 225
inconsistent widths 226
indicator 43
inf filter 285
infinite loop, *see* loop, infinite
infinity count array statistic 289
INFs 283
initial process 174
initialization search paths 38
initialization subdirectory 38
initializing an array slice 158
initializing debugging state 39
initializing the CLI 168
initializing TotalView 38
input files, setting 53
instructions
  data type for 266
  displaying 247, 248
int data type 261
int* data type 261
int[] data type 261
interactive CLI 165
interface to CLI 166
interpreted expressions 310
  performance 310
interrupting commands 167
intersection operator 228
intrinsics, *see* built-in functions
inverting array order 281
inverting axis 148
invoking CLI program from shell example 168
invoking TotalView on UPC 110
IP over the switch 83
iterating
  over a list 226
  over arenas 213

K

K state, unviewable 47
–KeepSendQueue command-line option 94
kernel 47
killing processes when exiting 43
killing programs 197
–ksq command-line option 94

L

L lockstep group specifier 218
labels, for machine instructions 248
LAM/MPI 86
  starting 86
Laminate > None command 291
Laminate > Process command 290
Laminate > Thread command 290
Laminate None command 291
Laminate Thread command. 99
Laminated Array and Structure figure 292
laminated data view 12
Laminated Scalar Variable figure 291
laminating Variable Window 292
lamination
  arrays and structures 291
  data panes and Visualizer 143
  diving in pane 292
  editing a pane 292
  variables 290, 291
launch
  configuring Visualizer 140
  options for Visualizer 140
  TotalView Visualizer from command line 152
  tvdsvr 63
Launch Strings Page 71
Launch Strings page 64, 140
lcomm assembler pseudo op 329
LD_LIBRARY_PATH environment variable 39, 110
left margin area 130
left mouse button 123
libraries
  dbfork 36
  debugging SHMEM library code 109
  naming 200
  see shared libraries
limiting array display 282
line number area 124
line numbers 130
  for specifying blocks 276
linear view 126
LINES_PER_SCREEN variable 171
linked lists, following pointers 249
list transformation, STL 237
lists of processes 124
lists of variables, seeing 10
lists with inconsistent widths 226
lists, iterating over 226
LM_LICENSE_FILE environment variable 39
lo16 assembler operator 328
lo32 assembler operator 328
Load All Symbols in Stack command 202
loader symbols, reading 200
loading
  core file 41
  file into TotalView 37
  new executables 40, 41
  programs 37
  remote executables 42
  shared library symbols 201
loading loader symbols 201
loading no symbols 201
local hosts 38
locations, toggling breakpoints at 295
lockstep group 25, 206, 213
  defined 23
  L specifier 218
  number of 216
  overview 216
Log page 61, 128
long variable names, displaying 241
$long_branch assembler pseudo op 328

Lookup Function command 107, 180, 182, 183
Lookup Variable command 99, 180, 239, 245, 246, 271
specifying slices 282
specifying slides 283
loop infinite, *see* infinite loop
lower adjacent array statistic 289
lower bounds 261
non default 262
of array slices 280
lysm TotalView pseudo op 329

M

M state 47
machine instructions
data type 266
data type for 266
displaying 247, 248
main() 107
stopping before entering 107
make_actions.tcl sample macro 161, 168
manager threads 20, 25
displaying 126
manual hold and release 185
manually starting tvdsvr 71
map templates 236
map transformation, STL 236
markers, in graphs 147
master process, recreating slave processes 117
master thread 95
OpenMP 97, 100
stack 98
matching processes 209
matching stack frames 291
maximum array statistic 290
mean array statistic 290
median array statistic 290
Memorize All command 135
Memorize command 135
memory
displaying areas of 246
memory locations, changing values of 259
menus, context 123
mesh, drawing as 151
message passing deadlocks 91
Message Passing Interface/Chameleon Standard, *see* MPICH
Message Passing Toolkit 91
Message Queue command 90, 91
message queue display 88, 94
Message Queue Graph 90
diving 90
rearranging shape 91
updating 90
Message Queue Graph command 90
message queue graph window 12
message states 90
message tags, reserved 108
message-passing programs 116
messages
envelope information 93
operations 92
reserved tags 108
unexpected 93
messages from TotalView, saving 171
middle mouse button 123
middle mouse dive 136
minimum array statistic 290
missing TID 214
mixed state 47
mixing arena specifiers 226
modify watchpoints, *see* watchpoints
modifying code behavior 307
module data definition 269
modules 269, 271
debugging Fortran 271
displaying Fortran data 269
monitoring TotalView sessions 61
more processing 171
more prompt 171
mouse button
diving 123
left 123
middle 123
right 123
selecting 123
mouse buttons, using 123
MP_ADAPTER_USE environment variable 83
MP_CPU_USE environment variable 83
MP_EUIDEVELOP environment variable 93
MP_TIMEOUT 84
mpcc_r compilers 91
MPI
attaching to 88
attaching to HP job 83
attaching to running job 82
buffer diving 93
communicators 91
library state 91
on HP Alpha 81
on HP machines 82
on IBM 83
on SGI 88
on Sun 89
process diving 92
processes, starting 87
starting on HP Alpha 81
starting on SGI 88
starting processes 81, 88
toolbar settings for 13
troubleshooting 94
MPI_Init() 85, 91
breakpoints and timeouts 118
MPI_Iprobe() 93
MPI_Recv() 93
MPICH 78, 79
and SIGINT 94
and the TOTALVIEW environment variable 78
attach from TotalView 80
attaching to 80
ch_lfshmem device 78, 80
ch_mpl device 78
ch_p4 device 78, 80
ch_shmem device 80
ch_smem device 78
configuring 78
debugging tips 118
diving into process 80
MPICH/ch_p4 118
mpirun command 78
naming processes 81
obtaining 78
P4 81
–p4pg files 81
starting TotalView using 78
using –debug 94
MPICH –tv command-line option 78

mpirun command 78, 83, 88, 118
examples 82
for HP MPI 82
options to TotalView through 118
passing options to 118
mpirun process 88
MPL_Init() 85
and breakpoints 85
mprun command 89
mpxlf_r compiler 91
mpxlf90_r compiler 91
MQD, *see* message queue display
multiple classes, resolving 182
Multiple indicator 291
multiple sessions 104
multiprocess debugging 10
multiprocess programming library 36
multiprocess programs
and signals 50
attaching to 44
compiling 35
process groups 188
setting and clearing breakpoints 300
multiprocessing 19
multi-threaded core files 46
multithreaded debugging 10
multithreaded signals 197

N

–n option, of rsh command 72
–n single process server launch command 69
named groups 128, 231
named sets 231
names of processes in process groups 189
namespaces 172
TV:: 172
TV::GUI:: 172
naming libraries 200
naming MPICH processes 81
naming rules
for control groups 189
for share groups 189
nan filter 285
nanq filter 285
NaNs 283, 285
array statistic 290
nans filter 285
navigating, source code 182
ndenorm filter 285
nested dive 133
defined 249
window 250
nested stack frame, running to 210
New Base Window
in Data Window 147
New Base Window command (Visualizer) 147
New Program command 37, 40, 42, 44, 46, 68, 71, 75
Next command 117, 191
"next" commands 195
Next Instruction command 192
$nid built-in variable 322
ninf filter 285
–no_stop_all command-line option 118
node ID 322
nodes, attaching from to poe 85
nodes, detaching 114
None (laminate) command 291
nonexistent operators 228
non-sequential program execution 166

O

–O option 36
offsets, for machine instructions 248
$oldval built-in variable 322
omitting array stride 280
omitting components in creating scope 277
omitting period in specifier 226
omitting width specifier 225, 226
<opaque> data type 265
opaque type definitions 265
Open process window at breakpoint check box 50
Open process window on signal check box 49
opening shared libraries 198
OpenMP 95, 96
debugging 95
debugging applications 95
master thread 95, 97, 98, 100
master thread stack context 98
on HP Alpha 97
private variables 97
runtime library 95
shared variables 97, 100
stack parent token 100
THREADPRIVATE common blocks 98
THREADPRIVATE variables 99
threads 97
TotalView-supported features 95
viewing shared variables 98
worker threads 95
operators
- difference 228
& intersection 228
| union 228
breakpoint 228
error 228
existent 228
held 228
nonexistent 228
running 228
stopped 228
unheld 228
watchpoint 228
optimizations, compiling for 36
options
for visualize 152
in Data Window 147
–patch_area 312
–patch_area_length 312
–sb 319
–serial 75
setting 59
surface data display 151
Options > Auto Visualize command (Visualizer) 146
Options command (Visualizer) 147, 148
Options page 135, 237
org assembler pseudo op 329
ORNL PVM, *see* PVM
"out" commands 196
out command, goal 196
outliers 289, 290
outlined routine 95, 99, 100

outlining, defined 95
output
assigning output to variable 170
from CLI 170
only last command executed returned 170
printing 170
returning 170
when not displayed 170
output files, setting 53

P

p width specifier 218
p.t notation 213
p/t expressions 211
p/t set browser 229
P/T Set Browser command 229
p/t sets
arguments to Tcl 212
arranged hierarchically 229
browser 229
defined 211
expressions 228
grouping 228
set of arenas 213
syntax 214
visualizing 229
p/t syntax, group syntax 217
p4 listener process 80
–p4pg files 81
–p4pg option 81
panes
action points list, *see* action points list pane
source code, *see* source code pane
stack frame, *see* stack frame pane
stack trace, *see* stack trace pane
panes, saving 136
parallel debugging tips 113
PARALLEL DO outlined routine 97
Parallel Environment for AIX, *see* PE
parallel environments, execution control of 175
Parallel Page 115
Parallel page 115
parallel program, defined 174
parallel program, restarting 117
parallel region 96
parallel tasks, starting 85
Parallel Virtual Machine, *see* PVM
parallel_attach variable 116
parallel_stop variables 115
parsing comments example 161
passing arguments 37
passing default arguments 172
passing environment variables to processes 60
Paste command 136
pasting 136
pasting between windows 136
pasting with middle mouse 123
patch space size, different than 1MB 312
patch space, allocating 311
–patch_area_base option 312
–patch_area_length option 312
patching
function calls 309
programs 309
PATH environment variable 42, 44, 50, 51
pathnames, setting in procgroup file 81
PC Arrow Over a Stop Icon figure 300
PC icon 202
pdenorm filter 285
PE 85, 91
adapter_use option 83
and slow processes 118
applications 83
cpu_use option 83
debugging tips 118
from command line 84
from poe 84
options to use 83
switch-based communication 83
PE applications 83
pending messages 91
pending receive operations 92, 93
pending send operations 92, 94
configuring for 94
pending unexpected messages 92
performance of interpreted, and compiled expressions 310
performance of remote debugging 63
–persist command-line option to Visualizer 141, 152
phase, UPC 113
$pid built-in variable 322
pid specifier, omitting 225
pid.tid to identify thread 130
pinf filter 285
pipe for Visualizer 140
piping data 136
Plant in share group check box 302, 308
poe
and mpirun 79
and TotalView 84
arguments 83
attaching to 85, 86
interacting with 118
on IBM SP 80
placing on process list 86
required options to 83
running PE 84
TotalView acquires poe processes 85
poe, and bulk server launch 71
POI defined 205
point of execution for multiprocess or multithreaded program 130
pointer data 133
Pointer Dive page 245
pointers 133
as arrays 245
chasing 245, 249
dereferencing 245
diving on 133
in Fortran 273
to arrays 261
pointer-to-shared UPC data 112
pop_at_breakpoint variable 50
pop_on_error variable 49
popping a window 133
port 4142 67
–port command-line option 67
port number for tvdsvr 38
precision 238

changing 238
changing display 58
predefined data types 263
preference file 39
Preferences
Action Points page 55
Bulk Launch page 56, 65, 67
Dynamic Libraries page 57
Fonts page 58
Formatting page 58
Launch Strings page 56, 64
Options page 49
Parallel page 57
Pointer Dive page 59
preferences, setting 59
preferences6.tvd file 39
preferences6.tvd startup file 39
preloading shared libraries 198
primary thread, stepping failure 209
print statements, using 4
printing an array slice 159
printing in an eval point 5
private variables 95
in OpenMP 97
procedures
debugging over a serial line 74
displaying 266
displaying declared and allocated arrays 266
process
detaching 45
holding 209
state 46
synchronization 209
Process > Create command 191
Process > Detach command 45
Process > Go command 81, 82, 84, 87, 88, 116, 190, 191, 197
Process > Halt command 117, 184, 194
Process > Hold command 186
Process > Hold Threads command 186
Process > Next command 191
Process > Next Instruction command 192
Process > Out command 207
Process > Release Threads command 186
Process > Run To command 192, 207
Process > Startup command 37, 53
Process > Startup Parameters 53
Arguments page 53
Environment Page 60
Environment page 60
Standard I/O page 54
Process > Startup Parameters command 53, 54
Process > Step command 191
Process > Step Instruction command 191
process as dimension in Visualizer 143
process barrier breakpoint
changes when clearing 306
changes when setting 306
defined 293
deleting 306
setting 304
process DUID 322
process groups 23, 209, 216
behavior 221
behavior at goal 209
displaying 190
stepping 208
synchronizing 209
process ID 322
process numbers are unique 174
process states 47, 130
process states, attached 47
process stepping 208
process synchronization 116
process width specifier 214
omitting 226
Process Window 4, 129
displaying 133
host name in title 125
raising 49
process/set threads
saving 215
process/thread identifier 174
process/thread notation 174
process/thread sets 174
as arguments 212
changing focus 212
default 212
implicitly defined 212
inconsistent widths 226
structure of 214
target 212
widths inconsistent 226
process_id.thread_id 213
process_load_callbacks variable 40
$processduid built-in variable 322
processes
see also automatic process acquisition
see also groups
acquiring 79, 80, 106
acquiring in PVM applications 104
acquisition in poe 85
apparently hung 117
attaching 42, 43, 127
attaching to 42, 43, 85, 107
barrier point behavior 306
behavior 208
breakpoints shared 301
call tree 138
cleanup 108
copy breakpoints from master process 79
creating 53, 190, 191
creating by single-stepping 191
creating new 169
creating using Go 191
creating without starting 191
deleting 197
deleting related 197
detaching from 45
displaying data 133
diving into 43, 85
diving on 133
groups 188
examining 190
held defined 303
holding 185, 303, 323
hung 42
initial 174
killing while exiting 43
list of 124
loading new executables 40
local 43

master restart 117
MPI 92
names 189
passing environment variables to 60
refreshing process info 185
released 304
releasing 185, 303, 306
remote 43
restarting 197
single-stepping 207
slave, breakpoints in 79
spawned 174
starting 191
state 46
status of 46
stepping 13, 117, 208
stop all related 300
stopped 304
stopped at barrier point 306
stopping 184, 307
stopping all related 49
stopping and deleting 308
stopping intrinsic 324
stopping spawned 79
switching between 11
synchronizing 176, 209
terminating 169
types of process groups 188
when stopped 208
process-level stepping 117
processors and threads 20
procgroup file 81
using same absolute path names 81
Program Browser 243
explaining symbols 243
program control groups
defined 216
naming 189
program counter (PC) 43, 130
arrow icon for PC 130
indicator 130
setting 202
setting program counter 202
setting to a stopped thread 202
program execution
advancing 175
controlling 175
program state, changing 166
program visualization 137
programming languages
determining which used 36
programming TotalView 13
programs
compiling 3, 35
compiling using –g 35
correcting 310
deleting 197
killing 197
loading by process ID 41
not compiled with –g 36
patching 6, 309
restarting 197
prompt and width specifier 220
PROMPT variable 173
Properties command 118, 294, 297, 300, 304, 308
properties, of action points 5
prototypes for temp files 66
prun command 87
prun, and bulk server launch 70
pthread ID 174
pthreads, *see* threads
pushing focus 212
PVM
acquiring processes 104
attaching procedure 107
attaching to tasks 107
automatic process acquisition 106
cleanup of tvdsvr 108
creating symbolic link to tvdsvr 104
daemons 108
debugging 103
message tags 108
multiple instances not allowed by single user 104
multiple sessions 104
running with DPVM 104
same architecture 107
search path 105
starting actions 106
tasker 106
tasker event 107
tasks 104
TotalView as tasker 104
TotalView limitations 104
tvdsvr 106
Update Command 107
pvm command 104, 105
PVM groups, unrelated to process groups 104
PVM Tasks command 107
pvm variable 106
pvm_joingroup() 108
pvm_spawn() 104, 106, 107
pvmgs process 104, 108
terminated 108

Q

QSW RMS applications 87
attaching to 87
debugging 87
starting 87
quad assembler pseudo op 329
Quadrics RMS 87
quartiles array statistic 290
queue state 90

R

R state 47, 48
raising process window 49
rank for Visualizer 140
ranks 90
read_symbols command 202
reading loader and debugger symbols 200
rebinding the Variable Window 248
recursive functions 196
single-stepping 195
redirecting
stdin 53
stdout 53
redive 250
redive all 250
redive buttons 250
redive icon 134, 250
redive/undive buttons 8
registers
editing 203
interpreting 203
relatives, attaching to 44
Release command 186
release state 186
Release Threads command 186
reloading breakpoints 85
remembering window positions 135
–remote command-line option 38, 65
Remote Debug Server Launch preferences 64
remote debugging 63

see also PVM applications
launching tvdsvr 63
performance 63
remote executables, loading 42
remote host, debugging on 41
remote hosts 38
remote login 84
–remote option 38, 42
remote shell command, changing 71
removing breakpoints 124
remsh command 71
used in server launches 68
replacing default arguments 172
researching directories 52
reserved message tags 108
Reset command 182, 183
Reset command (Visualizer) 151
resetting command-line arguments 53
resetting the program counter 202
resolving ambiguous names 182
resolving multiple classes 182
resolving multiple static functions 182
Restart Checkpoint command 198
Restart command 197
restarting
parallel programs 117
program execution 169
programs 197
restoring focus 212
results, assigning output to variables 170
resuming
executing thread 202
execution 185, 191
processes with a signal 196
returning to original source location 181
reusing windows 133
.rhosts file 72
right angle bracket (>) 133
right arrow is program counter 43
right mouse button 123
RMS applications 87
attaching to 87
starting 87
Root Window 11, 124, 126
Attached Page 86, 124, 126
collapsing entries 126
expanding entries 126
Groups page 13, 128, 190
Log page 61, 128
selecting a process 133
sorting columns 126
starting CLI from 167
state indicator 46
Unattached page 11, 42, 43, 46, 48, 80, 85, 127
Root Window: Group Page figure 190
rotating surface 152
rounding modes 203
routine visualization 137
routines, diving on 133
routines, selecting 130
rsh command 71, 84
rules for scoping 276
Run To command 4, 116
"run to" commands 195, 209
running CLI commands 39
running groups 115
running operator 228
running state 47

S

–s command-line option 39, 168
S share group specifier 217
S state 48
S width specifier 218
sample programs
make_actions.tcl 168
sane command argument 167
Satisfaction group items pulldown 305
satisfaction set 305
satisfied barrier 305
Save All (action points) command 319
Save All command 319
Save Pane command 136
saved action points 39
saving
action points 319
TotalView messages 171
window contents 136
–sb option 319
scaling a surface 152
scaling data window 149
scope
determining 242
scope pulldown 211
scopes, compiled in 275
scoping 274
ambiguous 277
as a tree 276
omitting components 277
rules 276
scoping issues 242
scrolling 123
output 171
undoing 183
Search Path command 42, 44, 50, 51, 52, 85
search order 50, 51
search paths
default lookup order 51
for initialization 38
order 51
setting 50, 105
–search_port command-line option 67
searching 180
case-sensitive 180
for source code 182
functions 181
locating closest match 181
source code 181
wrapping to front or back 180
Searching, *see* Edit > Find, View > Lookup Function, View > Lookup Variable
searching, variable not found 181
seeing structures 240
select button 123
selected line, running to 210
selecting
different stack frame 130
routines 130
source code, by line 202
source line 192
text 135
selection and Expression List window 255
sending signals to program 50
–serial command-line option 74
serial line
baud rate 74
debugging over a 74
radio button 75

starting TotalView 75
–serial option 75
server launch 64
command 65
enabling 64
replacement character %C 68
server on each processor 17
–server option 67
server_launch_enabled variable 64, 67, 71
server_launch_string variable 65
server_launch_timeout variable 65
service threads 21, 25
Set Barrier command 304
set expressions 228
set indicator, uses dot 214, 229
Set PC command 203
Set Signal Handling Mode command 105, 106
–set_pw command-line option 71
–set_pw single process server launch command 69
–set_pws bulk server launch command 70
setting
barrier breakpoint 304
breakpoints 84, 124, 161, 206, 295, 300
breakpoints while running 295
command arguments 53
command line arguments 53
environment variables 60
evaluation points 124, 308
groups 222
input and output files 53
options 59
preferences 59
search paths 50, 105
thread specific breakpoints 322
setting timeouts 84
setting up, debug session 35
setting up, parallel debug session 77
setting up, remote debug session 63
setting X resources 59
SGI, and bulk server launch 69
SGROUP variable 222
shading graph 151
shape arrays, deferred types 272
Share > Halt command 184
share groups 24, 188, 216
defined 22
determining 190
determining members of 190
discussion 189
naming 189
overview 216
S specifier 217
SHARE_ACTION_POINT variable 298, 301, 302
shared libraries 198
controlling which symbols are read 200
loading all symbols 201
loading loader symbols 201
loading no symbols 201
preloading 198
reading excluded information 201
shared library, exclusion list order 201
shared library, specifying name in scope 276
shared memory library code, *see* SHMEM library code debugging
shared variables 95
in OpenMP 97
OpenMP 97, 100
procedure for displaying 97
sharing action points 302
shell, example of invoking CLI program 168
SHLIB_PATH environment variable 39
SHMEM library code debugging 109
showing areas of memory 246
SIGALRM 118
SIGFPE errors (on SGI) 49
SIGINT signal 94
signal handling mode 49
signal/resignal loop 50
signal_handling_mode variable 49
signals
affected by hardware registers 48
clearing 197
continuing execution with 196
discarding 50
error option 50
handler routine 48
handling 48
handling in PVM applications 105, 106
handling in TotalView 48
handling mode 49
ignore option 50
resend option 50
sending continuation signal 196
SIGALRM 118
SIGTERM 105, 106
stop option 50
stops all related processes 49
that caused core dump 46
Signals command 49
SIGSTOP
used by TotalView 48
when detaching 45
SIGTERM signal 105, 106
stops process 105
terminates threads on SGI 97
SIGTRAP, used by TotalView 48
single process server launch 63, 64, 68
single process server launch command
%D 69
%L 69
%P 69
%R 68
%verbosity 69
–callback_option 69
–n 69
–set_pw 69
–working_directory 69
single-stepping 193, 207
commands 193
in a nested stack frame 210
into function calls 194
not allowed for a parallel region 96
on primary thread only 207

operating system dependencies 195, 197
over function calls 195
recursive functions 195
skipping elements 281
slash in group specifier 218
sleeping state 48
slices 8, 282
defining 280
descriptions 282
displaying one element 282
examples 280, 281
lower bound 280
of arrays 279
operations using 273
stride elements 280
UPC 111
upper bound 280
with the variable command 282
smart stepping, defined 207
SMP machines 78
sockets 74
Sorted Variable Window figure 288
sorting
array data 288
Root Window columns 126
Source As > Assembler 131
Source As > Both 131, 202
Source As > Both command 202
Source As > Source 131
source code
examining 131
finding 181, 182
navigating 182
Source command 131
source file, specifying name in scope 277
source lines
ambiguous 192
editing 183
searching 192
selecting 192
Source Pane 129, 130
source-level breakpoints 295
space allocation
dynamic 311
static 311, 312
spawned processes 174
stopping 79
specifier combinations 218
specifiers
and dfocus 219
and prompt changes 220
example 223
examples 219, 220
specifying groups 217
specifying search directories 52
splitting up work 17
stack
master thread 98
trace, examining 244
unwinding 203
stack context of the OpenMP master thread 98
stack frame 240
current 182
examining 244
matching 291
pane 130
selecting different 130
Stack Frame Pane 7, 130, 248
stack parent token 100
diving 100
Stack Trace Pane 130, 202
displaying source 133
standard deviation array statistic 290
Standard I/O Page 54
standard input, and launching tvdsvr 72
Standard Template Library 235
standard template library, *see* STL
start(), stopping within 107
start_pes() SHMEM command 109, 110
starting 101
CLI 37, 167
groups 190
parallel tasks 85
TotalView 4, 36, 37, 45, 84
tvdsvr 38, 63, 67, 106
tvdsvr manually 71
starting LAM/MPI programs 86
starting program under CLI control 168
Startup command 37
startup file 39
Startup Parameters command 53, 54
Arguments page 53
Environment page 60
Standard I/O page 54
state characters 48
states
and status 46
initializing 39
of processes and threads 46
process and thread 47
unattached process 48
static constructor code 191
static functions, resolving multiple 182
static patch space allocation 311, 312
statically linked, stopping in start() 107
statistics for arrays 289
status
and state 46
of processes 46
of threads 46
status registers
examining 203
interpreting 203
stdin, redirect to file 53
stdout, redirect to file 53
Step command 4, 117, 191
"step" commands 194
Step Instruction command 191
stepping
see also single-stepping
apparently hung 117
at process width 208
at thread width 208
goals 208
into 194
multiple statements on a line 194
over 195
primary thread can fail 209
process group 208
processes 117
Run (to selection) Group command 116
smart 207
target program 175
thread group 208
threads 227
using a numeric argument in CLI 194
workers 227
stepping a group 208
stepping a process 208
stepping commands 191
stepping processes and threads 13
STL 235
list transformation 237

map transformation 236
platforms supported 236
STL preference 237
STLView 235
$stop assembler pseudo op 328
$stop built-in function 324
Stop control group on error check box 50
Stop control group on error signal option 49
stop execution 4
STOP icon 124, 206, 295, 299
for breakpoints 124, 295
stop, defined in a multiprocess environment 175
STOP_ALL variable 298, 300
$stopall built-in function 324
Stopped Execution of Compiled Expressions figure 311
stopped operator 228
stopped process 306
stopped state 47
unattached process 48
stopping
all related processes 49
groups 115
processes 184, 308
spawned processes 79
threads 184
$stopprocess assembler pseudo op 328
$stopprocess built-in function 324
$stopthread built-in function 324
stride 280
default value of 280
elements 280
in array slices 280
omitting 280
Stride Displaying the Four Corners of an Array figure 281
string assembler pseudo op 329
<string> data type 265
structs
see also structures
defined using typedefs 263
how displayed 262
structure information 240
structures 249, 262
see also structs
collapsing 240
editing types 260
expanding 240
laminating 291
stty sane command 167
subroutines, displaying 133
suffixes of processes in process groups 189
suffixes variables 36
sum array statistic 290
Sun MPI 89
Suppress All command 299
suppressing action points 299
surface
coloring 151
display 151
in directory window 146
rotating 152
scaling 152
translating 152
zooming 152
Surface command (Visualizer) 146
Surface Data Window 149
display 151
Surface visualization window 145
surface window, creating 146
suspended windows 320
switch-based communication
for PE 83
switch-based communications 83
symbol lookup 275
and context 275
symbol name representation 274
symbol reading, deferring 200
symbol scoping, defined 276
symbol specification, omitting components 277
symbol table debugging information 35
symbolic addresses, displaying assembler as 131
Symbolically command 131, 132
symbols
loading all 201
loading loader 201
not loading 201
synchronizing execution 185
synchronizing processes 176, 209
syntax 217
system PID 174
system TID 174
system variables, *see* CLI variables
systid 130, 174
$systid built-in variable 322

T

T state 47, 48
t width specifier 218
tag field 299
tag field area 130
target process/thread set 175, 212
target program
stepping 175
target, changing 212
tasker event 107
tasks
attaching to 107
diving into 107
PVM 104
starting 85
Tcl
and CLI 165
and the CLI 13
CLI and thread lists 166
version based upon 165
Tcl and CLI relationship 166
TCP/IP address, used when starting 38
TCP/IP sockets 74
temp file prototypes 66
templates
lists 236
maps 236
STL 235
vectors 236
terminating processes 169
testing when a value changes 314
text
editing 135
locating closest match 181
saving window contents 136
selecting 135
text assembler pseudo op 329
text editor, default 183

third party visualizer 139
Thread > Continuation Signal command 45, 196
Thread > Go command 191
Thread > Hold command 186
Thread > Set PC command 203
thread as dimension in Visualizer 143
thread group 209
  stepping 208
thread groups 23, 209, 216
  behavior 221
  behavior at goal 209
thread ID 130, 174
  system 322
  TotalView 322
thread local storage 98
  variables stored in different locations 98
thread numbers are unique 174
Thread Objects command 274
thread objects, displaying 274
Thread of Interest 190
thread of interest 213, 215
  defined 184, 213
Thread Pane 130
thread selector 211
thread state 47
thread stepping 227
  platforms where allowed 209
thread width 208
thread width specifier 214
  omitting 226
THREADPRIVATE common block, procedure for viewing variables in 99
THREADPRIVATE variables 99
threads
  call tree 138
  changing in Expression List window 257
  changing in Variable window 248
  creating 18
  displaying manager 126
  displaying source 133
  diving on 130, 133
  finding window for 130
  holding 185, 209, 304
  ID format 130
  listing 130
  manager 20
  not available on all systems 23
  opening window for 130
  releasing 185, 303, 305
  resuming executing 202
  service 21
  setting breakpoints in 322
  single-stepping 207
  stack trace 130
  state 46
  status of 46
  stepping 13
  stopping 184
  switching between 11
  systid 130
  tid 130
  user 20
  workers 20, 22
threads model 18
thread-specific breakpoints 322
tid 130, 174
$tid built-in variable 322
TID missing in arena 214
timeouts
  avoid unwanted 118
  during initialization 85
  for connection 65
  TotalView setting 84
timeouts, setting 84
TOI defined 184
  again 205
Tool > P/T Set Browser command 229
toolbar
  controls 211
  using 184, 211
  width controls 211
Tools > Call Tree command 137
Tools > Command Line command 37, 167
Tools > Create Checkpoint command 198
Tools > Evaluate command 140, 144, 199, 253, 319, 320, 322
Tools > Evaluate command, *see* Expression List window
Tools > Evaluate Dialog Box figure 320, 321
Tools > Fortran Modules command 270
Tools > Global Arrays command 101
Tools > Laminate command 113
Tools > Manage Shared Libraries command 199
Tools > Message Queue command 90, 91
Tools > Message Queue Graph command 12, 90
Tools > Program Browser command 239
Tools > PVM Tasks command 107
Tools > Restart Checkpoint command 198
Tools > Statistics command 289
Tools > Thread Objects command 274
Tools > Variable Browser command 243
Tools > Visualize command 9, 143, 292
Tools > Visualize Distribution command 112
Tools > Watchpoint command 10, 315, 317
Tools > Watchpoint Dialog Box figure 315
TotalView
  and MPICH 78
  as PVM tasker 104
  core files 37
  initializing 38
  interactions with Visualizer 140
  invoking on UPC 110
  programming 13
  relationship to CLI 166
  starting 4, 36, 37, 45, 84
  starting on remote hosts 38
  starting the CLI within 167
  Visualizer configuration 140
TotalView assembler operators
  hi16 328
  hi32 328
  lo16 328
  lo32 328
TotalView assembler pseudo ops

$debug 328
$hold 328
$holdprocess 328
$holdprocessstopall 328
$holdstopall 328
$holdthread 328
$holdthreadstop 328
$holdthreadstopall 328
$holdthreadstopprocess 328
$long_branch 328
$stop 328
$stopall 328
$stopprocess 328
$stopthread 328
align 328
ascii 328
asciz 328
bss 329
byte 329
comm 329
data 329
def 329
double 329
equiv 329
fill 329
float 329
global 329
half 329
lcomm 329
lysm 329
org 329
quad 329
string 329
text 329
word 329
zero 329
totalview command 37, 39, 45, 81, 84, 88
for HP MPI 82
starting on a serial line 75
TotalView data types
<address> 263
<char> 263
<character> 263
<code> 264, 266
<complex*16> 264
<complex*8> 264
<complex> 264
<double precision> 264
<double> 264
<extended> 264
<float> 264
<int> 264
<integer*1> 264
<integer*2> 264
<integer*4> 264
<integer*8> 264
<integer> 264
<logical*1> 264
<logical*2> 264
<logical*4> 264
<logical*8> 264
<logical> 264
<long long> 264
<long> 264
<opaque> 265
<real* 16> 265
<real* 4> 265
<real* 8> 265
<real> 265
<short> 265
<string> 265
<void> 265
TotalView Debugger Server, *see* tvdsvr
TOTALVIEW environment variable 78, 79, 118
totalview subdirectory, *see* .totalview subdirectory
TotalView Visualizer *see* Visualizer
TotalView windows
action point List pane 130
editing cursor 135
totalviewcli command 37, 38, 39, 45, 88, 167, 169
–remote 38
starting on a serial line 75
transformations, creating 237
translating a surface 152
translating data window 149
transposing axis 148
TRAP_FPE environment variable on SGI 49
troubleshooting xxiv
MPI 94
ttf variable 237
–tv command-line option 78
TV:: namespace 172
TV::GUI:: namespace 172
TVD.breakpoints file 319
TVDB_patch_base_address object 312
tvdb_patch_space.s 313
tvdrc file, *see* .tvdrc initialization file
tvdsvr 38, 42, 63, 64, 65, 73, 74, 310
attaching to 107
–callback command-line option 71
cleanup by PVM 108
editing command line for poe 85
fails in MPI environment 94
launch problems 65, 67
launching 68
launching, arguments 72
manually starting 71
–port command-line option 67
–search_port command-line option 67
–server command-line option 67
–set_pw command-line option 71
starting 67
starting for serial line 74
starting manually 67, 71
symbolic link from PVM directory 104
with PVM 106
tvdsvr command 67
starting 63
timeout while launching 65, 66
use with PVM applications 104
TVDSVRLAUNCHCMD environment variable 68
Two Computers Working on One Problem figure 17
two-dimensional graphs 147
type casting 260
examples 266
type strings
built-in 263
editing 260
for opaque types 265
supported for Fortran 260
type transformation variable 237
type transformations, creating 237
typedefs
defining structs 263
how displayed 262
types supported for C language 260

types, user defined type 272

U

UDT 272
UDWP, *see* watchpoints
UID, UNIX 68
Unattached page 11, 42, 43, 46, 48, 80, 85, 127
unattached process states 48
undive 250
undive all 250
undive buttons 250
undive icon 134, 181, 250
undive/redive buttons 8
Undive/Redive Buttons figure 250
undiving, from windows 250
unexpected messages 91, 93
unheld operator 228
union operator 228
unions 262
  how displayed 263
unique process numbers 174
unique thread numbers 174
unsuppressing action points 299
unwinding the stack 203
UPC
  assistant library 110
  compilers supported 110
  phase 113
  pointer-to-shared data 112
  shared scalar variables 110
  slicing 111
  starting 110
  viewing shared objects 110
UPC debugging 110
Update command 86, 185
updating groups 231
updating visualization displays 143
upper adjacent array statistic 290
upper bounds 261
  of array slices 280
USEd information 271
user defined data type 272
user mode 20
user threads 20
Using Assembler figure 327
Using the Attached Page 126

V

value field 320
values
  changing 135
  editing 7
Variable Browser command 243
variable scoping 274
Variable Window
  closing 249
  displaying 238
  duplicating 251
  in recursion, manually refocus 240
  laminated display 290
  stale in pane header 239
  tracking addresses 239
  updates to 239
Variable window 253
  changing threads 248
  laminated 292
  rebinding 248
Variable Window for small_array figure 283
variables
  assigning p/t set to 215
  at different addresses 291
  CGROUP 216, 222
  changing the value 259
  changing values of 259
  display width 238
  displaying all globals 243
  displaying contents 133
  displaying long names 241
  displaying STL 235
  diving 133
  GROUP 222
  in modules 270
  in Stack Frame Pane 7
  intrinsic, *see* built-in functions
  laminated display 290
  locating 180
  precision 238
  previewing size and precision 238
  setting command output to 170
  SGROUP 222
  stored in different locations 98
  ttf 237
  watching for value changes 10
  WGROUP 222
variables and expressions 253
variables, viewing as list 254
–verbosity bulk server launch command 70
verbosity level 88
–verbosity single process server launch command 69
View > Add to Expression List command 255
View > Assembler > By Address command 131
View > Assembler > Symbolically command 131, 132
View > Collapse All command 240
View > Display Exited Threads figure 126
View > Display Manager Threads command 126
View > Dive command 258
View > Dive In All command 251, 252
View > Dive in New Window command 8
View > Dive Thread command 274
View > Dive Thread New command 274
View > Expand All command 240
View > Graph command 145
View > Graph command (Visualizer) 146
View > Laminate > None command 291
View > Laminate > Process command 290
View > Laminate > Thread command 290
View > Laminate Thread command 99
View > Lookup Function command 107, 180, 182, 183
View > Lookup Variable command 99, 180, 239, 245, 246, 271
  specifying slices 282, 283

View > Reset command 182, 183
View > Reset command (Visualizer) 149, 151
View > Source As > Assembler command 131
View > Source As > Both command 131, 202
View > Source As > Source command 131
View > Surface command (Visualizer) 145, 146
View > Variable command 98
View simplified STL containers preference 237
viewing assembler 131
viewing existed threads 126
Viewing manager threads 126
viewing shared UPC objects 110
viewing templates 235
viewing variables in lists 254
visualization
  deleting a dataset 145
  translating a surface 152
  zooming a surface 152
$visualize 324
Visualize command 9, 141, 143, 292
visualize command 152
visualize function 144
visualize, *see* $visualize
Visualizer 145, 292
  autolaunch options, changing 140
  choosing method for displaying data 142
  configuring 140
  configuring launch 140
  creating graph window 146
  creating surface window 146
  data sets to visualize 142
  data types 142
  data window 145, 146
  data window manipulation commands 149
  dataset defined 142
  dataset numeric identifier 142
  dataset parameters 152
  deleting datasets 146
  dimensions 143
  directory window 145
  display not automatically updated 143
  exiting from 146
  –file command-line option 141, 152
  graphs, display 147, 148
  graphs, manipulating 149
  interactions with TotalView 140
  laminated data panes 143
  launch command, changing shell 141
  launch from command line 152
  launch options 140
  method 142
  new or existing dataset 142
  number of arrays 142
  –persist command-line option 141, 152
  pipe 140
  rank 140
  relationship to TotalView 139
  rotating 152
  scaling a surface 152
  shell launch command 141
  slices 142
  surface data display options 151
  Surface Data Window 149
  third party 139
  using casts 144
  windows, types of 145
visualizer
  closing connection to 141
  customized command for 140
visualizing
  data 139, 145
  data sets from a file 152
  from variable window 143
  in expressions using $visualize 143
visualizing a dataset 144
<void> data type 265

W

W state 47
W width specifier 218
W workers group specifiers 218
Waiting for Command to Complete window 117
Waiting to Complete Message Box figure 321
warn_step_throw variable 49
watching memory 316
Watchpoint command 10, 315, 317
watchpoint operator 228
Watchpoint Properties dialog box 315
watchpoint state 47
watchpoints 10, 313
  $newval 317
  $oldval 317
  alignment 318
  conditional 314, 317
  copying data 317
  creating 315
  defined 176, 294
  disabling 315
  diving into 315
  enabling 315
  evaluated, not compiled 318
  evaluating an expression 314
  example of triggering when value goes negative 318
  length compared to $oldval or $newval 318
  lists of 130
  lowest address triggered 316
  modifying a memory location 313
  monitoring adjacent locations 316
  multiple 316
  not saved 319
  PC position 316
  platform differences 314
  problem with stack variables 316
  supported platforms 314
  testing a threshold 314

testing when a value changes 314
triggering 313, 316
watching memory 316
WGROUP variable 222
When a job goes parallel or calls exec() radio buttons 115
When a job goes parallel radio buttons 116
When Done, Stop radio buttons 305
When Hit, Stop radio buttons 305
width pulldown 211
width relationships 215
width specifier 213
omitting 225, 226
wildcards, when naming shared libraries 201
Window > Duplicate command 134, 251
Window > Memorize All command 135
Window > Memorize command 135
Window > Update command 86, 185
window contents, saving 136
windows 249
closing 134, 249
copying between 136
data 146
Data Window (Visualizer) 147
Directory Window 145
event log 61
graph data 147
pasting between 136
popping 133
resizing 134
Surface Data Window 149
suspended 320
Windows > Update command (PVM) 107
word assembler pseudo op 329
worker threads 20, 95
workers group 25, 210
defined 22
overview 216
workers group specifier 218
working directory 52
working independently 16
–working_directory bulk server launch command 69
–working_directory single process server launch command 69

X

X resources file 39
X resources setting 59
Xdefaults file, *see* .Xdefaults file
xterm, launching tvdsvr from 72

Z

Z state 48
zero assembler pseudo op 329
zero count array statistic 290
zombie state 48
zone coloring 151
zone maps 149
zooming a surface 152
Zooming and Rotating About an Axis figure 153
zooming data window 149