Developer Magic™: Debugger User's Guide

Document Number 007–2579–004

New Features

Developer Magic™: Debugger User's Guide 007–2579–004

This revision of the *Developer Magic: Debugger User's Guide* supports the 2.7 release of ProDev WorkShop tools.

Record of Revision

Version	*Description*
1.0	1991 Original Printing.
2.7	June 1998 Revised to reflect changes for the ProDev WorkShop 2.7 release.

Contents

Page

About This Guide **xix**

Related Publications xix

Obtaining Publications xx

Conventions xx

Reader Comments xxi

Getting Started with the WorkShop Debugger [1] **1**

Typical Debugger Usage 1

Starting and Exiting the Debugger 1

Using Main View 2

Setting Traps 3

Inspecting Debugger Data 4

Changing Source Code 4

Integration With Other WorkShop Tools 5

Accessing the Performance Analyzer from Main View 5

Accessing the Static Analyzer from Main View 5

Accessing Editors from Main View 5

Accessing Configuration Management Tools 5

Recompiling from Main View 6

Debugging with Fix and Continue 6

Redefining Functions Using Fix and Continue 6

Fix and Continue Functionality 6

Fix and Continue / WorkShop Integration 8

How Redefined Code Is Distinguished from Compiled Code 8

Page

Restrictions on Fix and Continue 9
The Fix and Continue Environment 10
Debugger With Fix and Continue Support 10
Change ID, Build Path, and Other Concepts 10
Debugging with the X/Motif Analyzer 11
Special Libraries 12
Using the X/Motif Analyzer 12
Examiners Overview 12
Examiners and Selections 12
Inspecting Data 13
Inspecting the Control Flow 13
Tracing the Execution 13
Restrictions and Limitations 13
Customizing the Debugger 14
Using a Startup File 14
Implementing User-Defined Buttons 15

Managing Source Files [2] **17**

Accessing Files Used by an Executable 17
Opening a New File 18
Path Remapping 19

A Short Debugger Tutorial [3] **23**

Starting the Debugger 23
Performing a Search 25
Setting Traps 27
Examining Data 30

Setting Traps [4] **39**

Page

Trap Terminology 39

Trap Triggers 39

Trap Actions 40

Setting Traps in Main View and `Source View` 40

Setting Traps with the Traps Menu in Main View 40

Setting Traps with the Mouse 42

Setting Traps in Trap Manager 43

Setting Single-process and Multiprocess Traps 43

Setting a Trap Condition 46

Setting a Trap Cycle Count 47

Setting a Trap with the Traps Menu 47

Moving around the Trap Display Area 48

Enabling and Disabling Traps 48

Saving and Reusing Trap Sets 48

Setting Traps With `Signal Panel` and `System Call Panel` 48

Controlling Process Execution [5] **51**

Main View Control Panel 51

Status and Entry Fields in the Main View Control Panel 51

Execution Control Buttons 52

Controlling Process Execution Using the `PC` Menu 55

Execution View 55

Examining Debugger Data [6] **57**

Tracing through `Call Stack View` 57

Evaluating Expressions 60

`Expression View` Window 60

Assigning Values to Variables 62

Evaluating Expressions in C 63

Page

C Function Calls 63
Evaluating Expressions in C++ 64
Limitations 64
Evaluating Expressions in Fortran 64
Fortran Variables 65
Fortran Function Calls 65

Debugging with Fix+Continue: A Tutorial [7] **67**

Setting Up the Sample Session 67
Redefining a Function 69
Editing a Function 69
Changing Code 71
Deleting Changed Code 72
Changing Code From the Debugger Command Line 72
Saving Changes 73
Setting Breakpoints in Redefined Code 73
Viewing Status 76
Comparing Original and Redefined Code 76
Switching Between Compiled and Redefined Code 76
Comparing Function Definitions 77
Comparing Source Code Files 78
Ending the Session 78

Detecting Heap Corruption [8] **79**

Typical Heap Corruption Problems 79
Detecting Heap Corruption Errors 79
Compiling with the Malloc Library 80
Setting Environment Variables 81
Trapping Heap Errors Using the Malloc Library 82

Page

Heap Corruption Detection Tutorial 83

Multiple Process Debugging [9] **89**

Debugging with Multiprocess View 89

Invoking the Parent Process 90

Viewing Process Status 91

Using Control Buttons 91

Multiprocess Traps 92

Adding and Removing Processes 92

Multiprocess Preferences 92

Controlling Execution and Setting Traps in a Multiprocess Program 92

Using Multiprocess View to Control Execution 95

Using the Trap Manager to Control Trap Inheritance 96

Debugging a Multiprocess Fortran Program 98

General Fortran Debugging Hints 98

Multiprocess Debugging Session 98

Using the X/Motif Analyzer: A Tutorial [10] **103**

Setting Up the Sample Session 103

Preparing the Fileset 103

Launching the X/Motif Analyzer 105

Navigating the Widget Structure 105

Examining Widgets 107

Setting Callback Breakpoints 109

Using Additional Features of the Analyzer 112

Ending the Session 116

Appendix A Debugger Reference **117**

Main View 117

Page

Admin Menu 123
Views Menu 126
Query Menu 127
Source Menu 128
Display Menu 130
Perf Menu 131
Traps Menu 133
PC Menu 134
Fix+Continue Menu 134
Show Difference Submenu 135
View Submenu 136
Preferences Submenu 136
Keyboard Accelerators 139
Help Menu 139
Basic Windows 140
Execution View 140
Multiprocess View 140
Viewing Status 141
Multiprocess Control Buttons 142
Controlling Processes 143
Controlling Preferences 144
Source View 145
Process Meter 148
Charts Menu 149
Scale Menu 149
Ada-specific Windows 149
Task View 149
Admin Menu 150
Config Menu 151
Layout Menu 151

Page

`Display` Menu . . . 151
`Exception View` . . . 152
X/Motif Analyzer Windows . . . 155
Global Objects . . . 156
`Admin` Menu . . . 157
`Examine` Menu . . . 157
Examiner Tabs . . . 158
`Return` Button . . . 158
Breakpoints Examiner . . . 158
Callback Breakpoints Examiner . . . 160
Trace Examiner . . . 175
Widget Examiner . . . 177
Tree Examiner . . . 178
Callback Examiner . . . 179
Window Examiner . . . 180
Event Examiner . . . 181
Graphics Context Examiner . . . 182
Pixmap Examiner . . . 183
Widget Class Examiner . . . 184
Project Management Window . . . 185
`Project View Admin` Menu . . . 186
Text Fields . . . 186
Project Display Area . . . 186
Project Pop-up Menu . . . 186
Trap Management Windows . . . 186
Trap Manager . . . 186
`Config` Menu . . . 188
`Traps` Menu . . . 188
`Display` Menu . . . 189

Page

`Signal Panel` 189

`Syscall Panel` 190

Data Examination Windows 190

`Array Browser` Window 191

`Spreadsheet` Menu 195

`Format` Menu 196

`Render` Menu 197

`Color` Menu 197

`Scale` Menu 199

Examiner Viewer Controls 200

`Examiner Viewer` Menu 202

`Call Stack View` Window 204

`Config` Menu 205

`Display` Menu 206

`Expression View` Window 206

`Config` Menu 207

`Display` Menu 207

`Language` Pop-up Menu 207

`Format` Pop-up Menu 207

`File Browser` Window 208

`Structure Browser` Window 209

Using the `Structure Browser Overview` Window to Navigate 211

Entering Expressions 211

Working in the `Structure Browser` Display Area 211

`Structure Browser Display` Menu 212

`Node` Menu 214

Formatting Fields 215

`Variable Browser` Window 219

Entering Variable Values 219

Changing Variable Column Widths 220

Page

Viewing Variable Changes . . . 220
Machine-level Debugging Windows . . . 221
`Disassembly View` . . . 221
Similarities with Main View . . . 222
`Disassemble` Menu . . . 223
Disassembly View Preferences . . . 225
`Register View` Window . . . 227
`Register View` Window . . . 228
Changing the `Register View` Display . . . 229
`Memory View` . . . 230
Viewing a Portion of Memory . . . 231
Changing the Contents of a Memory Location . . . 232
Changing the Memory Display Format . . . 232
Moving around the `Memory View` Display Area . . . 232
Fix+Continue Windows . . . 232
`Fix+Continue Status` Window . . . 234
`Admin` Menu . . . 235
`View` Menu . . . 235
`Fix+Continue` Menu . . . 236
`Fix+Continue Error Messages` Window . . . 238
`Admin` Menu . . . 239
View Menu . . . 239
`Fix+Continue Build Environment` Window . . . 239
Changes to Debugger Views . . . 241
Main View . . . 241
Command Line Interface . . . 242
`Call Stack View` . . . 243
Trap Manager . . . 243
Debugger Command Line . . . 244

Page

Appendix B Using the Build Manager **259**

`Build View` Window 259

Build Process Control Area 260

Transcript Area 261

Error List Area 261

`Build View Admin` Menu 262

`Build View Preferences` 262

Build Options 263

Using `Build View` 264

`Build Analyzer` Window 265

Build Specification Area 266

Build Graph Area 266

Build Graph Control Area 268

`Build Analyzer Overview` Window 269

`Build Analyzer` Menus 270

`Admin` Menu 270

`Build` Menu 270

`Filter` Menu 270

`Query` Menu 271

Index **273**

Figures

Figure 1. Major Areas of Main View 3

Figure 2. Fix and Continue Cycle 7

Figure 3. Launching the X/Motif Analyzer 11

Figure 4. User-Defined Button Example 15

Figure 5. `File Browser` Window 17

Figure 6. `Open` Dialog Box 18

Page

Figure 7. Path Remapping Dialog 20
Figure 8. Main View Window with `jello` Source Code 24
Figure 9. The `jello` Window 25
Figure 10. The Search Dialog 26
Figure 11. Search Target Indicators 27
Figure 12. Stop Trap Indicator 28
Figure 13. Trap Manager Window 29
Figure 14. `Call Stack View` at `spin` Stop Trap 31
Figure 15. `Variable Browser` at `spin` 32
Figure 16. `Variable Browser` after Changes 33
Figure 17. `Expression View` with `Language` and `Format` Menus Displayed 34
Figure 18. `Structure Browser` Window with `jello_conec` Structure 35
Figure 19. `Structure Browser` Window With `Next` Pointer Dereferenced 35
Figure 20. `Array Visualizer` Window for `shadow` Matrix 36
Figure 21. Subscript Control Area in `Array Visualizer` Window 37
Figure 22. `Traps` Menu in Main View 41
Figure 23. Typical Trap Icons 42
Figure 24. Trap Manager `Config`, `Traps`, and `Display` Menus 43
Figure 25. Trap Examples 46
Figure 26. `Signal Panel` and `System Call` Panel 49
Figure 27. Main View Control Panel 51
Figure 28. Pop-up Menu and `Step Into Dialog` 53
Figure 29. Pop-up Menu and `Step Over Dialog` 54
Figure 30. `Call Stack View` Window 58
Figure 31. Tracing through `Call Stack View` 59
Figure 32. `Expression View` with Major Menus Displayed 61
Figure 33. Change Indicators in `Expression View` 62

Page

Figure 34. Debugger Main View With `Fix+Continue` Menu 68
Figure 35. Program Results in `Execution View` 69
Figure 36. Selecting a Function for Redefinition 70
Figure 37. Redefined Function 71
Figure 38. Bounce Window 72
Figure 39. Stopping After Breakpoints in Redefined Code 75
Figure 40. Comparing Compiled and Redefined Function Code 77
Figure 41. Setting Traps to Detect Heap Corruption 85
Figure 42. Heap Corruption Warning Shown in `Execution View` 86
Figure 43. Call Stack at Boundary Overrun Warning 86
Figure 44. Main View at Bus Error 87
Figure 45. `Multiprocess View` 91
Figure 46. Examining Process State Using `Multiprocess View` 96
Figure 47. Modifying a Trap to Affect a Process Group 97
Figure 48. Comparing Variable Values from Two Processes 102
Figure 49. Debugger Main View 104
Figure 50. Program Results in `Execution View` 105
Figure 51. First View of the `X/Motif Analyzer` (`Widget` Examiner) 106
Figure 52. Widget Hierarchy Displayed by the Tree Examiner 107
Figure 53. Adding a Breakpoint for a Widget 108
Figure 54. Setting Breakpoints for a Widget Class 110
Figure 55. Callback Context Displayed by the Callback Examiner 111
Figure 56. Window Attributes Displayed by the Window Examiner 112
Figure 57. Selecting the Breakpoints Tab From the Overflow Area 114
Figure 58. Breakpoint Results Displayed by the `Call Stack View` 115
Figure 59. Major Areas of the Main View Window 118
Figure 60. Show/Hide Annotations Button in Main View 123

Page

Figure 61. The `Search` Dialog 129
Figure 62. `Go to Line` Dialog 129
Figure 63. Preferences Dialog 131
Figure 64. `Perf` Menu and Subwindows 132
Figure 65. `Fix+Continue Preferences Dialog` 137
Figure 66. `Execution View` 140
Figure 67. `Multiprocess View` 141
Figure 68. `Source View` 146
Figure 69. `Process Meter` 148
Figure 70. `Task View` 150
Figure 71. `Exception View` 153
Figure 72. Launching the `X/Motif Analyzer` Window 156
Figure 73. Examiner Tabs 158
Figure 74. Breakpoints Examiner Display in the `X/Motif Analyzer` Window 159
Figure 75. Event-Handler Breakpoints Examiner 162
Figure 76. Timeout-Procedure Breakpoints Examiner 166
Figure 77. Input-Handler Breakpoints Examiner 167
Figure 78. State-Change Breakpoints Examiner 169
Figure 79. X-Event Breakpoints Examiner 171
Figure 80. X-Request Breakpoints Examiner 173
Figure 81. `Request Type Selection` Dialog 174
Figure 82. Trace Examiner 176
Figure 83. Widget Examiner 177
Figure 84. Tree Examiner 178
Figure 85. Callback Examiner 180
Figure 86. Window Examiner 181
Figure 87. Event Examiner 182

Page

Figure 88. Graphics Context Examiner 183
Figure 89. Pixmap Examiner 184
Figure 90. Widget Class Examiner 185
Figure 91. Trap Manager Window 187
Figure 92. Signal Panel 189
Figure 93. Syscall Panel 190
Figure 94. Array Browser with Display Menu Options 192
Figure 95. Subscript Controls: Area in the Array Browser 193
Figure 96. Array Browser Spreadsheet Area 195
Figure 97. Example of Wrapped Array 196
Figure 98. Color Exception Portion of Array Browser Window 198
Figure 99. Array Browser Graphic Modes 199
Figure 100. Examiner Viewer with Controls and Menus 201
Figure 101. Examiner Viewer Preferences Dialog 203
Figure 102. Call Stack View 205
Figure 103. Expression View 206
Figure 104. Expression View Format Popup with Submenus 208
Figure 105. File Browser Window 209
Figure 106. Structure Browser with Menus Displayed 210
Figure 107. Tree and Linked List Arrangements of Structures 213
Figure 108. Structure Browser Preferences Dialog 216
Figure 109. Structure Browser Type Formatting Dialog 217
Figure 110. Variable Browser with Menus Displayed 220
Figure 111. Typical Variable Change Indicators 221
Figure 112. Disassembly View with Menu Displayed 222
Figure 113. Disassemble From Address Dialog 223
Figure 114. Disassemble Function Dialog 224

Page

Figure 115. Disassemble File Dialog 225
Figure 116. Disassembly View Preferences Dialog with Pop-up Menu 226
Figure 117. Register View 228
Figure 118. Register View Preferences Dialog 230
Figure 119. Memory View with Menu Displayed 231
Figure 120. Fix+Continue Menu Selections 233
Figure 121. Fix+Continue Status Window 234
Figure 122. Fix+Continue Status Window Menus 235
Figure 123. Fix+Continue Error Messages Window 238
Figure 124. Fix+Continue Build Environment Window 240
Figure 125. Debugger Main View 242
Figure 126. Command Line Interface with Redefined Function 243
Figure 127. Call Stack View 243
Figure 128. Trap Manager Window with Redefined Function 244
Figure 129. Build Process Control Area in Build View Window 260
Figure 130. Build View Window with Typical Data 261
Figure 131. Build View Preferences Dialog 263
Figure 132. Build Options Dialog 264
Figure 133. Build Analyzer Window 266
Figure 134. Build Graph Icons 268
Figure 135. Build Graph Control Area 268
Figure 136. Build Analyzer Overview Window with Build Analyzer Graph 269

Tables

Table 1. Fix and Continue Compile Time Cycle 6
Table 2. Valid C Operations 63
Table 3. Valid Fortran Operations 65
Table 4. Fix+Continue Keyboard Accelerators 139

About This Guide

This publication documents the WorkShop release 2.7 running on IRIX systems.

The Debugger is a source-level debugging tool that displays program and data and execution status.

This manual contains the following chapters:

- Chapter 1, page 1, describes how to get started using the Debugger and gives a general overview of Debugger functionality.
- Chapter 2, page 17, describes how to manage source files.
- Chapter 3, page 23, presents a short Debugger tutorial.
- Chapter 4, page 39, describes how to set various types of traps.
- Chapter 5, page 51, describes methods for controlling process execution.
- Chapter 6, page 57, explains how to examine Debugger data.
- Chapter 7, page 67, presents a short tutorial using Fix and Continue.
- Chapter 8, page 79, describes heap corruption problems and how to detect them.
- Chapter 9, page 89, describes debugging multiprocess programs.
- Chapter 10, page 103, presents a short tutorial using the X/Motif Analyzer
- Appendix A, page 117, describes all of the Debugger windows, menus, and other features in detail.
- Appendix B, page 259, describes use of the Build Manager.

Related Publications

The following documents contain additional information that may be helpful:

- *C++ Language System Library*
- *C++ Language System Overview*
- *C++ Language System Product Reference Manual*
- *C++ Programmer's Guide*

- *Developer Magic: Performance Analyzer User's Guide*
- *Developer Magic: ProDev WorkShop Overview*
- *Developer Magic: Debugger User's Guide*
- *Fortran 77 Language Reference Manual*

Obtaining Publications

Silicon Graphics maintains publications information at the following World Wide Web site:

`http://techpubs.sgi.com/library`

The preceding website contains information that allows you to browse documents online, order documents, and send feedback to Silicon Graphics.

To order a printed Silicon Graphics document, call 1–800–627–9307.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage(`*x*`)`	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers:

1	User commands
1B	User commands ported from BSD
2	System calls
3	Library routines, macros, and opdefs
4	Devices (special files)

	4P	Protocols
	5	File formats
	7	Miscellaneous topics
	7D	DWB-related information
	8	Administrator commands
	Some internal routines (for example, the `_assign_asgcmd_info`() routine) do not have man pages associated with them.	
variable	Italic typeface denotes variable entries and words or concepts being defined.	
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.	
[ ]	Brackets enclose optional portions of a command or directive line.	
...	Ellipses indicate that a preceding element can be repeated.	

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail at the following address:

  `techpubs@sgi.com`

- Contact your customer service representative and ask that an SPR or PV be filed. If filing an SPR, use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.

- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600

- Send a facsimile of your comments to the attention of “Software Publications Group” in Eagan, Minnesota, at fax number +1–612–683–5599.

We value your comments and will respond to them promptly.

Getting Started with the WorkShop Debugger [1]

The WorkShop Debugger is a UNIX source-level debugging tool that provides special windows (*views*) for displaying program data and execution status. These views update as the program executes. This chapter presents an overview of the WorkShop Debugger and is divided into these sections:

- Typical Debugger Usage, Section 1.1, page 1
- Debugging with Fix and Continue, Section 1.4, page 14

1.1 Typical Debugger Usage

This section provides a general description of debugging software with WorkShop. It covers these topics:

- Starting and Exiting the Debugger, Section 1.1.1, page 1
- Using Main View, Section 1.1.2, page 2
- Setting Traps, Section 1.1.3, page 3
- Inspecting Debugger Data, Section 1.1.4, page 4
- Changing Source Code, Section 1.1.5, page 4
- Integration With Other WorkShop Tools, Section 1.1.6, page 5

1.1.1 Starting and Exiting the Debugger

To start the Debugger, use the following syntax:

```
cvd [-pid pid] [-host host] [executable [corefile]] [&]
```

The `-pid` option lets you attach the Debugger to a running process. You can use this to determine why a live process is in a loop.

The `-host` option lets you specify a remote host on which the target executable will be run while the Debugger runs locally. This option is useful in the following circumstances:

- You do not want the Debugger windows to interfere with the application you are debugging.
- You are supporting an application remotely.
- You do not want to use the Debugger on the target machine for any other reason.

The *executable* argument is the name of the executable file for the process you want to run. It is optional; you can invoke the Debugger first and specify the executable file name later.

You can also invoke the Debugger with the *corefile* argument to specify a core file (with its executable file). This will aid you in determining why a program crashed.

To exit the Debugger, you can select from several options:

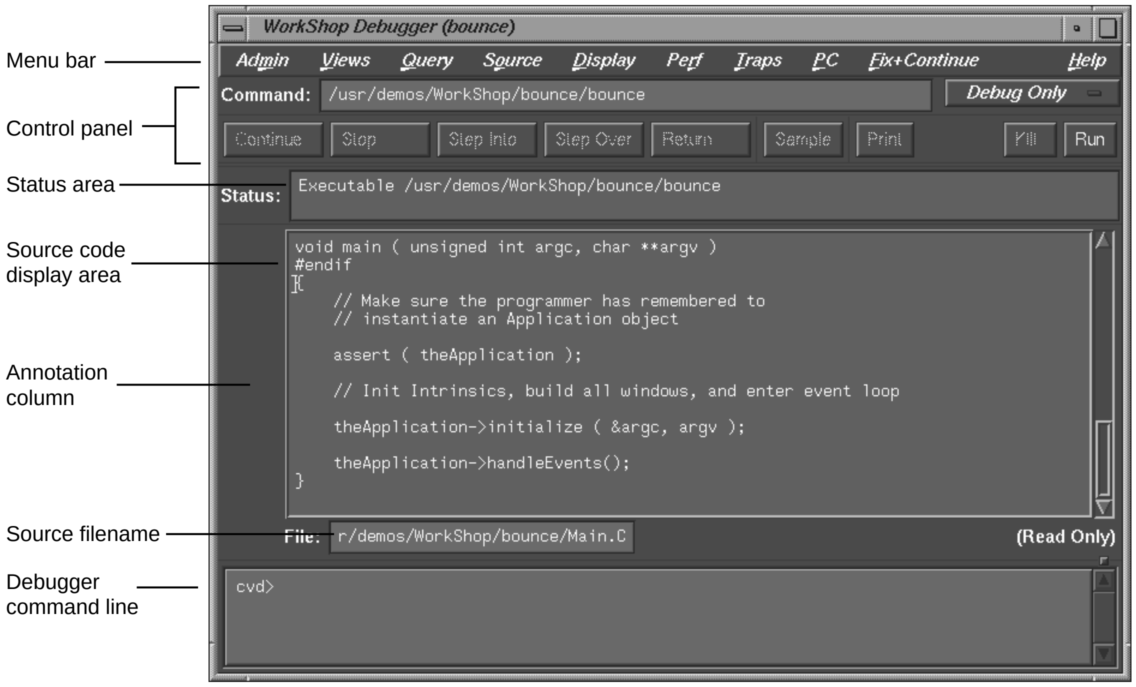
- Select `Exit` from the `Admin` menu in the Main View window.
- Type **`quit`** at the Debugger command line.
- Press `Ctrl-c` in the same window where you entered the **`cvd`** command.
- Double-click the window system menu or select the `Quit` entry.

1.1.2 Using Main View

Starting the Debugger with an executable file brings up the Main View, loaded with source code and ready to run the process with any specified arguments. You perform most of your work in Main View, which displays the following:

- A menu bar for performing Main View functions and for accessing other views
- A control panel for specifying and controlling the process to be debugged
- A source code display area for inspecting the source code
- A status line for viewing the state of the program
- The Debugger command line for entering debugging commands (see Section A.10, page 244, for the syntax)

The major areas of the Main View window are shown in Figure 1, page 3.



```
/usr/demos/WorkShop/bounce/bounce
Executable /usr/demos/WorkShop/bounce/bounce
void main ( unsigned int argc, char **argv )
#endif
{
    // Make sure the programmer has remembered to
    // instantiate an Application object

    assert ( theApplication );

    // Init Intrinsics, build all windows, and enter event loop

    theApplication->initialize ( &argc, argv );

    theApplication->handleEvents();
}
r/demos/WorkShop/bounce/Main.C
cvd>
```

Figure 1. Major Areas of Main View

1.1.3 Setting Traps

A major part of the debugging process is inspecting data at various points during execution. A trap is a mechanism for gathering this data. A *stop trap* halts a process so that you can manually examine data. A *sample trap* collects specific performance data without stopping.

The Debugger lets you set traps at the following points:

- At a line in a file (a *breakpoint*)
- At an instruction address
- On entry to or exit from a function
- When a signal is received
- When a system call is made, at either the entry or exit point

- When a given variable or address is written to, read from, or executed (a *watchpoint*)
- At set time intervals (a *pollpoint*)

For more information on traps, refer to Chapter 4, page 39.

1.1.4 Inspecting Debugger Data

When you stop a process, you have several options for examining the data. You can inspect the following types of data:

- The call stack at the breakpoint by using the `Call Stack` View
- The value of specified expressions by using `Expression View`
- The values, types, or addresses of variables by using the Variable Browser
- Data structures by using the Structure Browser
- The values of an array variable by using the Array Browser
- The values in specified memory locations by using `Memory View` or registers by using `Register View`
- The disassembled code by using `Disassembly View`

For more information on the various Debugger views, refer to Chapter 6, page 57.

1.1.5 Changing Source Code

To change your source code and recompile, follow these steps:

1. Switch to a text editor by using the `Fork Editor` selection in the `Source` menu.

   If you are using a configuration management system, you can check out the source code by selecting `Versioning` from the `Source` menu and accessing the source through the configuration management shell.

2. Make any changes and save them. In the Main View, pull down the `Source` menu and select the `Save...` option.

3. In the Main View, pull down the `Source` menu and select `Recompile`.

The `Build View` window displays and lets you start the compile. Any compile errors are listed in the window, and you can access the related source code by clicking the errors. For more information on the Build Manager, refer to Appendix B, page 259.

When the code is rebuilt successfully, the new executable file reattaches automatically to the Debugger and the Static Analyzer. Previously set traps are intact unless you have traps triggered at line numbers and have changed the line count.

1.1.6 Integration With Other WorkShop Tools

The WorkShop tools are designed so that you can move easily between them in a work session.

1.1.6.1 Accessing the Performance Analyzer from Main View

You can switch to the Performance Analyzer at any time while debugging. Selecting `Performance Task...` from the `Admin` menu lets you enable data collection for your experiment. You must specify a performance task before a process is run. This enables correct data collection. If you are in the middle of a run, you must terminate the run, select a task, and restart the target to collect the data. Selecting `Performance Analyzer` from the `Launch Tool` submenu displays the main Performance Analyzer view for analyzing experiment results.

1.1.6.2 Accessing the Static Analyzer from Main View

You can access the Static Analyzer from the `Launch Tool` submenu. The Static Analyzer displays source information making it easy to set traps in your source code.

1.1.6.3 Accessing Editors from Main View

After you solve a problem with the WorkShop tools, you may wish to make the change in `Source View` (or your preferred editor) and then recompile.

1.1.6.4 Accessing Configuration Management Tools

If you use ClearCase (available from Silicon Graphics), RCS, or SCCS for configuration management, you can integrate the tool into the WorkShop environment by typing

```
cvconfig [clearcase | rcs | sccs]
```

This enables the `Versioning` selection of the `Source` menu that provides selections for checking files in and out.

1.1.6.5 Recompiling from Main View

To access the `Build View` window, which lets you start the compile process, from the Main View pull down the `Source` menu and select `Recompile`. To examine the build dependencies from the Main View, pull down the `Admin` menu and select `Build Analyzer`. For more information on the Build Manager tools, see Appendix B, page 259.

1.2 Debugging with Fix and Continue

Fix and Continue is integrated with the Debugger. You issue Fix and Continue commands graphically from the `Fix+Continue` menu of the Debugger Main View. You may also issue Fix and Continue commands from the Debugger command line.

1.2.1 Redefining Functions Using Fix and Continue

Fix and Continue gives you the ability to make changes to a program you are debugging without having to recompile and link the entire program, and then continue debugging the code. With Fix and Continue, you can edit a function, parse the new function, and continue execution of the program being debugged.

Table 1, page 6, compares the cycle time in seconds between a full rebuild and a Fix and Continue for three typical programs.

Table 1. Fix and Continue Compile Time Cycle

Example	Time to Rebuild	Time to Fix+Continue
Program A	0:06	0:02
Program B	0:33	0:06
Program C	5:24	0:49

1.2.1.1 Fix and Continue Functionality

Fix and Continue lets you perform the following activities:

- Redefine existing function definitions
- Disable, reenable, save, and delete redefinitions
- Set breakpoints in and single-step within redefined code
- View the status of changes
- Examine differences between original and redefined functions

The basic cycle of using Fix and Continue is shown in Figure 2, page 7.

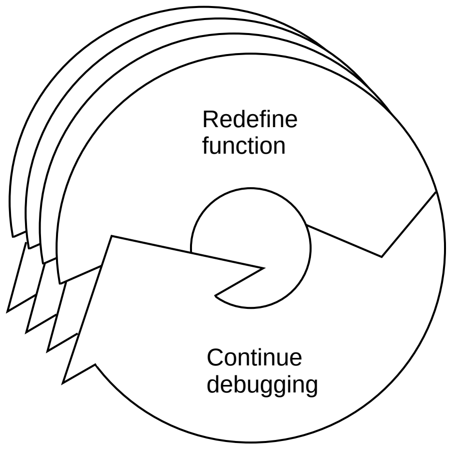


Figure 2. Fix and Continue Cycle

A typical session would have the following scenario:

1. Using Fix and Continue commands, you redefine a function. When you continue executing the program, the Debugger attempts to call the redefined function. If it cannot, an information pop-up window appears and the redefined function will be executed the next time the program calls that function.

2. You redefine other functions, alternating between debugging, disabling, reenabling, and deleting redefinitions. You might save function redefinitions to their own files, or save files to a different name, to be used later with the present or with other programs.

Frequently during debugging you can review the status of changes by listing them, showing specific changes, or looking at the Fix and Continue `Status`

`View`. You can compare changes to an individual function or to an entire file with the compiled versions. When you are satisfied with the behavior of your application, save the file thus replacing the compiled source.

1.2.1.2 Fix and Continue/WorkShop Integration

Using Fix and Continue interacts with the following WorkShop tools:

- The WorkShop Debugger Main View, the `Source View`, and the `Fix+Continue Status` window make a clear distinction between compiled and redefined code and allow editing only in redefined code.
- The following WorkShop entities are knowledgeable about redefined code:
  - `Call Stack` window
  - Trap Manager
  - Debugger command line

1.2.1.3 How Redefined Code Is Distinguished from Compiled Code

Redefined functions have an identification number and special line numbers and in the Debugger views are color–coded according to their state (edited, parsed, and so on).

Line numbers in the compiled file stay the same, no matter how redefined functions change. However, when you begin editing a function, the line numbers of the function body are represented in decimal notation (*n*.1, *n*.2, ..., *n*.*m*). *n* is the compiled line number where the function body begins. *m* is the line number relative to the beginning of the function body, starting with the number one.

The `Call Stack` window and the Trap Manager functions both use function-relative decimal notation when referring to a line number within the body of a redefined function.

The Debugger command line reports ongoing status. In addition to providing the same commands available from the menu, edit commands allow you to add, replace, or delete lines from files. Therefore, you can operate on several files at once.

1.2.1.4 Restrictions on Fix and Continue

Fix and Continue has the following restrictions when you fix a function in which you have stopped:

- When using Fix and Continue with C code. you must use the `-o32` compiler option. If you do not, you will be prompted for the option.
- Fix and Continue does not support C++ templates.
- You may not add, delete, or reorder local variables in a function.
- You may not change the type of a local variable.
- You may not change a local variable to be a register variable and vice- versa.
- You may not add any function calls that increase the size of the parameter area.
- You may not add an `alloca` function to a frame that did not previously use an `alloca` function.
- Both the old and new functions must be compiled with the `-g` option.

  In other words, the layout of the stack frames of both the old and new functions must be identical for you to continue execution in the function that is being modified. If not, execution of the old function continues and the new function is executed the next time the function is called.
- If you redefine functions that are in but not on top of the call stack, the modified code will not be executed when they combine. Modified functions will be executed only on their next call or on a rerun.

  For example, consider the following call stack:

  ```
  foo()
  bar()
  foobar()
  main()
  ```

  – If you redefine `foo()`, you can continue execution provided the layout of the stack frames are the same.

  – If you redefine `main()` after you have run, it will be executed only when you rerun.

   – If you redefine `bar()` or `foobar()`, the new code will not be executed when `foo()` returns. The code will be executed only on the next call of `bar()` or `foobar()`.

1.2.2 The Fix and Continue Environment

The interface to Fix and Continue is through the `Fix+Continue` menu and its associated windows: `Status, Message`, and `Build Environment`. These windows are dependent on Fix and Continue, and do not operate unless it is installed.

For more complete information on all of the Fix and Continue menus, windows, and functions, see Section A.9, page 232.

1.2.2.1 Debugger With Fix and Continue Support

Without Fix and Continue, the Debugger source views are `Read-Only` by default. That is so you can examine your files with no risk of changing them. When you select `Edit` from the `Fix+Continue` menu, the Debugger source code status indicator (in the lower-right corner of the Debugger window remains `Read-Only`. Edits made using Fix and Continue are saved in an intermediate state. You must choose the `Save File+Fixes As...` selection to save your edits.

When you edit a function, it is highlighted in color. If you switch to the compiled version of your code, the color changes to show that the function has been redefined. If you try to edit the compiled version of your code, the Debugger beeps indicating `Read-Only` status.

When you have completed your edits and want to see the results, select the `Parse and Load` menu option. When the parse and load has executed successfully, the color changes again. If the color does not change, there may be errors; check the `Message Window`.

1.2.2.2 Change ID, Build Path, and Other Concepts

The Fix and Continue methods for accessing functions through ID numbers, finding files, and so forth, are discussed below:

- Each redefined function is numbered with a change ID. Its status may be shown as `redefined`, `enabled`, `disabled`, `deleted`, or `detached`.
- Fix and Continue needs to know the location of include files and other parameters specified by compiler build flags. You can set the build

environment for all files or for a specific file. You can display the current build environment from the `Fix+Continue` menu, the command line, or the `Fix and Continue Status` Window. When you finish a Fix and Continue session, you can unset the build environment.

- Output from a successful run is displayed in the `Execution View`. This functionality is the same as it is in the Debugger without Fix and Continue.

1.3 Debugging with the X/Motif Analyzer

The X/Motif Analyzer provides specific debugging support for X/Motif applications. The X/Motif analyzer is integrated with the Debugger. You issue X/Motif analyzer commands graphically from the X/Motif analyzer subwindow of the Debugger Main View (see Figure 3, page 11). To access this subwindow, select `X/Motif Analyzer` from the `Views` menu.

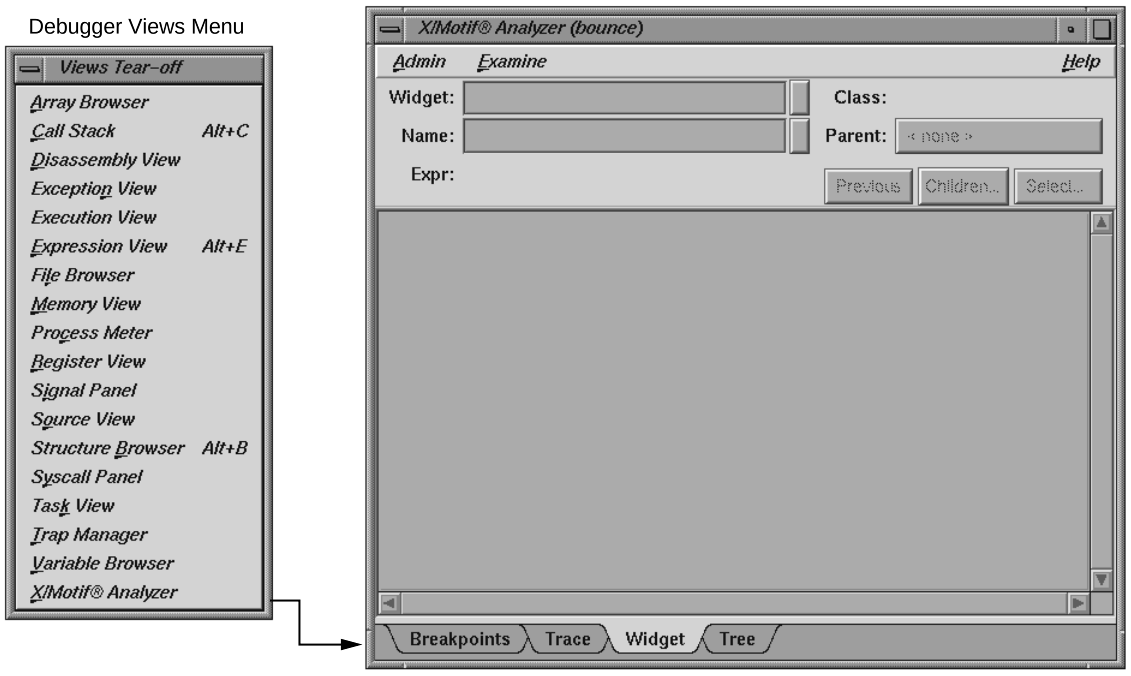


Figure 3. Launching the X/Motif Analyzer

1.3.1 Special Libraries

When you first bring up the X/Motif Analyzer, it may ask you if you want to change the `$LD_LIBRARY_PATH` variable to include `/usr/lib/WorkShop/Motif`. In that directory are instrumented versions of the Silicon Graphics `Xlib`, `Xt`, and `Xm` libraries. These versions include debugging symbols and special support for X/Motif Analyzer functions.

It is strongly recommended that you click on the `OK` button and use these libraries because they are the Silicon Graphics enhanced versions of these libraries. Clicking `OK` enables all features of the X/Motif Analyzer. There are no instrumented MIPS/ABI versions of the libraries.

1.3.2 Using the X/Motif Analyzer

The Analyzer contains examiners for X/Motif objects (for example, widgets and X graphics contexts) that can be difficult or impossible to inspect with ordinary debugger functionality. The Analyzer also allows you to set widget-level breakpoints and collect X–event history information in the same manner as using the `xscope`(1) command.

1.3.2.1 Examiners Overview

When you first bring up the X/Motif Analyzer, the `X/Motif Analyzer` window is displayed and, by default, it is set to examining widgets. The window may be blank or may display a widget found in the call stack of a stopped process.

At the bottom of the `X/Motif Analyzer` window is a tab panel showing the current set of examiners. In addition to the `Widget` examiner, `Breakpoints`, `Trace`, and `Tree` examiners are available by default and appear on tabs at the bottom of the window. These four tabs are always present. Other examiners are available from the `Examine` menu of the `X/Motif Analyzer` window.

Some examiners cannot be manually selected—they appear only when the call stack context is appropriate. For example, the `Callback` examiner appears only when a process is stopped somewhere in a widget callback.

1.3.2.2 Examiners and Selections

If you select text in one examiner and then choose another examiner by using the `Examine` menu, the new examiner is brought up and the text is used as an expression for it. If you selected text that is an inappropriate object for the new examiner, an error is generated.

Alternatively, you can select text, pull down the `Examine` menu, and choose `Selection`. The X/Motif Analyzer attempts to select an appropriate examiner for the type of selected text. If the type of the text is unknown, the `Couldn't examine selection in more detail` error message is displayed. Otherwise, the appropriate examiner is chosen and the text is evaluated.

You can accomplish the same thing by triple-clicking on a line of text. If the type of the text is unknown, nothing happens. Otherwise, the appropriate examiner is chosen and the text is evaluated.

1.3.2.3 Inspecting Data

X/Motif applications consist of collections of objects (Motif widgets) and make extensive use of X resources such as windows, graphics context, and so on. The construction model of an X window system hinders you from inspecting the internal structures of widgets and X resources because you are presented with ID values. The X/Motif Analyzer lets you to see the data structures behind the ID values.

1.3.2.4 Inspecting the Control Flow

Traditional debuggers enable you to set breakpoints only in source lines or functions. With the X/Motif Analyzer, you can set breakpoints for specific widgets or widget classes, for specific control flow constructs like callbacks or event handlers, and for specific X events or requests.

1.3.2.5 Tracing the Execution

The X/Motif Analyzer can trace `Xlib`-level server events and client requests, `Xt`-level event dispatching information, widget life cycle, and widget status information.

1.3.3 Restrictions and Limitations

The X/Motif Analyzer has the following restrictions and limitations:

- The `Breakpoints` examiner is active only after you have stopped a process and if you have changed `$LD_LIBRARY_PATH`.

- Sometimes, gadget names may be unavailable and are displayed as `<object>`. You can minimize this condition by getting the widget tree beforehand.
- `editres` requests (such as, widget selection and widget tree) work only if the process is running or if the process is stopped outside of a system call. This can be annoying when the process is stopped in `select()`, waiting for an X server event.
- The process state and appearance of the Debugger Main View flickers while the X/Motif Analyzer tries to complete an `editres` request when the process is stopped.
- `editres` requests may be unreliable if the process is stopped.

1.4 Customizing the Debugger

If there are Debugger commands or combinations of Debugger commands that you use frequently, you may find it convenient to create a script composed of Debugger commands. Debugger scripts are ASCII files containing one Debugger command and its arguments per line. A Debugger script can in turn call other Debugger scripts. There are three general methods for running scripts:

- Entering the `source` command and the filename at the Debugger command line. This is useful for scripts that you need only occasionally.
- Including the script in a startup file. This is useful for scripts that you want implemented every time you use the Debugger.
- Defining a button in the graphical interface to run the script. Use this method for scripts you use frequently but apply only at specific times during a debugging session.

1.4.1 Using a Startup File

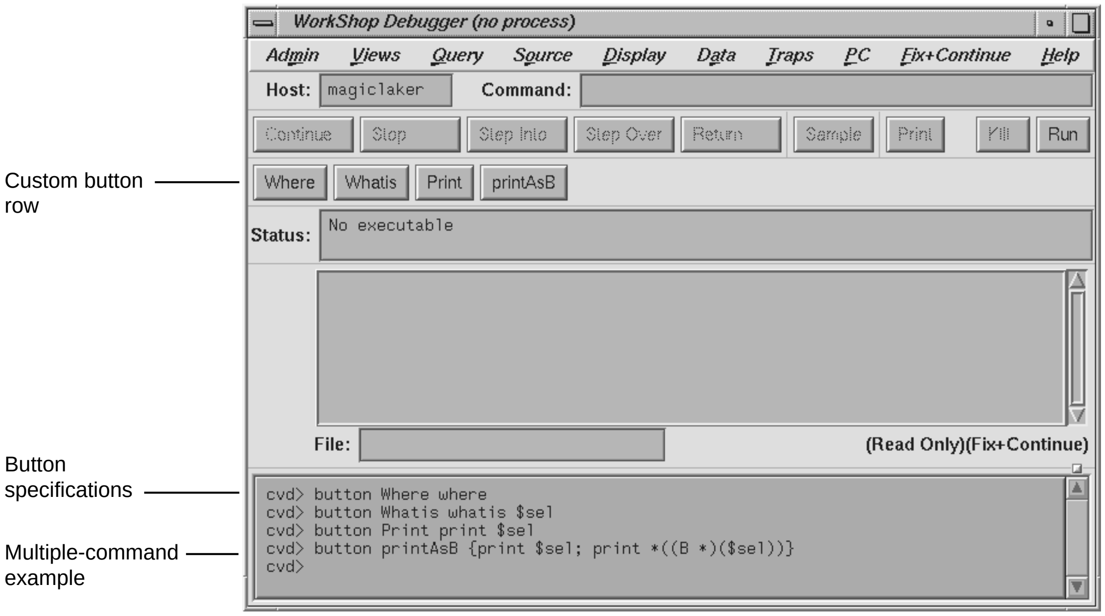
A startup file lets you preload your favorite buttons and aliases in a file that runs when `cvd` is invoked. It also is useful if you have traps that you set the same way each time. The suggested name for the startup file is `.cvdrc`. However, you can select a different name as long as you specify its path in the `CVDINIT` environment variable. The Debugger uses the following criteria when looking for a startup file:

1. Checks the `CVDINIT` environment variable.

2. Check for a `.cvdrc` file in the current directory.
3. Checks for a `.cvdrc` file in the user's home directory.

1.4.2 Implementing User-Defined Buttons

You can implement buttons by providing a special Debugger startup file or by creating them on the fly within a debugging session. Buttons appear in the order of implementation in a row at the bottom of the control panel area. Currently, you can define only one row of custom buttons. Figure 4, page 15, is a typical example of the Debugger Main View with user-defined buttons. The definitions for the user-defined buttons display in the Debugger command line area.


Custom button row
Button specifications
Multiple-command example
WorkShop Debugger (no process)
Admin Views Query Source Display Data Traps PC Fix+Continue Help
Host: magiclaker
Command:
Continue Stop Step Into Step Over Return Sample Print Kill Run
Where Whatis Print printAsB
Status: No executable
File:
(Read Only)(Fix+Continue)

```
cvd> button Where where
cvd> button Whatis whatis $sel
cvd> button Print print $sel
cvd> button printAsB {print $sel; print *((B *)($sel))}
cvd>
```

Figure 4. User-Defined Button Example

The syntax for creating a button is as follows:

```
button label command [$sel]
```

The syntax for creating a multiple-command button is as follows:

```
button label {command1 [$sel]; command2 [$sel]; ...}
```

The button command accepts the following options:

label	Specifies the button name. Button labels should be kept short since there is only room for a single row of buttons. There can be no spaces in a label.
command	Specifies one of the Debugger commands, which are entered at the command line at the bottom of Main View. See Section A.10, page 244.
`$sel`	Provides a substitute for the current cursor selection and should be appropriate as an argument to the selected command.
commandn...	Specifies Debugger commands to be applied in order. Commands must be separated by semicolons (;) and enclosed by braces ({}). The multiple-command button is a powerful feature; it lets you write a short script to be executed when you click the button.

The following command displays a list of all currently defined buttons:

```
button
```

The following command deletes the button corresponding to the label:

```
unbutton label
```

You might use this command if you needed room to create a new button. The effect of `unbutton` is temporary so that subsequently running the startup file reactivates the button.

The following command displays the definition of the specified button, if it exists. If the button does not exist, an error message is displayed:

```
button label
```

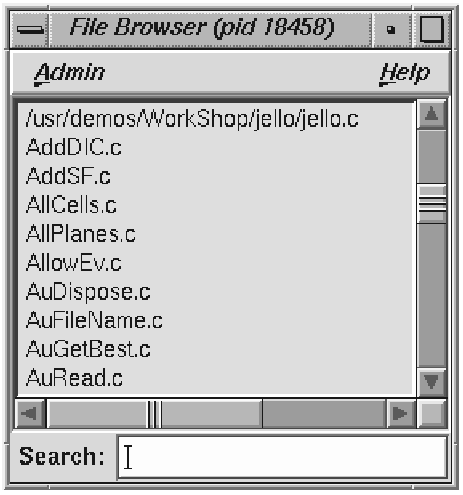
Managing Source Files [2]

This chapter looks at the details of working with source files. It covers these topics:

- Accessing Files Used by an Executable, Section 2.1, page 17
- Opening a New File, Section 2.2, page 18
- Path Remapping, Section 2.3, page 19

2.1 Accessing Files Used by an Executable

The `File Browser` window, available from the `Views` menu in the Main View, provides a scrollable list of the source files used by your executable file, including any files in linked libraries. See Figure 5, page 17.



```
/usr/demos/WorkShop/jello/jello.c
AddDIC.c
AddSF.c
AllCells.c
AllPlanes.c
AllowEv.c
AuDispose.c
AuFileName.c
AuGetBest.c
AuRead.c
```

Figure 5. `File Browser` Window

The `File Browser` has a field labeled `Search` for quickly locating files in the list. File searching is incremental—as you type the string you are searching for in the `Search` field, the first string that matches the entered string is highlighted.

To load a file directly into Main View from the `File Browser` window, simply double-click on the file name.

2.2 Opening a New File

Another way to load a file is to specify it by using the `Open...` selection from the `Source` menu. The dialog box, as shown in Figure 6, page 18, lists all available files and the currently selected directory in the `Selection` field.

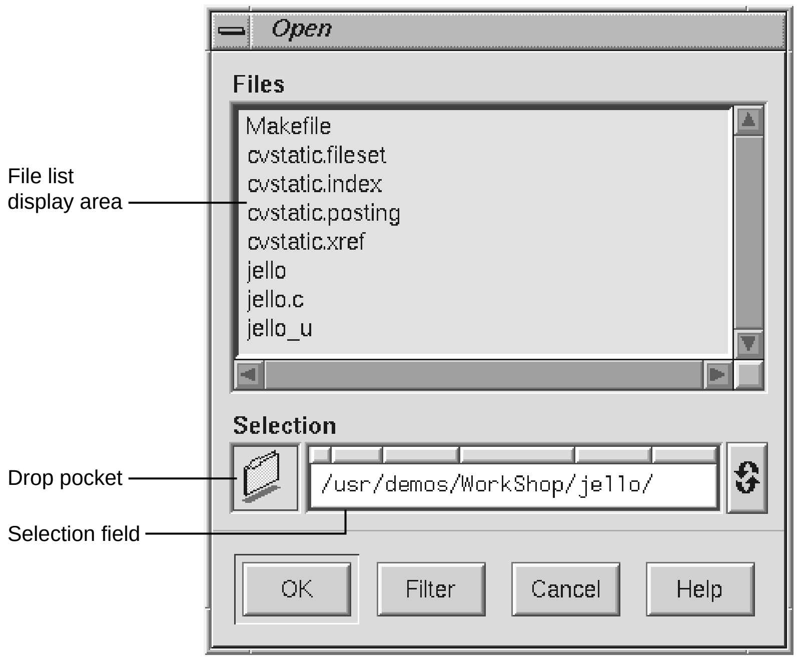


Figure 6. `Open` Dialog Box

There are several ways to load a file:

- Double-click on the file name
- Type the full pathname of the file in the `Selection` field and click the `OK` button
- Drag the file's icon into the drop pocket

If the file you want to load is not in the current directory, enter the appropriate directory in the `Selection` field. Files in the new directory will be listed in the file list.

If you specify a file name without a full path, the Debugger uses the current path remapping information to try to locate the file.

You can also open a file in Main View by entering the full file name in the `File` field, below the source code display area, and press the `Enter` key.

2.3 Path Remapping

Path remapping allows you to modify the set of mappings used to redirect file names located in your executable file to their actual locations in your file system. Since WorkShop uses full (absolute) pathnames, path remapping generally is not necessary. However, if you have mounted executable files on a different tree from the one on which they were compiled, you will need to remap the root prefix to get access to the files in that hierarchy.

The most basic remapping is for ".", which allows you to specify the directories to be searched for files. This basic function works just like `dbx` and can be modified by using the `use`(1) and `dir`(1) commands in the command line. To open the `Path Remapping` dialog box, choose `Remap Paths...` from the `Project` submenu in the Main View `Admin` menu. The `Path Remapping` dialog box appears (see Figure 7, page 20).

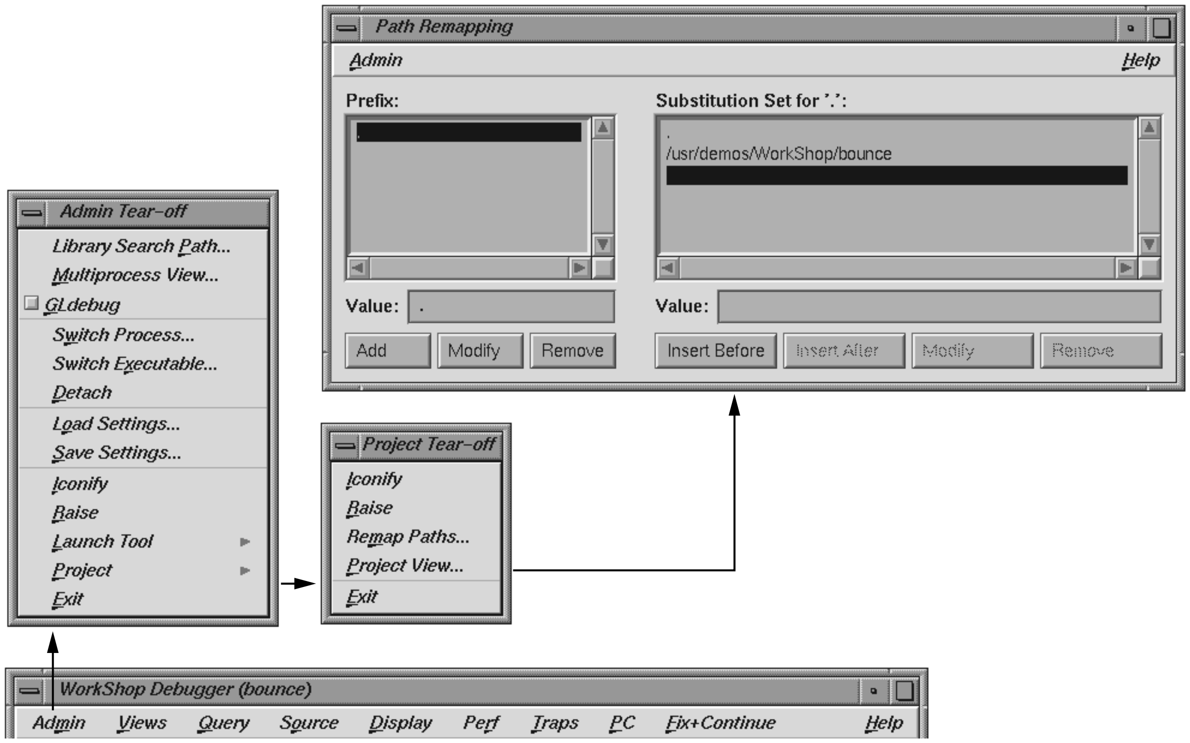


Figure 7. Path Remapping Dialog

For each prefix listed in the `Prefix` list, there is an ordered set of substitutions that is used to find a real file. By default, path remapping is initialized so that "." is mapped to the current directory. The `Substitution Set` list shows the substitution list for the currently highlighted item in the `Prefix` list. You can perform the following operations using the `Path Remapping` dialog box:

- To view the substitution set for a different prefix, click that prefix.
- To add a new prefix, enter the new value in the `Value field` below the `Prefix` list and click the Add button. A new, empty substitution set is created. Next, type the desired substitution in the `Value` field below the `Substitution Set` list.
- To modify the currently selected prefix, edit the string in the `Value` field and click the `Modify` button.

- To remove the current prefix and its substitution set, select the prefix and click the `Remove` button.

A Short Debugger Tutorial [3]

This chapter presents a short tutorial for using the Debugger. The tutorial applies the Debugger to a program called `jello`, which provides a walk through some typical debugging situations. The tutorial is divided into four parts:

- Starting the Debugger, Section 3.1, page 23
- Performing a Search, Section 3.2, page 25
- Setting Traps, Section 3.3, page 27
- Examining Data, Section 3.4, page 30

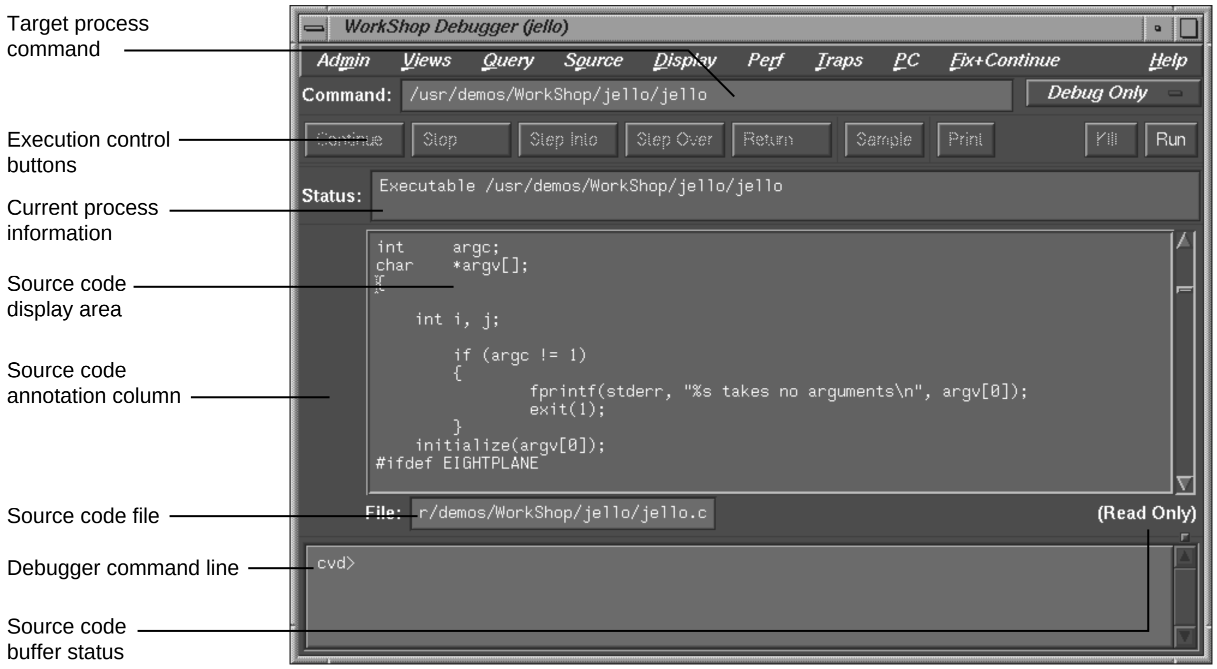
Note: WorkShop identifies files with the pathnames in which they were compiled. The pathnames in the tutorial may not match the ones on your system.

3.1 Starting the Debugger

In this part of the tutorial, you invoke the Debugger and start a typical process running. The `jello` program simulates an elastic polyhedron bouncing around inside of a revolving cube. The program's functionality is mainly contained in a single loop that calculates the acceleration, velocity, and position of the polyhedron's vertices.

1. **`cd`** to the directory `/usr/demos/WorkShop/jello`.
2. Enter **`ls`** to display directory contents.
3. If the `jello` file does not exist, type: **`make jello`**
4. To invoke the Debugger, type: **`cvd jello`**

   The Main View window appears as shown in Figure 8, page 24. The display scrolls automatically to the `main` function.



Command: /usr/demos/WorkShop/jello/jello

Status: Executable /usr/demos/WorkShop/jello/jello

```
int     argc;
char    *argv[];
{

    int i, j;

        if (argc != 1)
        {
                fprintf(stderr, "%s takes no arguments\n", argv[0]);
                exit(1);
        }
    initialize(argv[0]);
#ifdef EIGHTPLANE
```

File: r/demos/WorkShop/jello/jello.c

```
cvd>
```

Figure 8. Main View Window with `jello` Source Code

Note: Main View brings up the source file in read-only mode to avoid inadvertent changes during debugging. You can change this mode by selecting `Make Editable` from the `Source` menu (provided you have the proper file access permissions).

5. Click the `Run` button in the upper-right corner of the Main View to start the `jello` process.

   The `jello` window opens on your display (see Figure 9, page 25). Enlarge this window to watch the program execute. The polyhedron is initially suspended in the center of the cube.

6. Click the left mouse button anywhere inside the `jello` window.

   The polyhedron drops to the floor of the cube.

7. Hold down the right mouse button to display the pop-up menu and select `spin`.

The cube now rotates and the polyhedron bounces. If you select `Display` from the menu, you can change the appearance of the polyhedron to display points only, lines only, full color, visible points only, or single color.

Note: You may encounter flashing colors inside windows while running `jello`. This is normal.

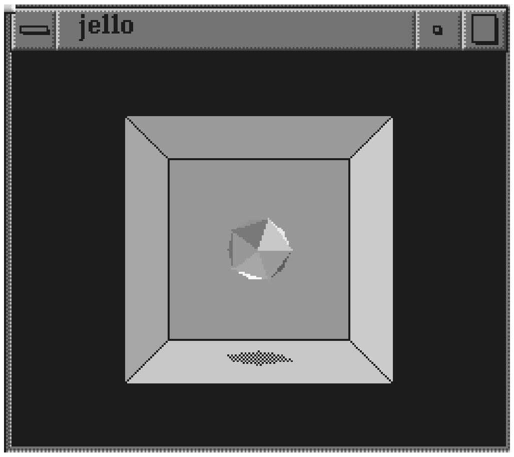


Figure 9. The `jello` Window

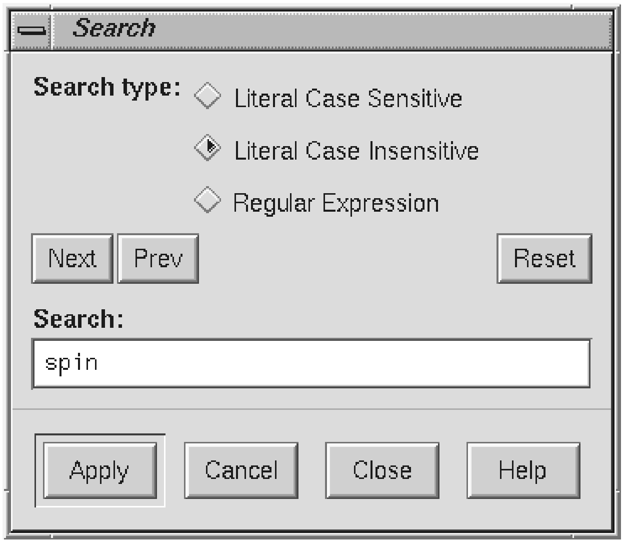
3.2 Performing a Search

This part of the tutorial covers the search facility in the Debugger. You will search through the `jello` source file for a function called `spin`. The `spin` function recalculates the position of the cube.

1. Choose `Search` from the `Source` menu.

   The `Search` dialog box appears.

2. Type **`spin`** in the entry field in the dialog box, as shown in Figure 10, page 26.



Search type: Literal Case Sensitive [✓ Literal Case Insensitive] Regular Expression

Search: spin

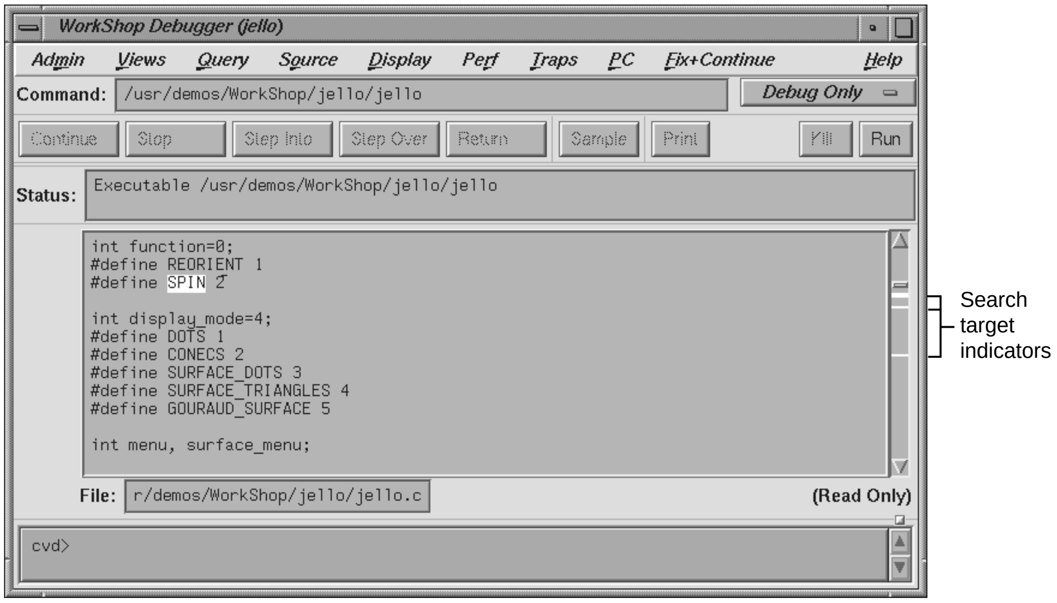
Figure 10. The Search Dialog

3. Click the `Apply` button.

   The search takes place and each instance of `spin` is highlighted in the source code and flagged in the scroll bar to the right of the display area. Figure 11, page 27 shows typical search target indicators. The `Next` and `Prev` buttons let you move from one occurrence to the next in the order indicated. For more information on Search, see Section A.1.4, page 128.

4. Click the `Close` button and the dialog box disappears.

5. Click the middle mouse button on the last search target indicator. This scrolls the source code down to the last occurrence, which is the location of the `spin` function.



Command: /usr/demos/WorkShop/jello/jello

Status: Executable /usr/demos/WorkShop/jello/jello

```
int function=0;
#define REORIENT 1
#define SPIN 2

int display_mode=4;
#define DOTS 1
#define CONECS 2
#define SURFACE_DOTS 3
#define SURFACE_TRIANGLES 4
#define GOURAUD_SURFACE 5

int menu, surface_menu;
```

File: r/demos/WorkShop/jello/jello.c (Read Only)

```
cvd>
```

Figure 11. Search Target Indicators

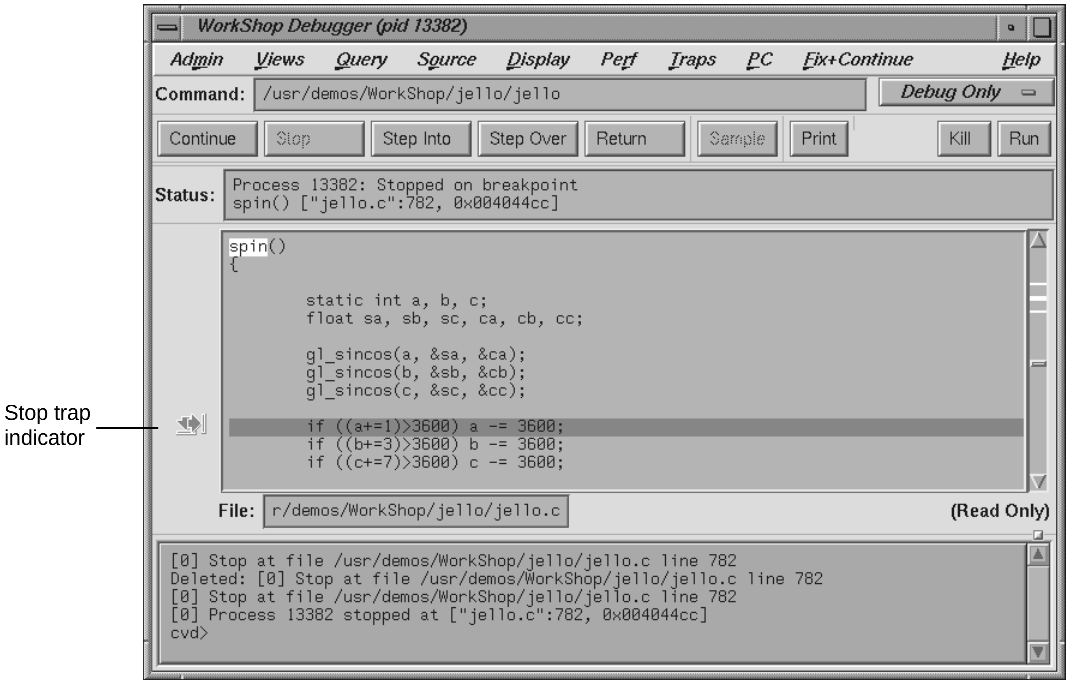
3.3 Setting Traps

Stop traps (also called breakpoints) stop program execution at a specified line in the code allowing you to track the progress of your program and to check the values of variables at that point. Typically, you set breakpoints in your program prior to running it under the Debugger. For more information on traps, refer to Chapter 4, page 39.

In this part of the tutorial, you set a breakpoint at the `spin` function.

1. Click the left mouse button in the source code annotation column next to the line containing `if ((a+=1)>3600) a -= 3600;`.

   A stop trap indicator appears in the annotation column as shown in Figure 12, page 28. This stop trap halts execution of `jello` at the beginning of the next call to the `spin` function. When the process stops, an icon indicating the current program counter (PC) appears and the line becomes highlighted.


Stop trap indicator
WorkShop Debugger (pid 13382)
Admin Views Query Source Display Perf Traps PC Fix+Continue Help
Command: /usr/demos/WorkShop/jello/jello
Debug Only
Continue Stop Step Into Step Over Return Sample Print Kill Run
Status: Process 13382: Stopped on breakpoint
spin() ["jello.c":782, 0x004044cc]
File: r/demos/WorkShop/jello/jello.c
(Read Only)

```
spin()
{

        static int a, b, c;
        float sa, sb, sc, ca, cb, cc;

        gl_sincos(a, &sa, &ca);
        gl_sincos(b, &sb, &cb);
        gl_sincos(c, &sc, &cc);

        if ((a+=1)>3600) a -= 3600;
        if ((b+=3)>3600) b -= 3600;
        if ((c+=7)>3600) c -= 3600;
```

```
[0] Stop at file /usr/demos/WorkShop/jello/jello.c line 782
Deleted: [0] Stop at file /usr/demos/WorkShop/jello/jello.c line 782
[0] Stop at file /usr/demos/WorkShop/jello/jello.c line 782
[0] Process 13382 stopped at ["jello.c":782, 0x004044cc]
cvd>
```

Figure 12. Stop Trap Indicator

2. Click the `Continue` button at the upper-left corner of the Main View several times so that `jello` goes through several iterations.

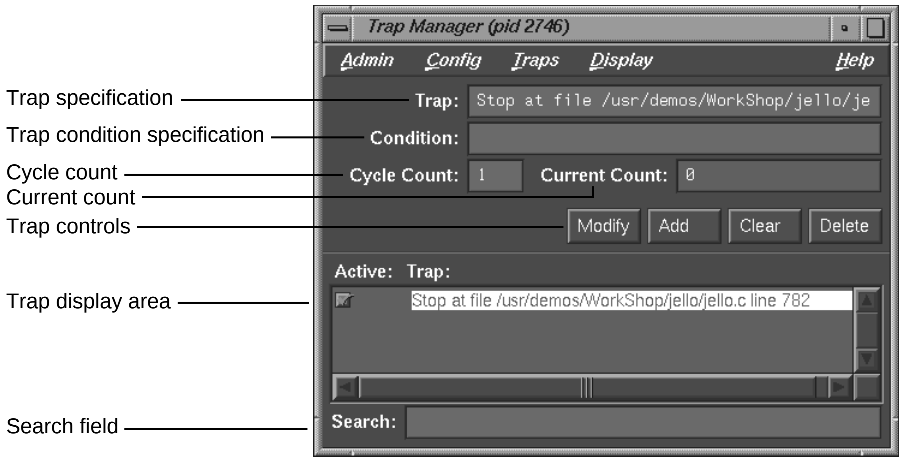
   The `Continue` button resumes execution until the next breakpoint (in this case, `spin`) is encountered. Stopping at the `spin` function allows you to view the `jello` image one frame at a time.

3. Select `Trap Manager` from the `Views` menu in Main View.

   The `Trap Manager` window appears as shown in Figure 13, page 29.

   The Trap Manager lets you list, add, edit, disable, or remove traps in a process. You set a breakpoint in the `spin` function by clicking in the source code annotation column. The trap is displayed in the trap display area.

   You can define other traps as well in the Trap Manager. You set conditional traps in the `Condition` field near the top of the window. The count information lets you specify the number of times a trap should be encountered before it activates. Trap controls let you manipulate traps. All traps (active and inactive) are shown in the trap display area.



```
Trap: Stop at file /usr/demos/WorkShop/jello/je
Cycle Count: 1   Current Count: 0
Active: Trap:
Stop at file /usr/demos/WorkShop/jello/jello.c line 782
```

Figure 13. Trap Manager Window

4. Click the button to the left of the stop trap in the trap display area.

   The trap is temporarily disabled. Trap Manager lets you turn traps on and off by clicking them.

5. Click the `Clear` button, move the cursor to the `Trap`: field, then type

   `watch display_mode`

   and click `Add.`

   This sets a watchpoint for the variable *display_mode*. A watchpoint is a trap that fires when a specified variable or address is read, written, or executed.

   After you continue the process, you can fire this watchpoint by holding down the right mouse button in the `jello` window and selecting a different display option from the `Display` menu. The variable *display_mode* is accessed and the watchpoint fires.

6. Click the `Continue` button to restart the process.

   The process now runs somewhat slower but still at a reasonable speed for debugging.

7. Hold down the right mouse button in the `jello` window to display the pop-up menu. From this menu, select `Display` and then select the `conecs` option with the right mouse button.

   This triggers the watchpoint and stops the process.

8. Go to the Trap Manager window and click the button next to the `display_mode` watchpoint to deactivate it. Click the button next to the `spin` stop trap to reactivate it.

   This resets the traps for use in this tutorial.

9. Enter **`100`** in the `Cycle Count` field, press `Enter`, and click the `Continue` button in Main View.

   This takes the process through the stop trap for the specified number of times, provided no other interruptions occur. The `Current Count` field keeps track of the actual number of iterations since the last stop, which is useful if an interrupt occurs. Note that it updates at interrupts only.

10. Select `Close` from the `Admin` menu in Trap Manager to close it.

3.4 Examining Data

This part of the tutorial describes how to examine data after the process stops.

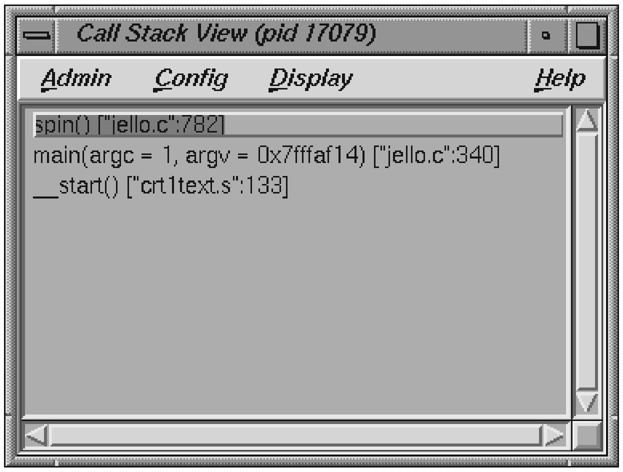
1. Select `Call Stack` from the `Views` menu in Main View.

   The `Call Stack View` window appears as in Figure 14, page 31. The `Call Stack View` window shows each frame in the call stack at the time of the breakpoint with the calling parameters and their values. You can also display the calling parameters' types, locations, and PC (program counter) through the `Display` menu. For more information, see Section 6.1, page 57.

   In this example, the `spin` and `main` stack frames are displayed in `Call Stack View`, and the `spin` stack frame is highlighted, indicating that it is the current stack frame.

2. Pull down the `Admin` menu and choose the `Active` selection.

   By default, the `Active` toggle button in the `Admin` menu is turned on. Active views are those that have been specified to change their contents at stops or at call stack context changes. If the toggle is on, the call stack is updated automatically whenever the process stops.



```
spin() ["jello.c":782]
main(argc = 1, argv = 0x7fffaf14) ["jello.c":340]
__start() ["crt1text.s":133]
```

Figure 14. `Call Stack View` at `spin` Stop Trap

3. Double-click the `main` stack frame.

   This shifts the stack frame to the `main` function, scrolls the source code in Main View (or `Source View`) to the place in `main` where `spin` was called, and highlights the call in the designated context color. Any active views are updated according to the new stack frame.

4. Double-click the `spin` stack frame.

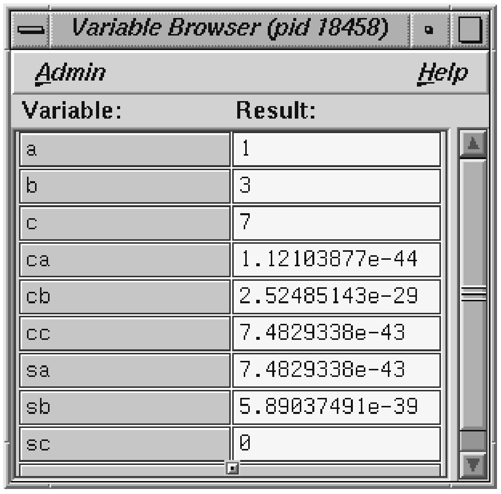
   This returns the stack frame to the `spin` function.

   Select `Variable Browser` from the `Views` menu in Main View.

   The `Variable Browser` window appears. This window shows you the value of local variables at the breakpoint. The variables appear in the left column (read-only), and the corresponding values appear in the right column (editable). Since the right column is editable, you can change the values of the variables if you want.

   Your `Variable Browser` window should resemble the one in Figure 15, page 32, although you may need to enlarge the window to see all the variables (the values will be different).

   The `jello` program uses variables *a*, *b*, and *c* as angles (in tenths); *ca, cb, cc* as their corresponding cosines; and *sa, sb, sc* as their sines. Whenever you stop at `spin`, these values change.

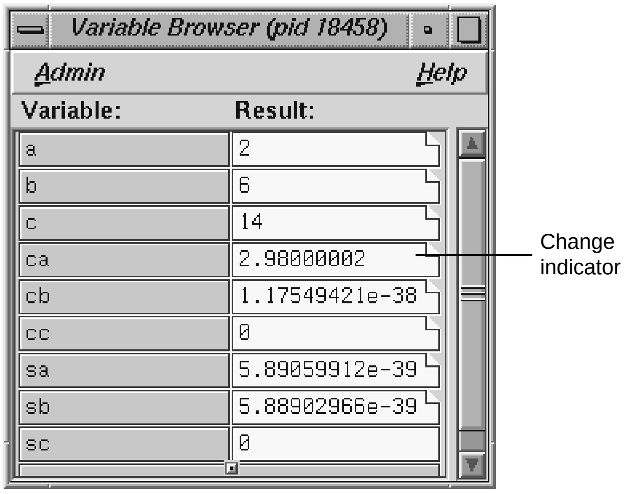


Variable:	Result:
a	1
b	3
c	7
ca	1.12103877e-44
cb	2.52485143e-29
cc	7.4829338e-43
sa	7.4829338e-43
sb	5.89037491e-39
sc	0

Figure 15. `Variable Browser` at `spin`

5. Double-click some different frames in `Call Stack View` and observe the changes to `Variable Browser` and Main View.

   These views update appropriately whenever you change frames in `Call Stack View`. Notice also the change indicators in the upper-right corners of the `Result` fields (see Figure 16, page 33). These appear if the value has changed. If you click the *folded* corner, the previous value displays (and the indicator appears *unfolded*). You can then toggle back to the current value.



Variable:	Result:
a	2
b	6
c	14
ca	2.98000002
cb	1.17549421e-38
cc	0
sa	5.89059912e-39
sb	5.88902966e-39
sc	0

Figure 16. `Variable Browser` after Changes

6. Select `Close` from the `Admin` menu in `Variable Browser` and `Close` from the `Admin` menu in `Call Stack View` to close them.

7. Select `Expression View` from the `Views` menu in Main View.

   The `Expression View` window appears. It lets you evaluate an expression involving data from the process. The expression can be typed in or more conveniently cut and pasted from your source code. You can view the value of variables (or expressions involving variables) any time the process stops. Enter the expression in the left column, and the corresponding value appears in the right column. For more information, see Section 6.2, page 60.

8. Hold down the right mouse button in the `Expression` column to bring up the `Language` menu. Then hold down the right mouse button in the `Result` column to display the `Format` menu.

   The `Language` menu (shown on the left side of Figure 17, page 34) lets you apply the language semantics to the expression.

   The `Format` menu (shown on the right side of Figure 17, page 34) lets you view the value, type, address, or size of the result. You can further specify the display format for the value and address.

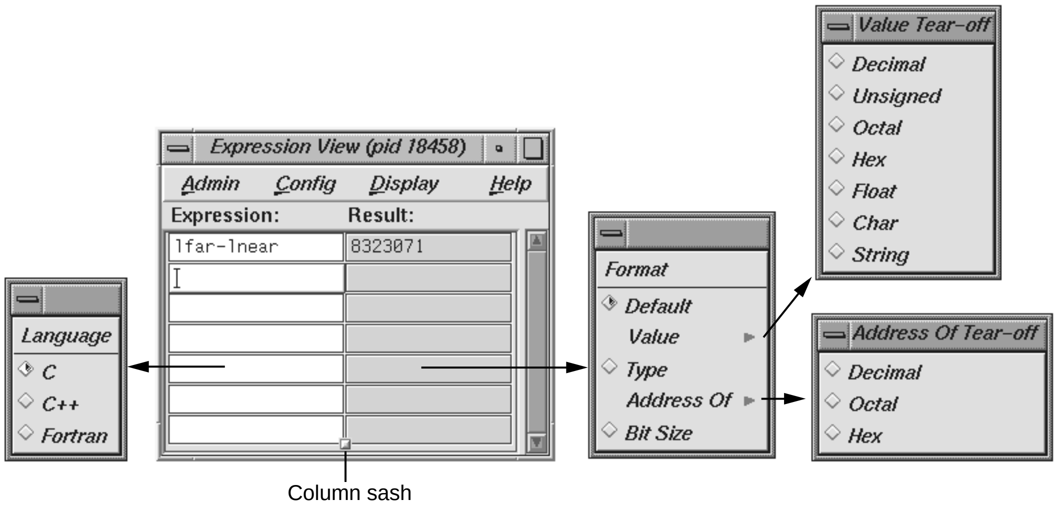


Figure 17. `Expression View` with `Language` and `Format` Menus Displayed

9. Click on the first `Expression` field in the `Expression View` window. Then enter **`(a+1)>3600`** in the field and press `Enter`.

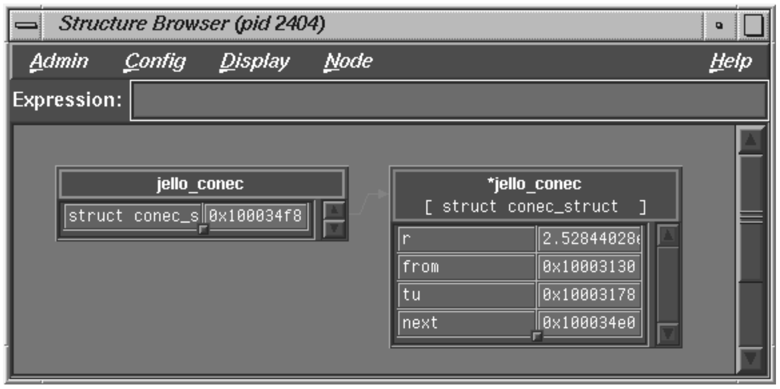
   This is a test performed in `jello` to ensure that the value of *a* is less than 3600. This uses the variable *a* that was displayed previously in `Variable Browser`. After you press `Enter`, the result is displayed in the right column; 0 signifies false.

10. Select `Close` from the `Admin` menu in `Expression View` to close it.

11. Select `Structure Browser` from the `Views` menu in Main View.

12. Enter **`jello_conec`** in the `Expression` field and press `Enter`.

    The `Structure Browser` window displays the structure for the given expression; field names are displayed in the left column, and values in the right column. If only pointers are available, the Structure Browser will dereference the pointers automatically until actual values are encountered. You can then perform any further dereferencing by double-clicking pointer addresses in the right column of the data structure objects. A window similar to the one shown in Figure 18, page 35, now appears.

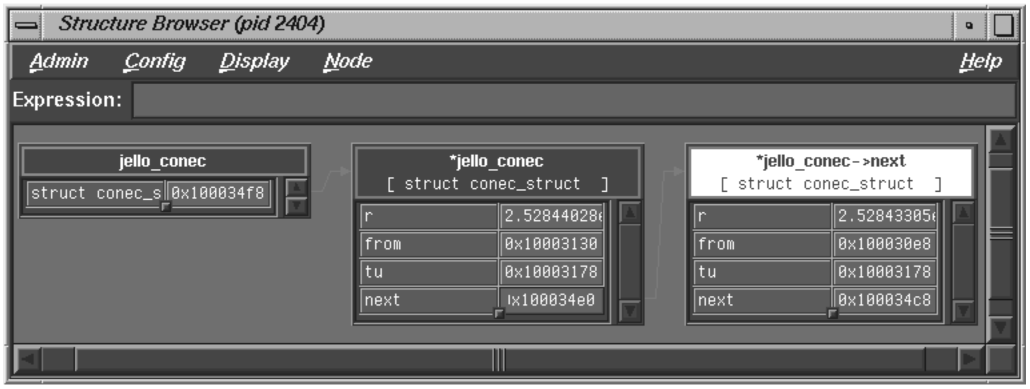


```
[ struct conec_struct  ]
struct conec_s  0x100034f8
r      2.52844028e
from   0x10003130
tu     0x10003178
next   0x100034e0
```

Figure 18. `Structure Browser` Window with `jello_conec` Structure

13. Click once to focus, then double-click the address of the `next` field (in the right column of the `jello_conec` structure).

    Double-clicking the address corresponding to a pointer field dereferences it. Double-clicking the field name displays the complete name of the field in the `Expression` field at the top of the `Structure Browser` window. (See Figure 19, page 35.)



```
struct conec_s  0x100034f8

[ struct conec_struct  ]
r      2.52844028e
from   0x10003130
tu     0x10003178
next   0x100034e0

[ struct conec_struct  ]
r      2.52843305e
from   0x100030e8
tu     0x10003178
next   0x100034c8
```

Figure 19. `Structure Browser` Window With `Next` Pointer Dereferenced

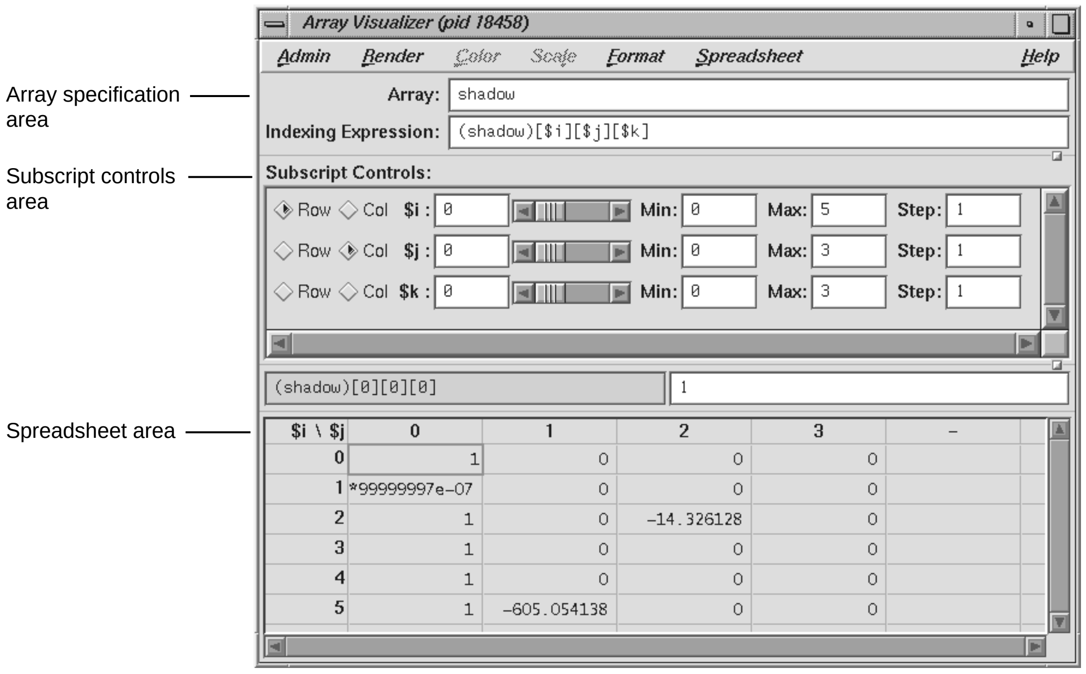
14. Select `Close` from the `Admin` menu of `Structure Browser` window to close it.

15. Select `Array Browser` from the `Views` menu in Main View.

    The Array Browser lets you see or change values in an array variable. It is particularly valuable for finding bad data in an array or for testing the effects of values you enter.

16. Type **shadow** in the `Array` field and press `Enter`.

    You can now see the values of the `shadow` matrix, which displays the polyhedron's shadow on the cube. The Array Browser template should resemble Figure 20, page 36, but with different data values. If any fields are hidden, you can drag the sash buttons at the right of the window to expose them.



Array: shadow

Indexing Expression: (shadow)[$i][$j][$k]

[✓ Row] Col	$i : 0	Min: 0	Max: 5	Step: 1
Row [✓ Col]	$j : 0	Min: 0	Max: 3	Step: 1
Row Col	$k : 0	Min: 0	Max: 3	Step: 1

(shadow)[0][0][0] 1

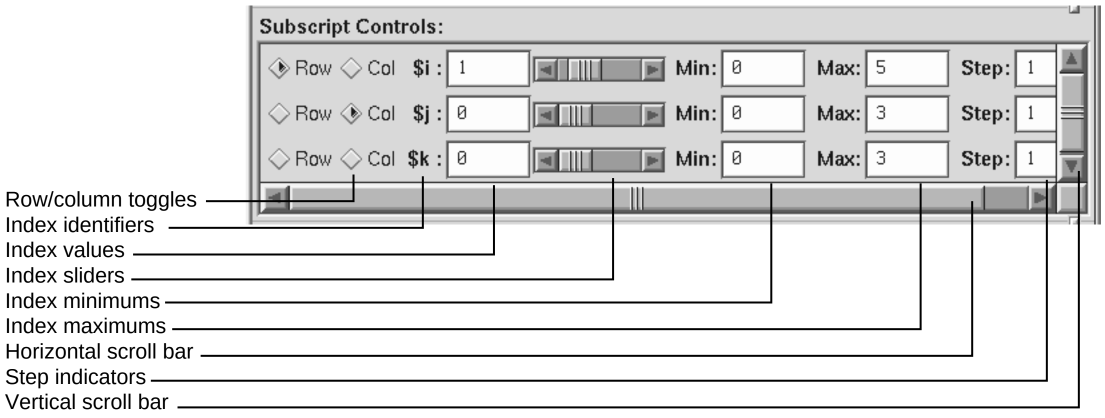
$i \ $j	0	1	2	3	–
0	1	0	0	0	
1	*99999997e-07	0	0	0	
2	1	0	-14.326128	0	
3	1	0	0	0	
4	1	0	0	0	
5	1	-605.054138	0	0	

Figure 20. `Array Visualizer` Window for `shadow` Matrix

17. Select the `Col` button next to the `$k` index.

    The Array Browser can handle matrices containing up to six dimensions but displays only two dimensions at a time. Selecting the `Col` button for `$k` has the effect of switching from a display of `$i` by `$j` to a display of `$i` by `$k`.

Figure 21, page 37, shows a close-up view of the subscript control area.



Subscript Controls:

◆ Row ◇ Col $i : 1 Min: 0 Max: 5 Step: 1

◇ Row ◆ Col $j : 0 Min: 0 Max: 3 Step: 1

◇ Row ◇ Col $k : 0 Min: 0 Max: 3 Step: 1

Figure 21. Subscript Control Area in `Array Visualizer` Window

The row and column toggles indicate whether a vector appears as a row, column, or not at all in the spreadsheet area. Although any number of vectors can reside in an array, you can view only two vectors at a time. The index values show the number of elements in a vector and are used to change the dimensions of the matrix. The index sliders let you move the focus cell along the particular vector. The index minimums and index maximums identify the beginning and ending elements, respectively, in the vectors. Use the horizontal and vertical scroll bars to expose hidden portions of the `Array Visualizer` window.

18. Select `Surface` from the `Render` menu.

    The `Render` menu displays the data from the selected array variable graphically, in this case as a three-dimensional surface. The selected cell is highlighted by a rectangular prism. The selected subscripts correspond to the x- and y-axes in the rendering with the corresponding value plotted on the z-axis. The data can be rendered as a surface, bar chart, multiple lines, or points.

    Select `Exit` from the `Admin` menu in Main View to end this tutorial.

Setting Traps [4]

Setting traps is one of the most important functions of a debugger or performance analyzer. A trap enables you to select a location or condition within your program at which you can stop the process or collect performance data automatically. In general, you set or clear traps from Main View or the Trap Manager. You can also specify traps in the Debugger command line at the bottom of the Main View. For signal traps, you can also use the `Signal Panel` window. For system call traps, use the `Syscall Panel` window.

When you are debugging a program, you typically set a trap in a process to determine if there is a problem at that point. WorkShop lets you inspect the call stack, examine variables, or perform other procedures to get information about the state of the process.

Traps are also useful for analyzing program performance. They let you collect data related to resource usage without stopping the process.

This chapter covers the following topics:

- Trap Terminology, Section 4.1, page 39
- Setting Traps in Main View and `Source View`, Section 4.2, page 40
- Setting Traps in Trap Manager, Section 4.3, page 43
- Setting Traps With `Signal Panel` and `Syscall Panel`, Section 4.4, page 48

For a tutorial on the use of traps, see Section 3.3, page 27.

4.1 Trap Terminology

In WorkShop, the term trap refers to any intentional process interruption. A trap has two dimensions: the *trigger*, which specifies when the trap fires; and the *action*, which is what happens when the trap fires. A trap can either stop a process or capture data about a process.

4.1.1 Trap Triggers

You can set traps at a specified location or when a specified event occurs. You can set a trigger at any of the following points:

- At a given line in a file (traditionally referred to as a *breakpoint*)
- At a given instruction address
- At the entry or exit for a given function
- After set time intervals (referred to as a *pollpoint*)
- When a given variable or address is read, written, or executed (referred to as a *watchpoint*)
- When a given signal is received
- When a given system call is entered or exited

In addition, you can specify a condition (as an expression) that must be met before a trap fires. You can also specify a *cycle count*, which specifies the number of passes through a trap before firing it.

4.1.2 Trap Actions

Two actions can occur when a trap is fired:

- One or all processes can stop by using a *stop trap*. In single process debugging, a stop trap stops the current process. In multiprocess debugging, you can specify the stop trap to stop all processes or the current process only.
- Sample performance data can be taken by using a *sample trap*. Sample traps are used only in performance analysis, not directly in debugging. They collect data without stopping the process. You can specify sample traps to collect such information as call stack data, function counts, basic block counts, PC profile counts, `mallocs/frees`, system calls, and page faults. Sample traps can use any of the triggers that stop traps use. Sample traps are often set up as pollpoints so that they collect data at set time intervals.

4.2 Setting Traps in Main View and `Source View`

You can set traps directly in Main View by using the `Traps` menu or by clicking the mouse in the source annotation column. You can also specify traps in the Debugger command line.

4.2.1 Setting Traps with the Traps Menu in Main View

The `Traps` menu in Main View is shown in Figure 22, page 41.

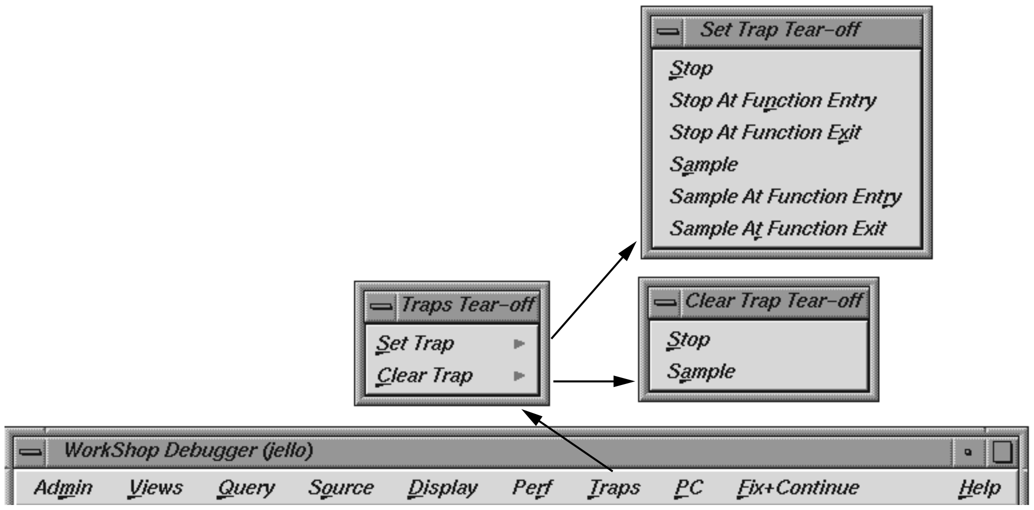


Figure 22. `Traps` Menu in Main View

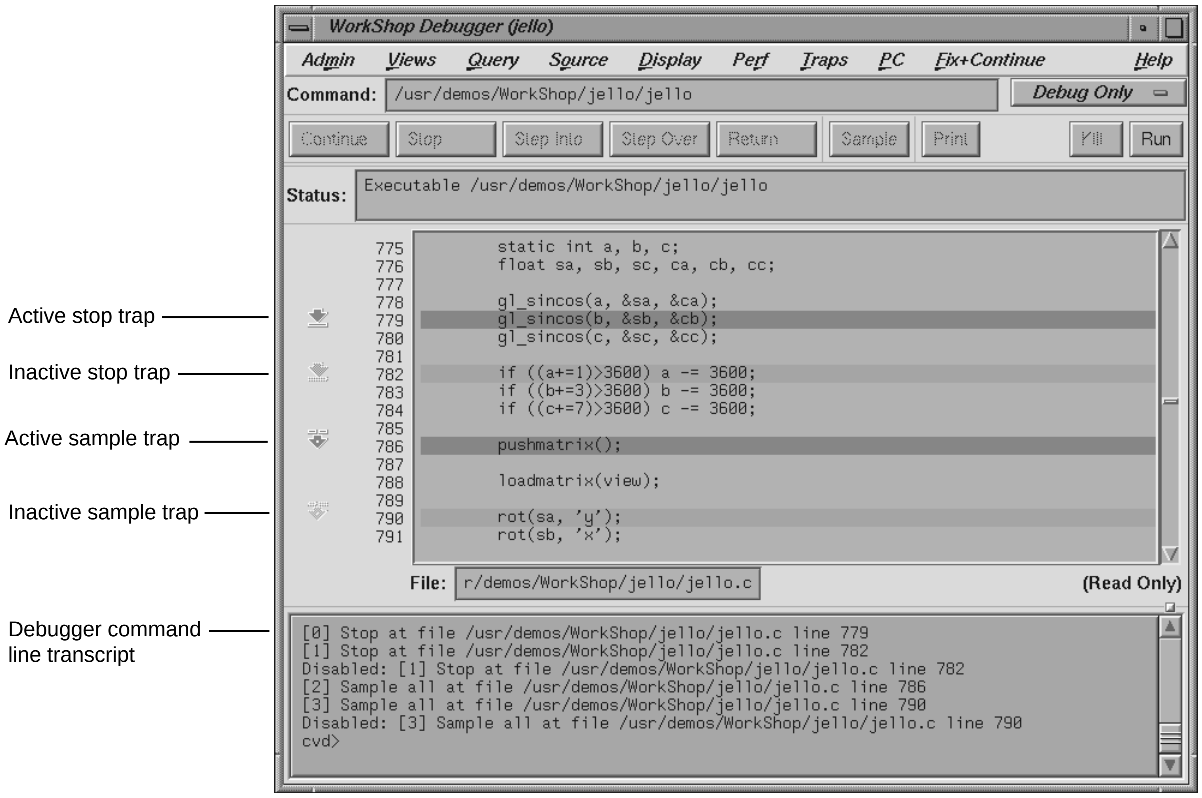
To set a trap using the `Traps` menu, you need to identify where trap location and trap type.

To set a stop trap at a line displayed in Main View (or `Source View`), click the cursor in the source annotation column next to the appropriate line in the source code and select `Set Trap`, then select `Stop` or `Sample`.

For a trap at the beginning or end of a function, highlight the function name in the source code display area and select `Set Trap`, then `Stop At Function Entry`, `Stop At Function Exit`, `Sample At Function Entry`, or `Sample At Function Exit`, as appropriate.

Traps are indicated by icons in the source annotation column (and also appear in `Trap Manager` window if you have it open). Figure 23, page 42, shows some typical trap icons. Sampling is indicated by a dot in the center of the icon. Traps appear in normal color or grayed out, depending on whether they are active or inactive. A transcript of the trap activity appears in the Debugger command line area. The active/inactive nature of traps is discussed in Section 4.3.6, page 48.

The `Clear Trap` selection in the `Traps` menu deletes the trap on the line containing the cursor. You must designate a `Stop` or `Sample` trap type, since both types can exist at the same location appearing superimposed on each other.


Active stop trap
Inactive stop trap
Active sample trap
Inactive sample trap
Debugger command line transcript
WorkShop Debugger (jello)
Admin Views Query Source Display Perf Traps PC Fix+Continue Help
Debug Only
Continue Stop Step Into Step Over Return Sample Print Kill Run
(Read Only)

Command: /usr/demos/WorkShop/jello/jello

Status: Executable /usr/demos/WorkShop/jello/jello

```
775         static int a, b, c;
776         float sa, sb, sc, ca, cb, cc;
777
778         gl_sincos(a, &sa, &ca);
779         gl_sincos(b, &sb, &cb);
780         gl_sincos(c, &sc, &cc);
781
782         if ((a+=1)>3600) a -= 3600;
783         if ((b+=3)>3600) b -= 3600;
784         if ((c+=7)>3600) c -= 3600;
785
786         pushmatrix();
787
788         loadmatrix(view);
789
790         rot(sa, 'y');
791         rot(sb, 'x');
```

File: r/demos/WorkShop/jello/jello.c

```
[0] Stop at file /usr/demos/WorkShop/jello/jello.c line 779
[1] Stop at file /usr/demos/WorkShop/jello/jello.c line 782
Disabled: [1] Stop at file /usr/demos/WorkShop/jello/jello.c line 782
[2] Sample all at file /usr/demos/WorkShop/jello/jello.c line 786
[3] Sample all at file /usr/demos/WorkShop/jello/jello.c line 790
Disabled: [3] Sample all at file /usr/demos/WorkShop/jello/jello.c line 790
cvd>
```

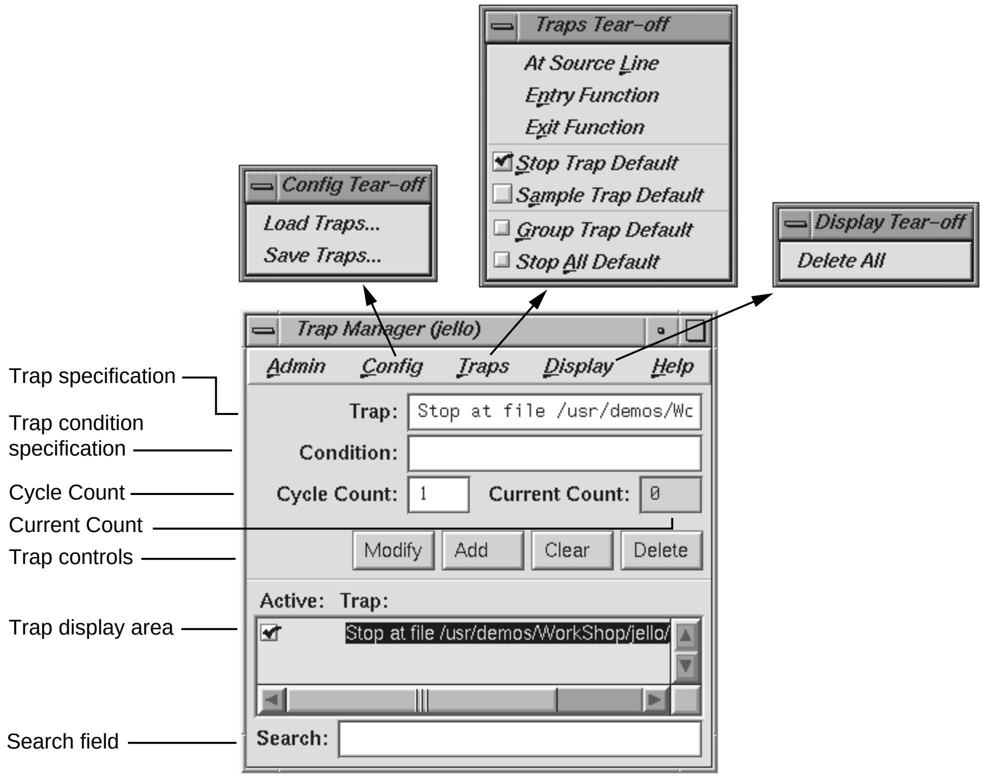
Figure 23. Typical Trap Icons

4.2.2 Setting Traps with the Mouse

The quickest way to set a trap is to click in the source annotation column in Main View or `Source View`. A subsequent click removes the trap. When the trap is set, an icon appears representing the trap. If data collection mode has been specified in the `Performance Data` window, clicking produces a sample trap; otherwise, a stop trap is entered. (To determine if data collection is on, look at the upper-right corner of the Debugger Main View to see which debugging option is selected (`Debug Only`, `Performance`, or `Purify`).

4.3 Setting Traps in Trap Manager

The Trap Manager helps you manage all traps in a process. The Trap Manager's two major functions are to list all traps in the process (except signals) and to let you add, delete, modify, or disable traps. The Trap Manager appears in Figure 24, page 43 with the `Config`, `Traps`, and `Display` menus shown.



Trap: Stop at file /usr/demos/Wc
Condition:
Cycle Count: 1 Current Count: 0
Stop at file /usr/demos/WorkShop/jello/

Figure 24. Trap Manager `Config`, `Traps`, and `Display` Menus

4.3.1 Setting Single-process and Multiprocess Traps

New or modified traps are entered in the `Trap:` field. Traps have the following general form:

```
[stop | sample] [all] [pgrp] location | condition
```

The entry [`stop | sample`] refers to the trap action. You can set a default for the action by using the `Stop Trap Default` or `Sample Trap Default` selections of the `Traps` menu and omitting it on the command line.

The entries [`all`] and [`pgrp`] are used in multiprocess analysis. The [`all`] entry causes all processes in the process group to stop or sample when the trap fires. The [`pgrp`] entry sets the trap in all processes within the process group that contains the code where the trap is set. You can set a default for the action by setting the `Stop All Default` or `Group Trap Default` toggles in the `Traps` menu.

Sample syntax for the *location* and *condition* are shown below.

```
[stop | sample] [all] [pgrp] at [file \filename]
   [line line-number]
```

Sets trap at the specified line in the specified file.

```
[stop | sample] [all] [pgrp] addr instruction-address
```

Sets trap on the specified instruction address.

```
[stop | sample] [all] [pgrp] entry function [[file] \
   filename]
```

```
[stop | sample] [all] [pgrp] in function [[file] \
    filename]
```

Sets trap on entry to the specified function. If the filename is given, the function is assumed to be in that file's scope.

```
[stop | sample] [all] [pgrp] exit function [[file] \
   filename]
```

Sets trap on exit from the specified function. If the filename is given, the function is assumed to be in that file's scope.

```
[stop | sample] [all] [ pgrp] watch expression \ [
   [for] read | write | execute [access]]
```

Sets a watchpoint on the specified expression (using the address and size of the expression for the watchpoint span). The watchpoint may be specified to fire on write, read, or execution (or some combination thereof). If not specified, the write condition is assumed.

```
[stop | sample] [all] [pgrp] watch addr[ess] address \
   [[size] size] [for] read | write | execute \
   [access]
```

Sets a watchpoint for the specified address and size in bytes. The watchpoint may be specified to fire on `write`, `read`, or `execute` (or some combination thereof) of memory in the given span. If not specified, the size defaults to 4 bytes.

```
[stop | sample] [all] [pgrp] signal signal-name
```

Sets a trap upon receipt of the given signal. Same as the `dbx`(1) `catch` subcommand.

```
[stop | sample] [all] [pgrp] syscall entry sys-call-name
```

Sets a trap on entry to the specified system call. This is slightly different from setting a trap on entry to the function by the same name. A syscall entry trap sets a trap on entry to the actual system call. A function entry trap sets a trap on entry to the stub function that calls the system call.

```
[stop | sample] [all] [pgrp] syscall exit sys-call-name
```

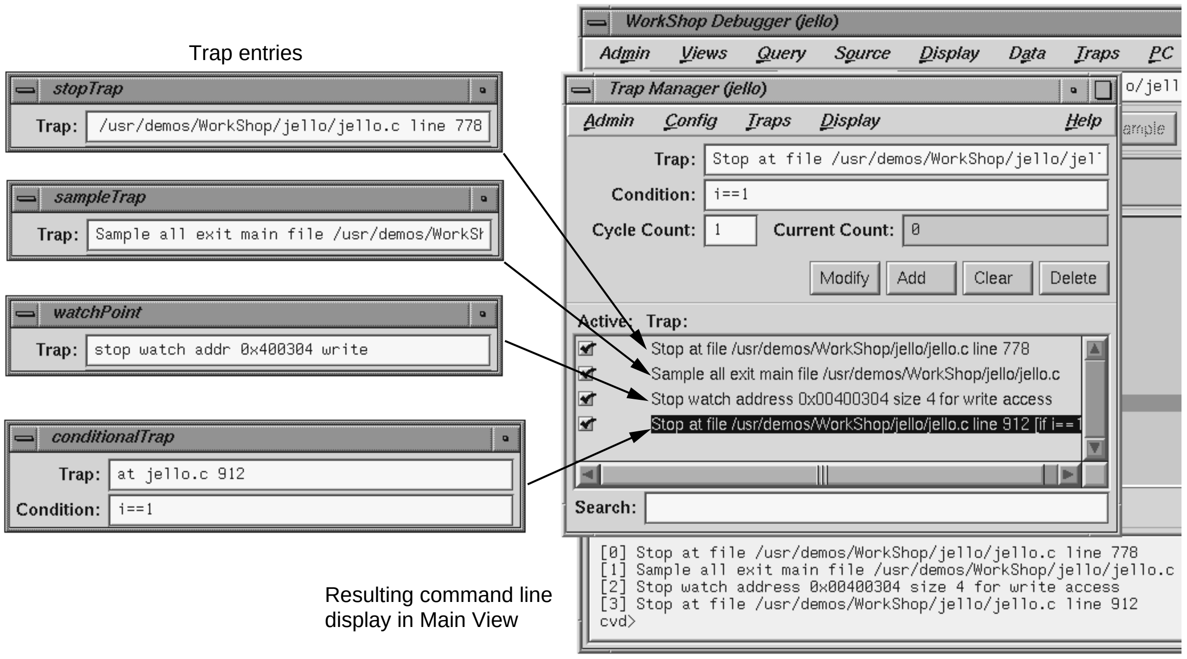
Sets a trap on exit from the specified system call. This is slightly different from setting a trap on exit from the function by the same name. A syscall exit trap sets a trap on exit from the actual system call. A function exit trap sets a trap on exit from the stub function that calls the system call.

```
[stop | sample] pollpoint [interval] time [seconds]
```

Sets a trap at regular intervals of seconds. This is typically used only for sampling.

After you enter the trap (by using the `Add` or `Modify` button or by pressing `Enter`), the full syntax of the specification appears in the field. The `Clear` button clears the `Trap` and `Condition` fields and the cycle fields.

Some typical trap examples are provided in Figure 25, page 46. The entries made in the `Trap` field are shown in the left portion of the figure, the trap display in Trap Manager resulting from these entries is shown on the right, and the trap display shown at the command line in Main View is shown at the bottom.


Trap entries
stopTrap
sampleTrap
watchPoint
conditionalTrap
WorkShop Debugger (jello)
Admin Views Query Source Display Data Traps PC
Trap Manager (jello)
Admin Config Traps Display Help
Modify Add Clear Delete
Active: Trap:
Search:
Resulting command line display in Main View

stopTrap — Trap: /usr/demos/WorkShop/jello/jello.c line 778

sampleTrap — Trap: Sample all exit main file /usr/demos/WorkSh

watchPoint — Trap: stop watch addr 0x400304 write

conditionalTrap — Trap: at jello.c 912 — Condition: i==1

Trap Manager (jello) — Trap: Stop at file /usr/demos/WorkShop/jello/jel — Condition: i==1 — Cycle Count: 1 — Current Count: 0

- ✓ Stop at file /usr/demos/WorkShop/jello/jello.c line 778
- ✓ Sample all exit main file /usr/demos/WorkShop/jello/jello.c
- ✓ Stop watch address 0x00400304 size 4 for write access
- ✓ Stop at file /usr/demos/WorkShop/jello/jello.c line 912 [if i==1

```
[0] Stop at file /usr/demos/WorkShop/jello/jello.c line 778
[1] Sample all exit main file /usr/demos/WorkShop/jello/jello.c
[2] Stop watch address 0x00400304 size 4 for write access
[3] Stop at file /usr/demos/WorkShop/jello/jello.c line 912
cvd>
```

Figure 25. Trap Examples

4.3.2 Setting a Trap Condition

The `Condition:` field lets you specify the condition necessary for the trap to be fired. A condition can be any legal expression and is considered to be true if it returns a nonzero value when the corresponding trap is encountered. The expression must be valid in the context in which it will be evaluated. For example, a Fortran condition like **`a .gt .2`** cannot be evaluated if it is tested while the program is stopped in a C function.

There are two possible sequences for entering a trap with a condition:

1. Define the trap.
2. Define the condition.
3. Click `Add`.

and

1. Define the trap.
2. Click `Add`.
3. Define the condition.
4. Click `Modify` or press `Enter`.

An example of a trap with a condition is shown in Figure 25, page 46. The expression `i==1` has been entered in the `Condition`: field. (If you were debugging in Fortran, you would use the Fortran syntax, for example, `i .eq .1`.) After the trap has been entered, the condition appears as part of the trap definition in the display area. During execution, any requirements set by the trigger must be satisfied first for the condition to be tested. A condition is true if the expression (valid in the language of the program you are debugging) evaluates to a nonzero value.

4.3.3 Setting a Trap Cycle Count

The `Cycle Count` field lets you pass through a trap a specific number of times without firing. If you set a cycle count of *n*, the trap will fire the *n*th time the trap is encountered and every *n*th iterations thereafter. The `Current Count` field indicates the number of times the process has passed the trap since either the cycle count was set or the trap last fired. The current count updates only when the process stops.

4.3.4 Setting a Trap with the Traps Menu

The `Traps` menu of the Trap Manager lets you specify traps in conjunction with Main View or `Source View`. Clicking `At Source Line` sets a trap at the line in the source display area containing the current selection. To set a trap at the beginning or end of a function, select the function name in the source display and click `Entry Function` or `Exit Function`.

4.3.5 Moving around the Trap Display Area

The trap display area displays all traps set for the current process. There are vertical and horizontal scroll bars for moving around the display area. The `Search` field lets you incrementally search for any string in any trap.

4.3.6 Enabling and Disabling Traps

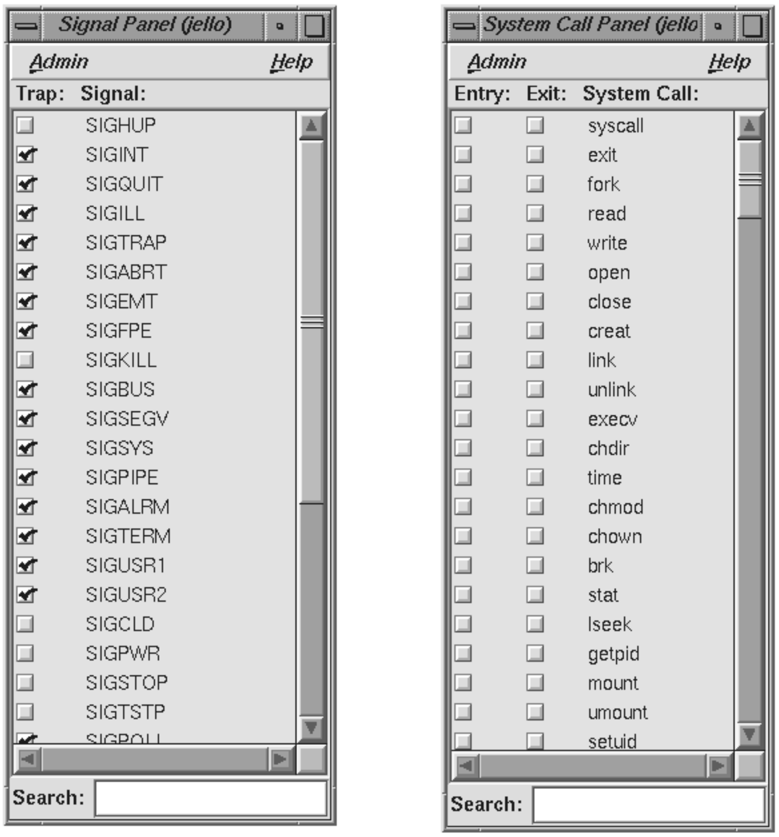
Each trap has an indicator to its left for toggling back and forth between active and inactive trap states. This feature lets you accumulate traps and turn them on only as needed. Thus, when you do not need the trap, it will not be in your way. When you do need it, it is readily reenabled.

4.3.7 Saving and Reusing Trap Sets

The `Load Traps...` selection in the `Config` menu lets you bring in previously saved trap sets. This is useful for reestablishing a set of traps between debugging sessions. The `Save Traps...` selection of the `Config` menu lets you save the current traps to a file.

4.4 Setting Traps With `Signal Panel` and `System Call Panel`

You can trap signals by using the `Signal Panel` and system calls using `System Call Panel` (see Figure 26, page 49).



Trap:	Signal:
☐	SIGHUP
✓	SIGINT
✓	SIGQUIT
✓	SIGILL
✓	SIGTRAP
✓	SIGABRT
✓	SIGEMT
✓	SIGFPE
☐	SIGKILL
✓	SIGBUS
✓	SIGSEGV
✓	SIGSYS
✓	SIGPIPE
✓	SIGALRM
✓	SIGTERM
✓	SIGUSR1
✓	SIGUSR2
☐	SIGCLD
☐	SIGPWR
☐	SIGSTOP
☐	SIGTSTP
✓	SIGPOLL

Entry:	Exit:	System Call:
☐	☐	syscall
☐	☐	exit
☐	☐	fork
☐	☐	read
☐	☐	write
☐	☐	open
☐	☐	close
☐	☐	creat
☐	☐	link
☐	☐	unlink
☐	☐	execv
☐	☐	chdir
☐	☐	time
☐	☐	chmod
☐	☐	chown
☐	☐	brk
☐	☐	stat
☐	☐	lseek
☐	☐	getpid
☐	☐	mount
☐	☐	umount
☐	☐	setuid

Figure 26. `Signal Panel` and `System Call` Panel

You can select either panel from the `Views` menu in Main View. The `Signal Panel` sets a trap on receipt of the signal(s) selected. The `System Call Panel` sets a trap at the selected entry to or return from the system call.

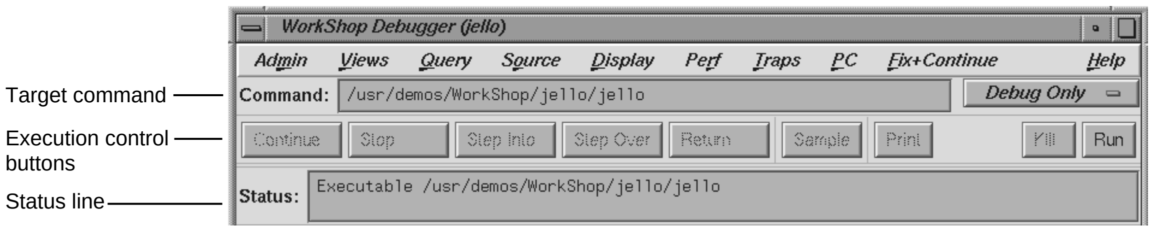
Controlling Process Execution [5]

This chapter tells you how to control process execution. It includes the following topics:

- Main View Control Panel, Section 5.1, page 51
- Controlling Process Execution Using the `PC` Menu, Section 5.2, page 55
- `Execution View`, Section 5.3, page 55

5.1 Main View Control Panel

Process execution is controlled by using the top portion of the Main View window. See Figure 27, page 51.



```
Command: /usr/demos/WorkShop/jello/jello
Status: Executable /usr/demos/WorkShop/jello/jello
```

Figure 27. Main View Control Panel

The Main View window contains a row of execution control buttons that enable you to control program execution. These buttons are located above the Source View area. To activate a button, click on it by using the left mouse button. The Main View control panel is described below.

5.1.1 Status and Entry Fields in the Main View Control Panel

The control panel contains the following fields:

`Command`	Lets you enter the command for running the process with any argument(s).
`Status`	Displays information about the execution status of the program you are debugging. The top line

in this box tells you whether the program is running or stopped. The next line lists the current call stack frame, if applicable. (To see all of the stack frames, open the `Call Stack View` from the `Views` menu.)

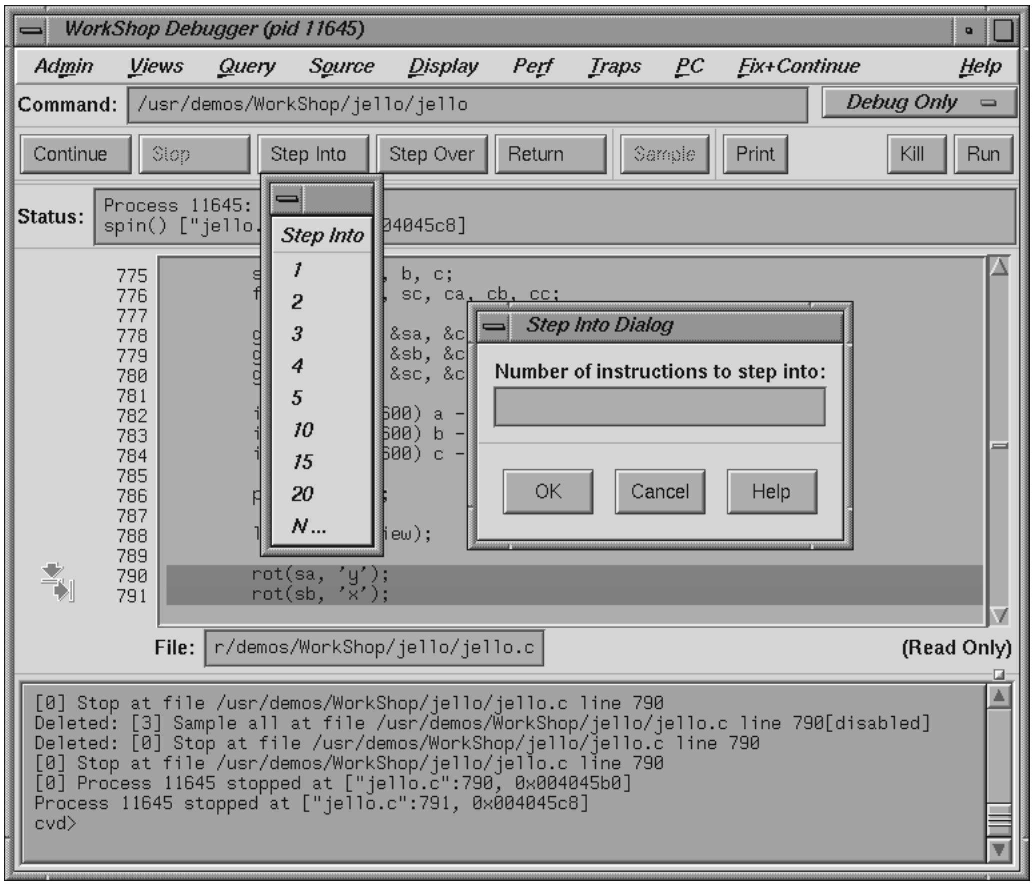
5.1.2 Execution Control Buttons

The execution control buttons enable you to control program execution. The two control buttons for starting and terminating a process are:

`Run`	Creates a new process for the program and starts execution. It is also used to rerun a program.
`Kill`	Kills the active process.

The following control buttons are used for process interruptions:

`Continue`	Resumes program execution after a halt and continues until a breakpoint or other event stops execution.
`Stop`	Stops execution of the program. When program execution stops, the current source line is highlighted in the Main View and annotated with an arrow indicating the program counter (PC).
`Step Into`	Steps to the next source line and into function calls. To step a specific number of lines, hold down the right mouse button over the `Step Into` button. This displays the pop-up menu shown in Figure 28, page 53. You can select one of the fixed values or enter your own number of steps by selecting `N...`. Selecting `N...` displays the dialog box shown at the right in Figure 28, page 53.



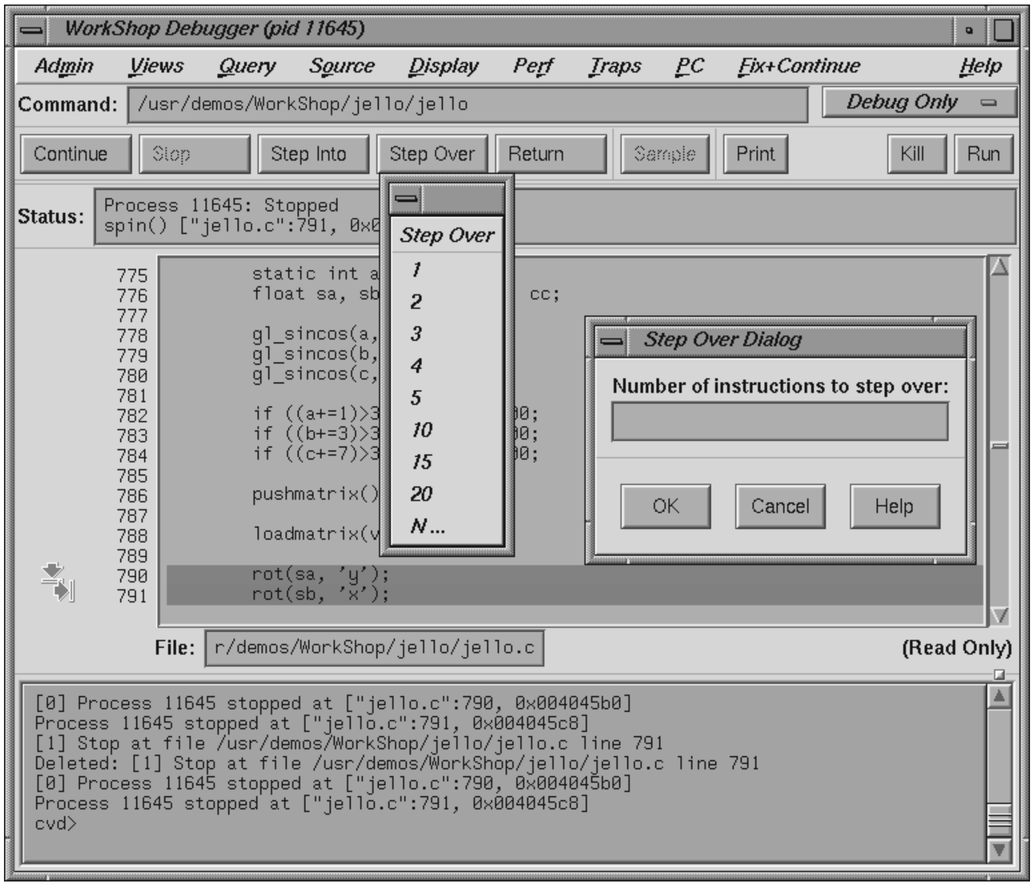
```
Process 11645:
spin() ["jello.                04045c8]
```

```
rot(sa, 'y');
rot(sb, 'x');
```

```
[0] Stop at file /usr/demos/WorkShop/jello/jello.c line 790
Deleted: [3] Sample all at file /usr/demos/WorkShop/jello/jello.c line 790[disabled]
Deleted: [0] Stop at file /usr/demos/WorkShop/jello/jello.c line 790
[0] Stop at file /usr/demos/WorkShop/jello/jello.c line 790
[0] Process 11645 stopped at ["jello.c":790, 0x004045b0]
Process 11645 stopped at ["jello.c":791, 0x004045c8]
cvd>
```

Figure 28. Pop-up Menu and `Step Into Dialog`

`Step Over`	Steps to the next source line and over function calls. To step a specific number of lines, hold down the right mouse button over the `Step Over` button. This displays the pop-up menu shown in Figure 29, page 54. You can select one of the fixed values or enter your own number of steps by selecting `N...`. Selecting `N...` displays the dialog box shown at the right in Figure 29, page 54.



Command: /usr/demos/WorkShop/jello/jello

Status:
```
Process 11645: Stopped
spin() ["jello.c":791, 0x0
```

```
        static int a
        float sa, sb                    cc;

        gl_sincos(a,
        gl_sincos(b,
        gl_sincos(c,

        if ((a+=1)>3                    0;
        if ((b+=3)>3                    0;
        if ((c+=7)>3                    0;

        pushmatrix()

        loadmatrix(v

        rot(sa, 'y');
        rot(sb, 'x');
```

Number of instructions to step over:

File: r/demos/WorkShop/jello/jello.c

```
[0] Process 11645 stopped at ["jello.c":790, 0x004045b0]
Process 11645 stopped at ["jello.c":791, 0x004045c8]
[1] Stop at file /usr/demos/WorkShop/jello/jello.c line 791
Deleted: [1] Stop at file /usr/demos/WorkShop/jello/jello.c line 791
[0] Process 11645 stopped at ["jello.c":790, 0x004045b0]
Process 11645 stopped at ["jello.c":791, 0x004045c8]
cvd>
```

Figure 29. Pop-up Menu and `Step Over Dialog`

`Return`	Executes the remaining instructions in the current function. Program execution stops upon return from that procedure.

There is one button in the control panel for spontaneous sampling:

`Sample`	Collects performance data when clicked. A performance task must have been previously

specified in the `Performance Task` window and data collection must have been enabled.

5.2 Controlling Process Execution Using the PC Menu

The PC (program counter) menu in Main View provides a quick and informal means of controlling process execution. Options let you manually control process execution without setting traps. The target location is determined by the location of the cursor in the source display area. There are two selections:

`Continue To`	Lets you select a target location in the current process (by placing the cursor in the line). The process proceeds from the current PC to that point (provided there are no interruptions) and stops there, as it would for a stop trap. `Continue To` is equivalent to setting a one-time trap. If the process is interrupted before reaching the target location, then the command is cancelled.
`Jump To`	Lets you select a target location in the current process (by placing the cursor in the line). The location must be in the same function. Instead of starting from the current PC, `Jump To` skips over any intervening code and restarts the process at the target. This is particularly useful if you want to get around bad code or irrelevant portions of the program. It also lets you back up and reexecute a portion of code.

5.3 Execution View

The `Execution View` window is a simple shell that lets you set environment variables and inspect error messages. Your target program I/O, if any, is displayed in the `Execution View` window. If the program is I/O-based, then all interaction takes place in `Execution View`.

Note: When you launch the debugger, the `Execution View` window is launched in iconified form.

Examining Debugger Data [6]

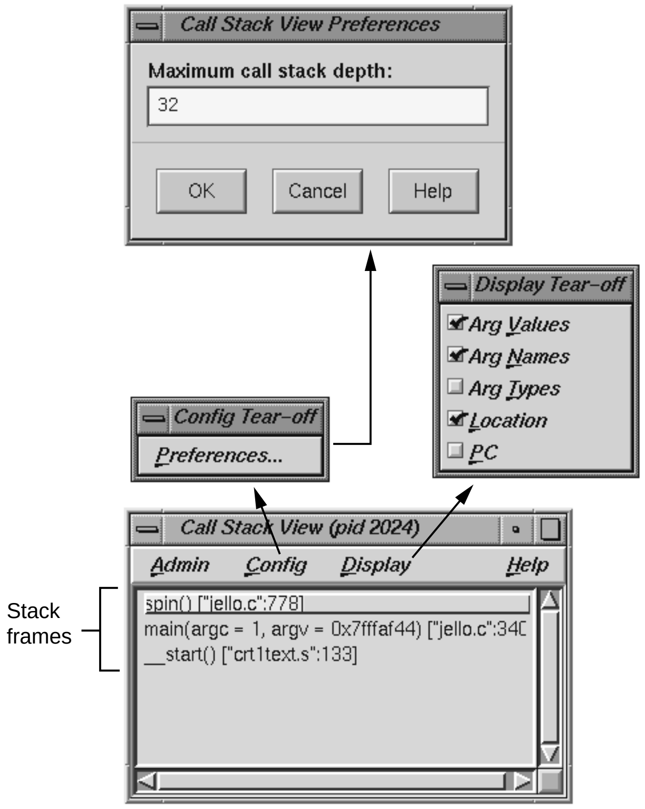
After you have learned how to set traps, the next step is to look at the facilities for examining the data. This chapter covers:

- Tracing Through `Call Stack View`, Section 6.1, page 57
- Evaluating Expressions. Section 6.2, page 60

The Debugger also lets you examine data at the machine level. The tools for viewing disassembled code, machine registers, and data by specific memory location are described in Appendix A, page 117.

6.1 Tracing through `Call Stack View`

The `Call Stack View` window displays the functions in the call stack (referred to as frames) when the process has stopped. The window is shown in Figure 30, page 58, with the major menus displayed.



```
spin() ["jello.c":778]
main(argc = 1, argv = 0x7fffaf44) ["jello.c":340
__start() ["crt1text.s":133]
```

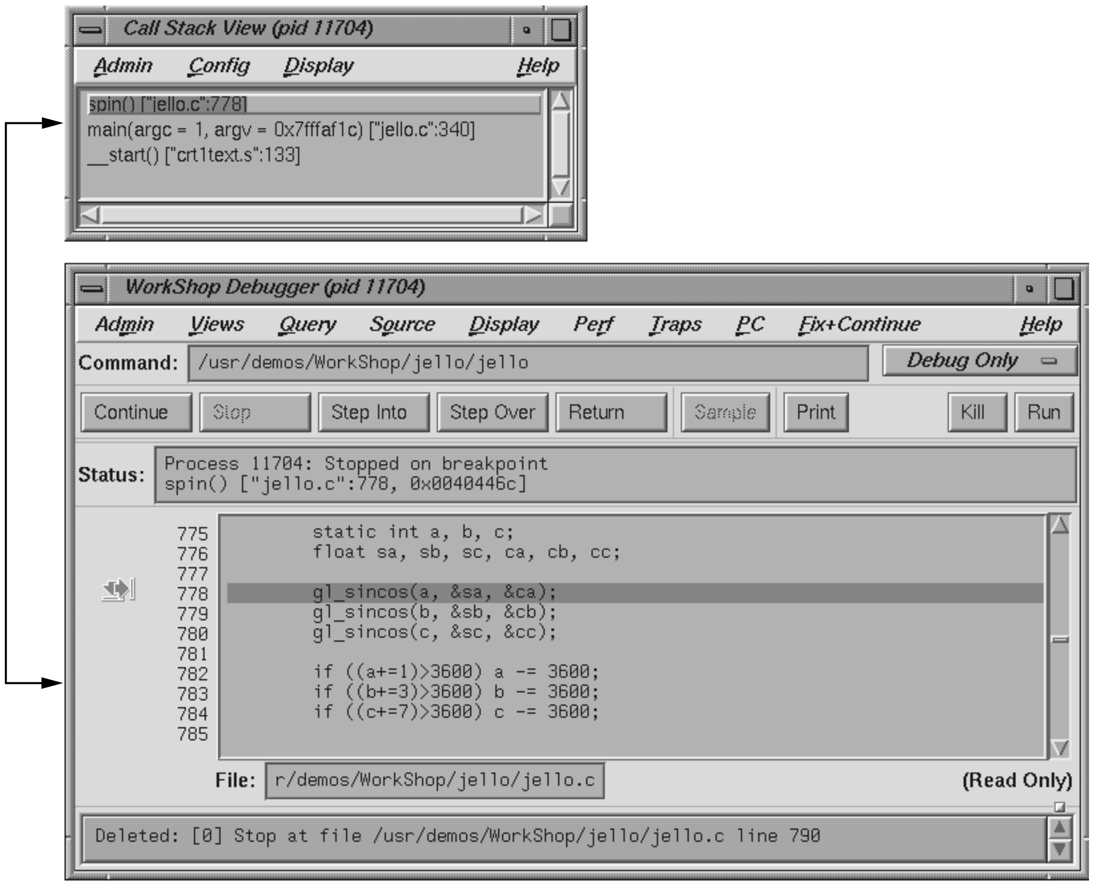
Figure 30. `Call Stack View` Window

The `Call Stack View` window lets you see the argument names, values, and types as well as the locations of functions and the program counter (PC). If symbolic information for the arguments has been stripped from the executable file, the label `<stripped>` appears in place of the arguments. By default call stack depth is set to 10, but you can reset the depth of the `Call Stack View` by selecting `Preferences...` from the `Config` menu.

To move through the call stack, double-click a frame in the stack. The frame becomes highlighted to indicate the current context. The source display in Main View (or `Source View`) scrolls automatically to the location where the function was called and any other active views update. The source display has two special annotations:

- The location of the current program state is indicated by a large green (depending on color scheme) arrow representing the PC.
- The location of the call to the function selected in the `Call Stack View` window is indicated by a smaller blue (depending on color scheme) arrow representing the current context, and the source line becomes highlighted.

Figure 31, page 59, illustrates the correspondence between a frame and the source code when a frame is clicked in the `Call Stack View` window. In this example, the stack frame `spin` has been selected; Main View scrolls to the place where the trap occurred. If the second stack (`main`) had been selected, the window would have scrolled to the place where the function `main` calls `spin`.



```
spin() ["jello.c":778]
main(argc = 1, argv = 0x7fffaf1c) ["jello.c":340]
__start() ["crt1text.s":133]
```

```
/usr/demos/WorkShop/jello/jello
```

```
Process 11704: Stopped on breakpoint
spin() ["jello.c":778, 0x0040446c]
```

```
775         static int a, b, c;
776         float sa, sb, sc, ca, cb, cc;
777
778         gl_sincos(a, &sa, &ca);
779         gl_sincos(b, &sb, &cb);
780         gl_sincos(c, &sc, &cc);
781
782         if ((a+=1)>3600) a -= 3600;
783         if ((b+=3)>3600) b -= 3600;
784         if ((c+=7)>3600) c -= 3600;
785
```

```
r/demos/WorkShop/jello/jello.c
```

```
Deleted: [0] Stop at file /usr/demos/WorkShop/jello/jello.c line 790
```

Figure 31. Tracing through `Call Stack View`

6.2 Evaluating Expressions

You can evaluate any valid expression at a stopping point in the process and trace it through the process. Expressions are evaluated by default in the frame and language of the current context. Expressions may contain data names or constants; however, they may not contain names known only to the C preprocessor, as in a `#define` directive or a macro.

To evaluate expressions, you can use `Expression View`, which lets you evaluate multiple expressions simultaneously, updating their values each time the process stops.

Note: You can also evaluate expressions from the command line.

6.2.1 `Expression View` Window

The `Expression View` window is shown in Figure 32, page 61, with its major menus displayed. The `Expression View` window has two pop-up menus. The `Language` menu is invoked by holding down the right mouse button while the cursor is in the `Expression` column. The `Format` menu is displayed by holding down the right mouse button in the `Result` column.

To specify the expression to be evaluated, first click in the `Expression` column and then enter the expression in the selected field. This expression can be typed directly or pasted in from the source code display. It must be a valid expression in the current or selected language: Ada, C, C++, or Fortran. To change languages, display the `Language` menu and make your selection. When you press `Enter`, the result of the expression is displayed in the `Result` column.

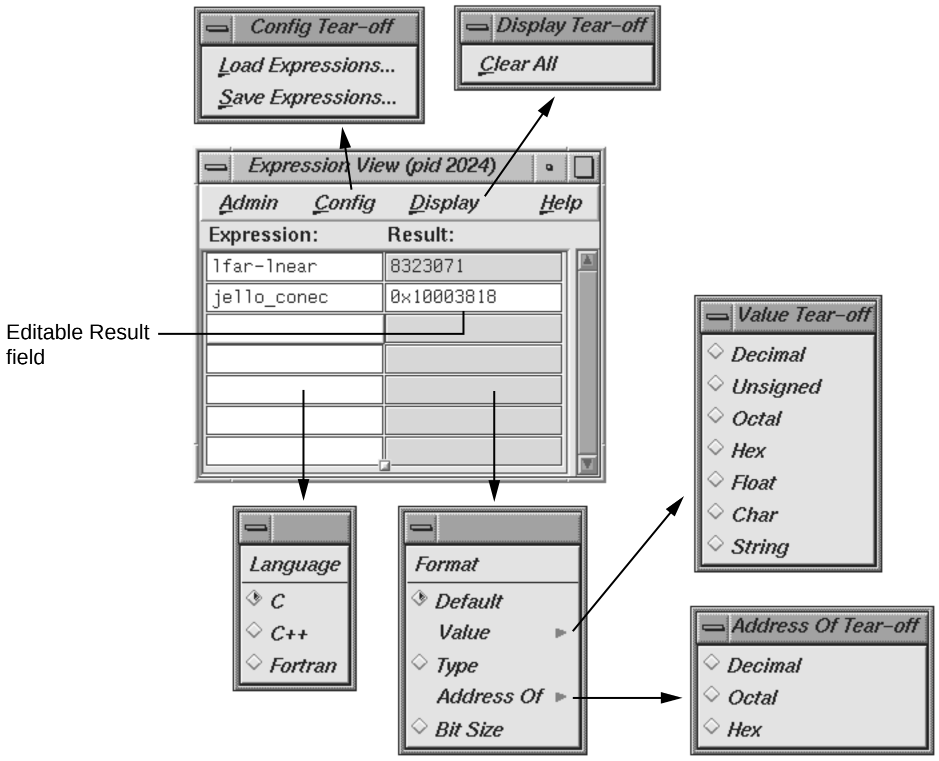


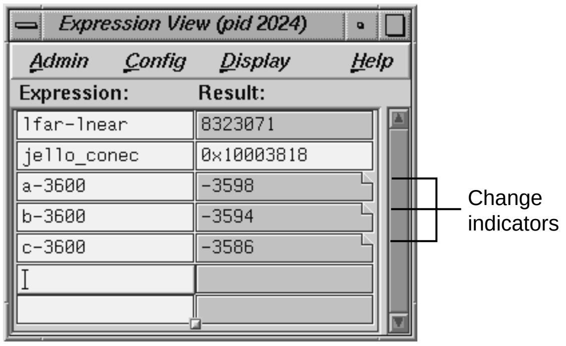
Figure 32. `Expression View` with Major Menus Displayed

To change the type of result information displayed in the right column, hold down the right mouse button over the right column. This displays the `Format` menu. You can see the value as a string, decimal, unsigned, octal, hexadecimal, float, or characters. You can also display the type, the address (in decimal, octal, or hexadecimal), or the size of the result in bits.

⚠ **Caution:** The Debugger uses the symbol table of the target program to determine variable type. Some variables in libraries, such as `errno` and `_environ`, are not fully described in the symbol table. As a result, the Debugger may not know their types. When the Debugger evaluates such a variable, it assumes that the variable is a fullword integer. This gives the correct value for fullword integers or pointers, but the wrong value for non-fullword integers and for floating-point values.

To see the value of a variable of unknown type, use C type cast syntax to cast the address of the variable to a pointer that points to the correct type, for example, the global variable `_environ` should be of type `char**`. You can see its value by evaluating `*(char***)&_environ`.

After you display the current value of the expression, you may find it useful to leave the window open so that you can trace the expression as it changes value from trap to trap (or when you change the current context by double-clicking in the call stack). Like other views involved with variables, `Expression View` has variable change indicators for value fields that let you see previous values, as shown in Figure 33, page 62.



Expression:	Result:
lfar-lnear	8323071
jello_conec	0x10003818
a-3600	-3598
b-3600	-3594
c-3600	-3586

Figure 33. Change Indicators in `Expression View`

Another useful technique is to save your expressions to a file for later reuse. Expressions are saved by choosing `Save Expressions...` from the `Config` menu and retrieved by selecting `Load Expressions...` from the same menu.

6.2.2 Assigning Values to Variables

To assign a value to a variable, click the left column and enter the variable name. The current value appears in the right column. If this `Result` field is editable (highlighted), you can click it and enter a new value or legal expression. Pressing `Enter` performs the assignment. You can perform an assignment to any expression that evaluates to a legal `lvalue` (in C). The C operator = is not valid in `Expression View`. Valid expression operations are shown in the following paragraphs.

6.2.3 Evaluating Expressions in C

The valid C expressions are shown in Table 2, page 63.

Table 2. Valid C Operations

Operation	Symbol
Arithmetic (unary)	+ - ++ – (increment and decrement do not have side effects)
Arithmetic (binary)	+ - * / %
Logical	&& || !
Relational	< > <= >= == !=
Bit	& | ^ << >> ~
Dereference	*
Address	&
Array indexing	[ ]
Conditional	? :
Member extraction	. -> (these operations are interchangeable)
Sizeof	
Type-cast	
Function call	
Assignment	= += -= /= %= >>= <<= &= ^= |= (Note that a new assignment is made at each stepping point. Use assignments with caution to avoid inadvertently modifying variables.)

6.2.3.1 C Function Calls

Function calls can be evaluated in expressions, as long as enough actual parameters are supplied. Arguments are passed by value. Following the rules of C, each actual parameter is converted to a value of the same type as the formal parameter, before the call. If the types of the formal parameters are unknown, integral arguments are widened to full words, and floating-point arguments are converted to doubles.

Functions may return pointers, scalar values, unions, or structs. Note that if the function returns a pointer into its stack frame (rarely a good programming practice), the value pointed to will be meaningless, since the temporary stack frame is destroyed immediately after the call is completed.

Function calls may be nested. For example, if your program contains a successor function `succ`, the Debugger will evaluate the expression `succ(succ(succ(3)))` to 6.

6.2.4 Evaluating Expressions in C++

C++ expressions may contain any of the C operations. You can use the word `this` to explicitly reference data members of an object in a member function. When stopped in a member function, the scope for `this` is searched automatically for data members. Names may be used in either mangled or demangled form. Names qualified by class name are supported (for example, `Symbol::a`).

If you wish to look at a static member variable for a C++ class, you need not specify the variable with the class qualifier if you are within the context of the class. For example, you would specify `myclass::myvariable` for the static variable *myvariable* outside of class *myclass* and `myvariable` inside *myclass*.

6.2.4.1 Limitations

Constructors may be called from `Expression View`, just like other member functions. To call a constructor, you must pass in a first argument that points to the object to be created. C++ function calls have the same possibility of side effects as C functions.

6.2.5 Evaluating Expressions in Fortran

Fortran expressions may contain any of the arithmetic, relational, or logical operators. Relational and logical operator keywords may be spelled in upper case, lower case, or mixed case.

The usual forms of Fortran constants, including complex constants, may be used in expressions. String constants and string operations, however, are not supported. The operators in Table 3, page 65, are supported on data of integral, real, and complex types.

Table 3. Valid Fortran Operations

Operation	Symbol
Arithmetic (unary)	- +
Arithmetic (binary)	- + * / **
Logical	.NOT. .AND. .OR. .XOR. .EQV .NEQV.
Relational	.GT. .GE. .LT. .LE. .EQ. .NE.
Array indexing	( )
Intrinsic function calls (except string intrinsics)	
Function subroutine calls	
Assignment	= (Note that a new assignment is made at each stepping point. Use assignments with caution to avoid inadvertently modifying variables.)

6.2.5.1 Fortran Variables

Names of Fortran variables, functions, parameters, arrays, pointers, and arguments are all supported in expressions, as are names in common blocks and equivalence statements. Names may be spelled in upper case, lower case, or mixed case.

6.2.5.2 Fortran Function Calls

The Debugger evaluates function calls the same way that compiled code does. If an argument can be passed by reference, it is; otherwise, a temporary expression is allocated and passed by reference. Following the rules of Fortran, actual arguments are not converted to match the types of formal arguments. Side effects can be caused by Fortran function calls. A useful technique to protect the value of a parameter from being modified by a function subroutine is to pass an expression such as `(parameter + 0)`instead of just the parameter name. This causes a reference to a temporary expression to be passed to the function rather than a reference to the parameter itself; the value is the same.

Debugging with Fix+Continue: A Tutorial [7]

This chapter provides an interactive sample session that demonstrates most of the Fix+Continue functions. The session outlines common tasks you can perform with Fix+Continue using example C++ application source to illustrate the use of each function. For complete reference information on the Fix+Continue user interface, see Section A.9, page 232.

> **Note:** Fix+Continue functionality within the Debugger is limited to programs compiled with the `-o32` compiler option.

Most steps in the session let you use either the graphical interface or the command line alternatives.

This chapter contains the following sections:

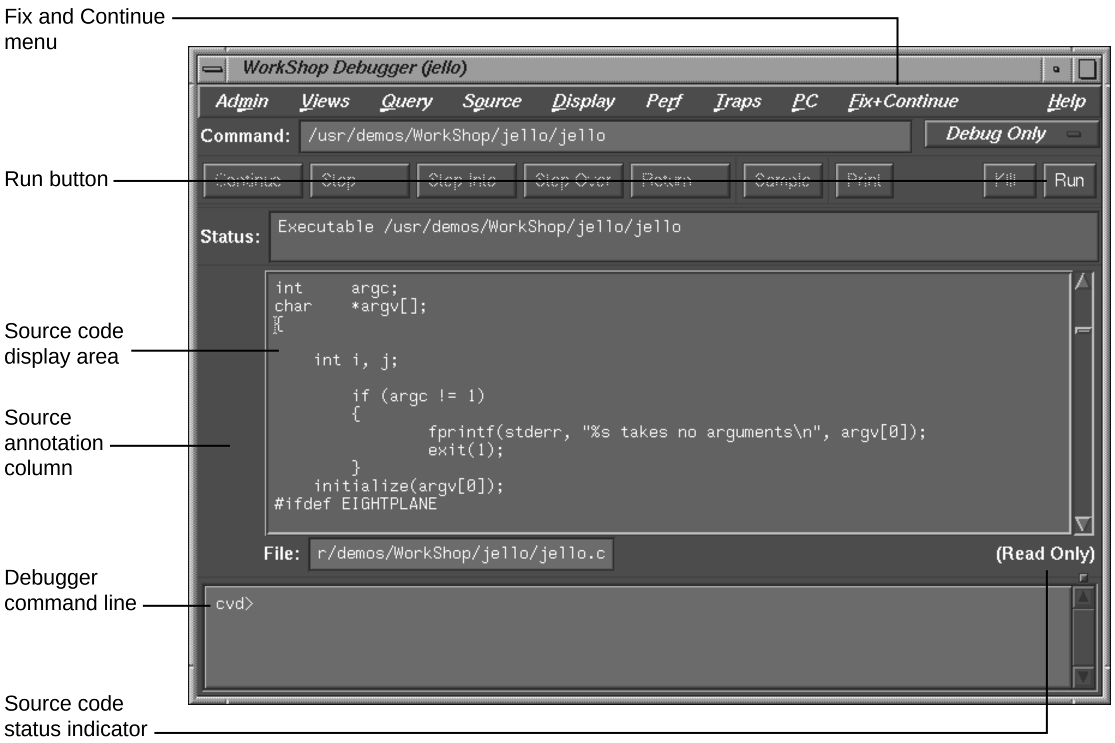
- Setting Up the Sample Session, Section 7.1, page 67
- Redefining a Function, Section 7.2, page 69
- Setting Breakpoints in Redefined Code, Section 7.3, page 73
- Viewing Status, Section 7.4, page 76
- Comparing Original and Redefined Code, Section 7.5, page 76
- Ending the Session, Section 7.6, page 78

7.1 Setting Up the Sample Session

For this tutorial, use the demo files in the `/usr/demos/WorkShop/bounce` directory that contains the complete source code for the C++ application `bounce`. To prepare for the session, you must create the fileset and launch Fix+Continue from the Debugger as shown below:

1. **`cd /usr/demos/WorkShop/bounce`**
2. **`make bounce`**
3. **`cvd bounce &`**

   The `cvd` command brings up the Debugger, from which you can use the Fix+Continue utility. The `Execution View` icon and the Main View window (as shown in Figure 34, page 68) appear. Note that the Debugger shows a source code status indicator of `(Read Only)`.



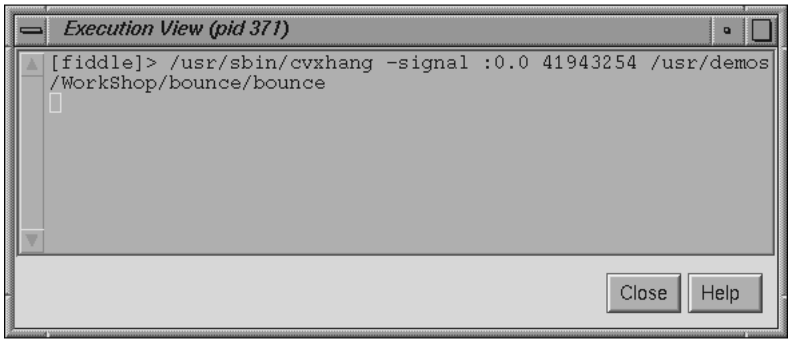
```
/usr/demos/WorkShop/jello/jello
Executable /usr/demos/WorkShop/jello/jello
int     argc;
char    *argv[];
{

    int i, j;

        if (argc != 1)
        {
                fprintf(stderr, "%s takes no arguments\n", argv[0]);
                exit(1);
        }
    initialize(argv[0]);
#ifdef EIGHTPLANE
r/demos/WorkShop/jello/jello.c
cvd>
```

Figure 34. Debugger Main View With `Fix+Continue` Menu

4. Open the `Execution View` and position the window so you can see it and the Debugger Main View.

5. To see what the program does, click `Run`. The `bounce` program opens a window on your desktop. Click `Run` in the new window, and then add balls from the `Actors` menu to see how the program executes. (You may need to resize the `bounce` window.)

6. The `Execution View` shows the program output (see Figure 35, page 69).



```
[fiddle]> /usr/sbin/cvxhang -signal :0.0 41943254 /usr/demos
/WorkShop/bounce/bounce
```

Figure 35. Program Results in `Execution View`

If your screen shows different results, the program files may have been modified during a previous tutorial session.

7.2 Redefining a Function

In this section, you will do the following:

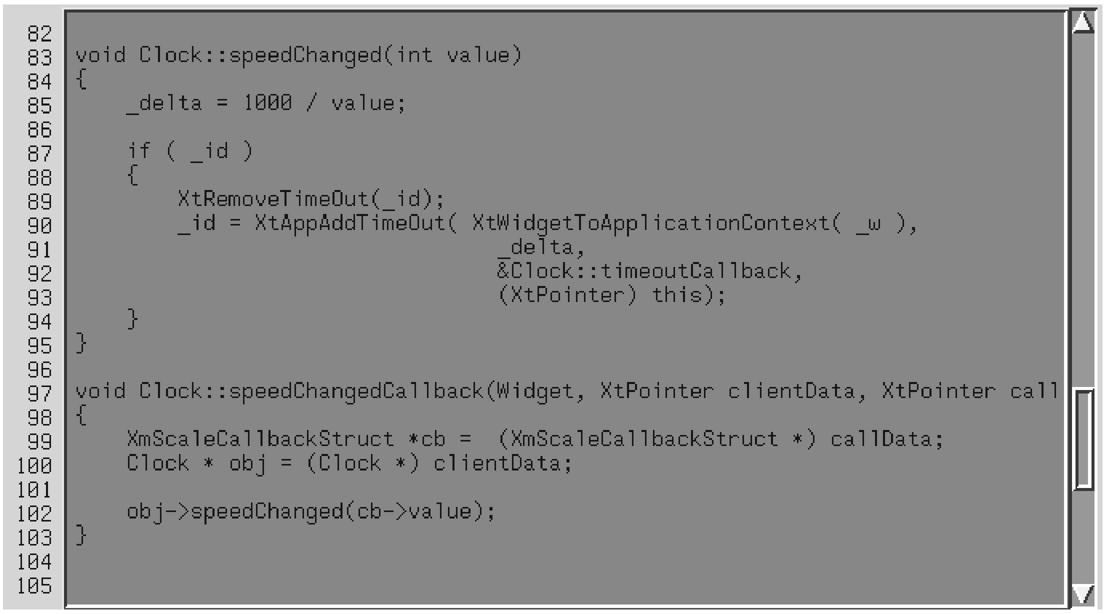
- Edit a function
- Change the code of an existing function and then parse and load the function, rebuilding your program to see the effect of your changes on program output (without recompiling)
- Save the changed function to its own separate file

7.2.1 Editing a Function

1. Choose a function to edit by entering the following on the command line:

   cvd> **func Clock::speedChanged**

   This opens the `Clock.C` file and places the cursor at the beginning of the `Clock::speedChanged` function, as shown in Figure 36.



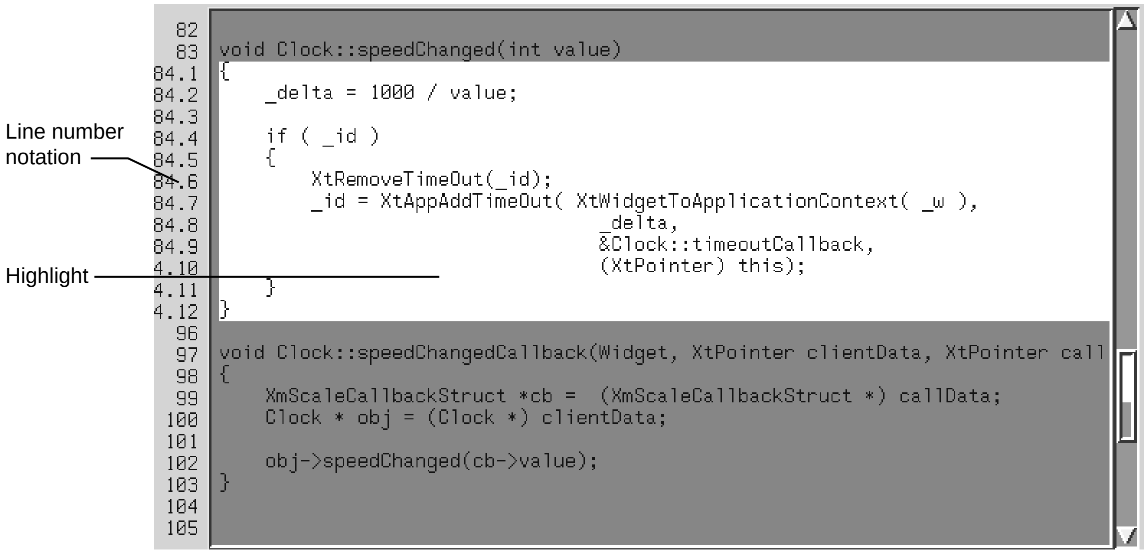
```
 82
 83  void Clock::speedChanged(int value)
 84  {
 85      _delta = 1000 / value;
 86
 87      if ( _id )
 88      {
 89          XtRemoveTimeOut(_id);
 90          _id = XtAppAddTimeOut( XtWidgetToApplicationContext( _w ),
 91                                  _delta,
 92                                  &Clock::timeoutCallback,
 93                                  (XtPointer) this);
 94      }
 95  }
 96
 97  void Clock::speedChangedCallback(Widget, XtPointer clientData, XtPointer call
 98  {
 99      XmScaleCallbackStruct *cb =  (XmScaleCallbackStruct *) callData;
100      Clock * obj = (Clock *) clientData;
101
102      obj->speedChanged(cb->value);
103  }
104
105
```

Figure 36. Selecting a Function for Redefinition

2. Show line numbers by selecting `Show Line Numbers` from the Debugger `Display` menu.

3. Select `Edit` from the Debugger `Fix+Continue` menu, or enter the `Alt-Ctrl-E` keyboard accelerator. The function is highlighted.

4. Note the results as shown in Figure 37, page 71. Line numbers changed to a decimal notation based on the first line number of the function body. The function body highlights to show that it is being edited. The line numbers of the rest of the file are not affected.



```
  82
  83 void Clock::speedChanged(int value)
84.1 {
84.2     _delta = 1000 / value;
84.3
84.4     if ( _id )
84.5     {
84.6         XtRemoveTimeOut(_id);
84.7         _id = XtAppAddTimeOut( XtWidgetToApplicationContext( _w ),
84.8                                _delta,
84.9                                &Clock::timeoutCallback,
4.10                                (XtPointer) this);
4.11     }
4.12 }
  96
  97 void Clock::speedChangedCallback(Widget, XtPointer clientData, XtPointer call
  98 {
  99     XmScaleCallbackStruct *cb =  (XmScaleCallbackStruct *) callData;
 100     Clock * obj = (Clock *) clientData;
 101
 102     obj->speedChanged(cb->value);
 103 }
 104
 105
```

Figure 37. Redefined Function

7.2.2 Changing Code

1. To increase the speed of the ball, change the value of *_delta* from a value of `1000` to a value of 100 by editing the value within the highlighted area. Alternatively, you can use the `replace_source` command to modify the line.

2. Click the `Stop` button in the Debugger to halt the `bounce` process.

3. Select `Parse and Load` from the Debugger `Fix+Continue` menu or enter the `Alt-Ctrl-X` keyboard accelerator.

   Any errors are reported by the `Fix+Continue Error Messages` window.

   If you do have an error, correct it and repeat steps 1 through 3. You can go to the error location by double-clicking the appropriate line in the `Fix+Continue Error Message` window. When you see the change ID and activated status, continue with the next step.

   When the parse and load operation is complete, the highlighting color of the function changes and a report of successful redefinition is displayed. A sample report for changing clock speed is shown in the following example:

```
cvd> func Clock::speedChanged
Change id: 1 redefined
Change id: 1 saved func
Change id: 1 file not saved
Change id: 1 modifed
cvd>
```

4. Select `Continue` from the Debugger Main View.
5. The new value is not active until the function is called. To call the function, adjust the slider bar in the `Bounce` window (see Figure 38, page 72).

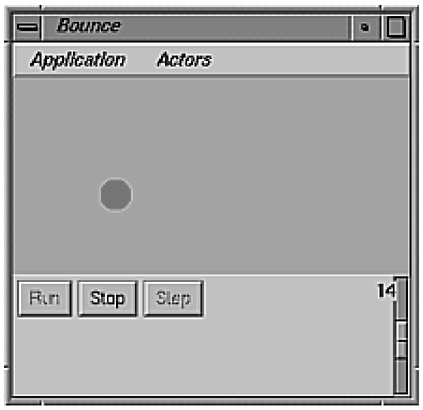


Figure 38. Bounce Window

7.2.2.1 Deleting Changed Code

To cancel any change you have made, select `Delete Edits` from the `Fix+Continue` menu in the Main View.

7.2.2.2 Changing Code From the Debugger Command Line

You also can redefine and check the syntax for a function from the Debugger command line. Try changing *_delta* to 100 by entering the following at the Debugger command line:

```
cvd> replace_source "Clock.C":85
‘‘Clock.C’’:84.0>
‘‘Clock.C’’:84.1>
‘‘Clock.C’’:84.2> _delta = 100 / value;
```

```
‘‘Clock.C’’:84.3> .
```

This generates the following output:

```
Change id: 2 redefined
Change id: 2 modified
Process 5779 stopped at [‘‘select.s’’:12, 0x0fac2010]
Change id: 2 activated
Change id: 2 , build results:
  2  enabled  /usr/demos/WorkShop/bounce/
Clock.C  Clock::speedChanged(int)
cvd>
```

If you prefer to use the command line, experiment with the `add_source` and `redefine` commands to get the same functionality described for the menu commands. For details on each command, refer to Section A.10, page 244.

7.2.3 Saving Changes

Your original source files are not updated until the changed source file is saved. You could save redefined function changes to the `Clock.C` file. However, if you did, the file would not match the tutorial. So observe the following steps:

1. Select `Save As...` from the `Fix+Continue` menu. A *file_name* dialog box opens.

2. The dialog box enables you to save your file changes back to the original source files or save them to a different file. However, since you do not want to save your changes, press the `Cancel` button on the bottom of the dialog box.

Note: You usually want to wait until you are finished with Fix+Continue before you save your changes. In addition to the method described above, you can also save your changes with the `Save All Files...` option of the `Fix+Continue` menu.

7.3 Setting Breakpoints in Redefined Code

To see how the Debugger works with traps in redefined code you will now set breakpoints, run the Debugger, and view the results (Figure 39, page 75).

1. Choose the function `BouncingBall::BouncingBall` by entering the following on the command line:

```
cvd> func BouncingBall::BouncingBall
```

This opens the `BouncingBall.C` file and places the cursor at the beginning of the `BouncingBall::BouncingBall` function.

2. Select `Edit` from the `Fix+Continue` menu or enter `Alt-Ctrl-E`.

3. Enter the following line after line 35.3:

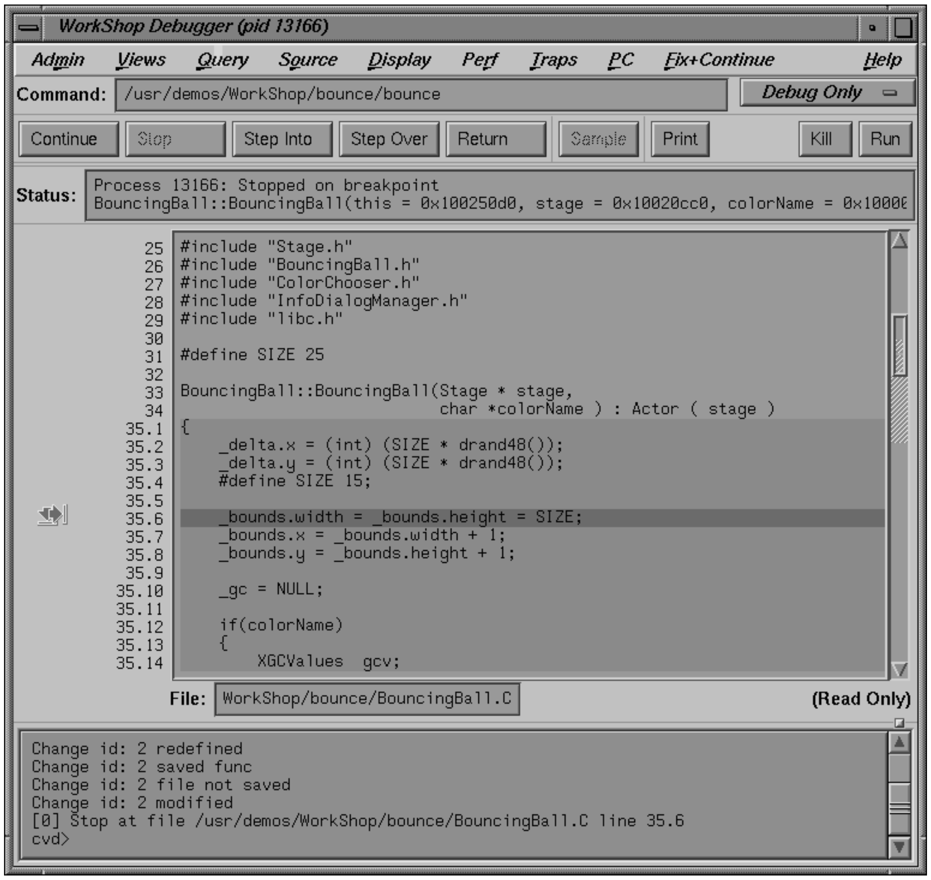
   ```
   #define SIZE 15
   ```

   This makes the size of the balls smaller.

4. Select `Parse and Load` from the `Fix+Continue` menu.

5. Set a breakpoint just after the `SIZE` definition by clicking in the source annotation column at line 35.5.

   Alternatively, you can set a breakpoint by using the command line by entering **`stop at`** *#* or **`b`** *#*. The *#* option is the line number at which you want your breakpoint. Note that in code that has already been parsed and loaded, the line number is in decimal notation.


WorkShop Debugger (pid 13166)
Admin Views Query Source Display Perf Traps PC Fix+Continue Help
Debug Only
Continue Stop Step Into Step Over Return Sample Print Kill Run

Command: /usr/demos/WorkShop/bounce/bounce

Status:
```
Process 13166: Stopped on breakpoint
BouncingBall::BouncingBall(this = 0x100250d0, stage = 0x10020cc0, colorName = 0x1000
```

```
     25 #include "Stage.h"
     26 #include "BouncingBall.h"
     27 #include "ColorChooser.h"
     28 #include "InfoDialogManager.h"
     29 #include "libc.h"
     30
     31 #define SIZE 25
     32
     33 BouncingBall::BouncingBall(Stage * stage,
     34                            char *colorName ) : Actor ( stage )
   35.1 {
   35.2     _delta.x = (int) (SIZE * drand48());
   35.3     _delta.y = (int) (SIZE * drand48());
   35.4     #define SIZE 15;
   35.5
   35.6     _bounds.width = _bounds.height = SIZE;
   35.7     _bounds.x = _bounds.width + 1;
   35.8     _bounds.y = _bounds.height + 1;
   35.9
  35.10     _gc = NULL;
  35.11
  35.12     if(colorName)
  35.13     {
  35.14         XGCValues  gcv;
```

File: WorkShop/bounce/BouncingBall.C (Read Only)

```
Change id: 2 redefined
Change id: 2 saved func
Change id: 2 file not saved
Change id: 2 modified
[0] Stop at file /usr/demos/WorkShop/bounce/BouncingBall.C line 35.6
cvd>
```

Figure 39. Stopping After Breakpoints in Redefined Code

6. Select `Run`, then in the `Bounce` window pull down the `Actors` menu and select `Add Red Ball`. The Debugger command line reports that the process stopped at some point in the code. You see the following information in the Debugger command line:

```
[1] Stop at file /usr/demos/WorkShop/bounce/BouncingBall.C line 35.6
[0] Process 595 stopped at [‘‘BouncingBall.C’’:35, 0x004088d0]
```

7. Select `Call Stack` from the `Views` menu to view the results of the breakpoint.

8. Select `Trap Manager` from the `Views` menu to view the locations of the traps.
9. Remove the breakpoint by clicking on it in the source annotation column.

7.4 Viewing Status

Pull down the `Fix+Continue` menu, choose the `Views` submenu, and select `Status Window`. The `Fix+Continue Status` window opens.

7.5 Comparing Original and Redefined Code

You can use Fix+Continue to compare modified code to the original source. This section shows you several ways to view your changes.

7.5.1 Switching Between Compiled and Redefined Code

If you want to see how the redefined code makes your executable different, follow these steps:

1. Click the `Run` button to view your redefined code. Notice that the balls you add are smaller in your modified version.
2. Place the insertion point in the `BouncingBall` function.
3. Select `Edit<-->Compiled` from the `Fix+Continue` menu. This disables your changes.
4. Click the `Continue` button. Notice that the balls you add are now their original size, and that the Debugger command line states that the change has been deactivated.

   You can get the same results by entering the `disable_changes` # command from the Debugger command line, where # is the redefined function ID number.

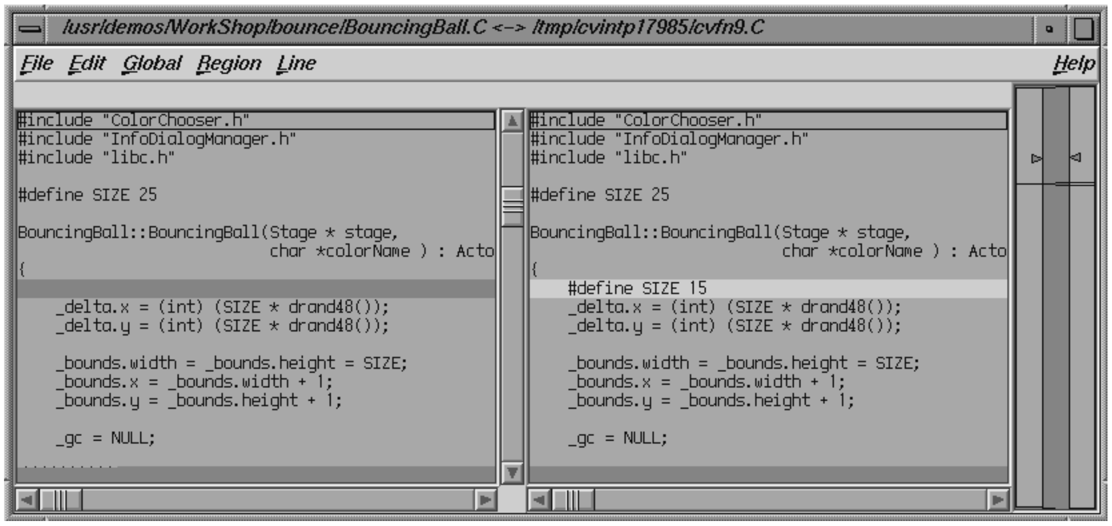
   To reenable your changes, do the following:

5. Click on the `Stop` button.
6. Select `Edit<-->Compiled` from the `Fix+Continue` menu. This reenables your changes. The balls you add will now be smaller.

You can get the same results by entering the `enable_changes` # command at the Debugger command line.

7.5.2 Comparing Function Definitions

1. Place the cursor in the `BouncingBall` function.
2. Pull down the `Fix+Continue` menu and choose the `Show Difference` submenu. From the submenu, select the `For File` option. A window opens displaying an `xdiff` comparison of the files as shown in Figure 40, page 77.


/usr/demos/WorkShop/bounce/BouncingBall.C <-> /tmp/cvintp17985/cvfn9.C
File
Edit
Global
Region
Line
Help

```
#include "ColorChooser.h"
#include "InfoDialogManager.h"
#include "libc.h"

#define SIZE 25

BouncingBall::BouncingBall(Stage * stage,
                           char *colorName ) : Acto
{

    _delta.x = (int) (SIZE * drand48());
    _delta.y = (int) (SIZE * drand48());

    _bounds.width = _bounds.height = SIZE;
    _bounds.x = _bounds.width + 1;
    _bounds.y = _bounds.height + 1;

    _gc = NULL;
```

```
#include "ColorChooser.h"
#include "InfoDialogManager.h"
#include "libc.h"

#define SIZE 25

BouncingBall::BouncingBall(Stage * stage,
                           char *colorName ) : Acto
{
    #define SIZE 15
    _delta.x = (int) (SIZE * drand48());
    _delta.y = (int) (SIZE * drand48());

    _bounds.width = _bounds.height = SIZE;
    _bounds.x = _bounds.width + 1;
    _bounds.y = _bounds.height + 1;

    _gc = NULL;
```

Figure 40. Comparing Compiled and Redefined Function Code

You can get the same result by entering the **`show_diff`** # command from the Debugger command line.

If you do not like `xdiff`, you can change the comparison tool by pulling down the `Fix+Continue` menu, choosing the `Show Difference` submenu, and selecting `Set Diff Tool...`.

7.5.3 Comparing Source Code Files

When you have made several redefinitions to a file, you may need a side-by-side comparison of the entire file. To see how changes to the entire file look, pull down the `Fix+Continue` menu, choose the `Show Difference` submenu, and select `For Function`. This opens a `xdiff` window that displays the entire file rather than just the function.

You can get the same results from the Debugger command line if you enter the following command:

```
show_diff -file BouncingBall.C
```

7.6 Ending the Session

Exit the Debugger by pulling down the `Admin` menu and choosing `Exit`.

Detecting Heap Corruption [8]

This chapter describes heap corruption detection and covers the following topics:

- Typical Heap Corruption Problems, Section 8.1, page 79
- Detecting Heap Corruption Problems, Section 8.2, page 79
- Heap Corruption Detection Tutorial, Section 8.3, page 83

8.1 Typical Heap Corruption Problems

Due to the dynamic nature of allocating and deallocating memory, the heap is vulnerable to these common corruption problems:

Boundary overrun	Occurs when a program writes beyond the `malloc` region.
Boundary underrun	Occurs when a program writes in front of the `malloc` region.
Access to uninitialized memory	Occurs when a program attempts to read memory that has not yet been initialized.
Access to freed memory	Occurs when a program attempts to read or write to memory that has been freed.
Double frees	Occur when a program frees some structure that it had already freed. In such a case, a subsequent reference can pick up a meaningless pointer, causing a segmentation violation.
Erroneous frees	Occur when a program calls `free`() on addresses that were not returned by `malloc`, such as static, global, or automatic variables, or other invalid expressions.

8.2 Detecting Heap Corruption Errors

To detect heap corruption problems, you must relink your executable with a special WorkShop `malloc` library (`-lmalloc_cv`) instead of the standard `malloc` library (`-lmalloc`). By default, the `-lmalloc_cv` library catches the following errors:

- `malloc` call failing (returning NULL)
- `realloc` call failing (returning NULL)
- `realloc` call with an address outside the range of heap addresses returned by `malloc` or `memalign`
- `memalign` call with an improper alignment
- `free` call with an address that is improperly aligned
- `free` call with an address outside the range of heap addresses returned by `malloc` or `memalign`

If you additionally set the `MALLOC_FASTCHK` environment variable, you can detect these errors:

- `free` or `realloc` calls where the words prior to the user block have been corrupted
- `free` or `realloc` calls where the words following the user block have been corrupted
- `free` or `realloc` calls where the address is that of a block that has already been freed. This error may not always be detected if the area around the block is reallocated after it was first `freed`.

8.2.1 Compiling with the Malloc Library

You can compile your executable from scratch as follows:

`cc -g -o` *targetprogram targetprogram.c* **`-lmalloc_cv`**

You can also relink it by using:

`ld -o` *targetprogram targetprogram.o* **`-lmalloc_cv ...`**

An alternative to rebuilding your executable is to use the `_RLD_LIST` environment variable to link the `-lmalloc_cv` library. See the `rld(1)` man page.

8.2.2 Setting Environment Variables

After compiling, invoke the Debugger with your executable as the target. In `Execution View`, you can set environment variables to enable different levels of heap corruption detection from within the `malloc` library, as follows:

`MALLOC_CLEAR_FREE`

Clears data in any memory allocation freed by `free`. It requires that `MALLOC_FASTCHK` be set.

`MALLOC_CLEAR_FREE_PATTERN` *pattern*

Specifies a pattern to clear the data if `MALLOC_CLEAR_FREE` is enabled. The default pattern is `0xcafebeef` for the 32-bit version, and `0xcafebeefcafebeef` for the 64-bit versions. Only full words (double words for 64-bits) are cleared to the pattern.

`MALLOC_CLEAR_MALLOC`

Clears data in any memory allocation returned by `malloc`. It requires that `MALLOC_FASTCHK` be set.

`MALLOC_CLEAR_MALLOC_PATTERN` *pattern*

Specifies a pattern to clear the data if `MALLOC_CLEAR_MALLOC` is enabled. The default pattern is `0xfacebeef` for the 32-bit version, and `0xfacebeeffacebeef` for the 64-bit versions. Only full words (double words for 64-bits) are cleared to the pattern.

`MALLOC_FASTCHK`

Enables additional corruption checks when you call the routines in this library, `libmalloc_cv`. Error detection is done by allocating a space larger than the requested area, and putting specific patterns in front of and behind the area returned to the caller. When `free` or `realloc` is called on a block, the patterns are checked, and if the area was overwritten, an error message is printed to `stderr` using an internal call to the routine `cvmalloc_error`. Under the Debugger, a trap may be set at exit from this routine to catch the program at the error.

`MALLOC_MAXMALLOC` *n*

Where *n* is an integer in any base, sets a maximum size for any `malloc` or `realloc` allocation. Any request exceeding that size is flagged as an error, and returns a NULL pointer.

`MALLOC_NO_REUSE`

Specifies that no area that has been freed can be reused. With this option enabled, no actual free calls are made and process space and swap requirements can grow quite large.

`MALLOC_TRACING`

Prints out all `malloc` events including address and size of the `malloc` or `free`. When running a trace in the course of a performance experiment, you need not set this variable because running the experiment automatically enables it. If the option is enabled when the program is run independently, and the `MALLOC_VERBOSE`environment variable is set to 2 or greater, trace events and program call stacks are written to `stderr`.

`MALLOC_VERBOSE`

Controls message output. If set to 1, minimal output displays; if set to 2, full output displays.

For further information, see the man page for `malloc_cv`.

8.2.3 Trapping Heap Errors Using the Malloc Library

If you are using the `-lmalloc_cv` library, you can use the Trap Manager to set a stop trap at the exit from the function `cvmalloc_error` that is called when an error is detected. Errors are detected only during calls to heap management routines, such as `malloc()` and `free()`. Some kinds of errors, such as overruns, are not detected until the block is `freed` or `realloced`.

When you run the program, the program halts at the stop trap if a heap corruption error is detected. The error and the address are displayed in `Execution View`. You can also examine the `Call Stack View` at this point to get stack information. To find the next error, click the `Continue` button.

If you need more information to isolate the error, set a watchpoint trap to detect a `write` at the displayed address. Then rerun your program. Use `MALLOC_CLEAR_FREE` and `MALLOC_CLEAR_MALLOC` to catch problems from attempts to access uninitialized or freed memory.

Note: You can run programs linked with the -lmalloc_cv library outside of the Debugger. The trade-off is that you have to browse through the stderr messages and catch any errors through visual inspection.

8.3 Heap Corruption Detection Tutorial

This tutorial demonstrates how to detect corruption errors by using the corrupt program. The corrupt program has already been linked with the WorkShop malloc library (libmalloc_cv). The corrupt program listing is as follows:

```
#include <string.h>
void main (int argc, char **argv)
{
  char *str;
  int **array, *bogus, value;

  /* Let us malloc 3 bytes */
  str = (char *) malloc(strlen(‘‘bad’’));

  /* The following statement writes 0 to the 4th byte */
  strcpy(str, ‘‘bad’’);

  free (str);

  /* Let us malloc 100 bytes */
  str = (char *) malloc(100);
  array = (int **) str;

  /* Get an uninitialized value */
  bogus = array[0];

  free (str);
  /* The following is a double free */
  free (str);
/* The following statement uses the uninitialized value as a pointer */
   value = *bogus;
}
```

To start the tutorial:

1. **`cd /usr/demos/WorkShop/mallocbug`**
2. Invoke the Debugger by typing:

   ```
   cvd corrupt &
   ```

   The Debugger Main View window displays with `corrupt` as the target executable.
3. Open the `Execution View` window (if it is minimized) and set the `MALLOC_FASTCHK` and `MALLOC_CLEAR_MALLOC` environment variables.

   If you are using the C shell, type:

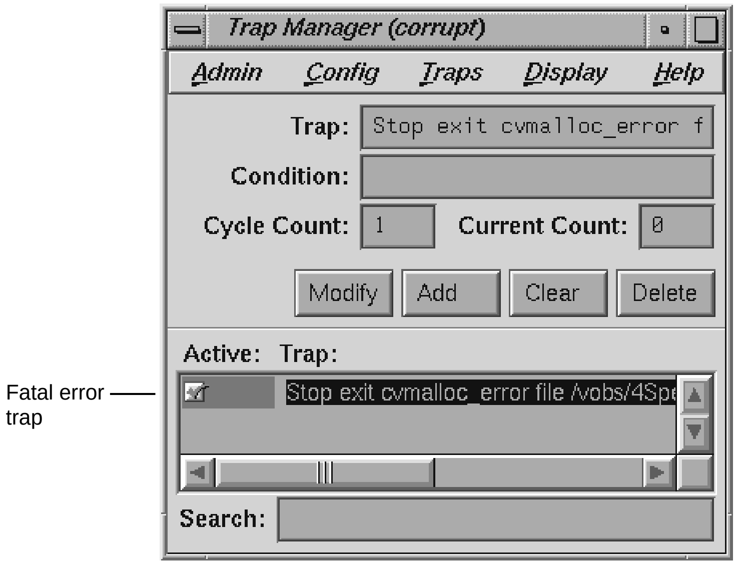
   ```
   setenv MALLOC_FASTCHK
   setenv MALLOC_CLEAR_MALLOC
   ```

   If you are using the Korn or Bourne shell, type:

   ```
   MALLOC_FASTCHK=
   MALLOC_CLEAR_MALLOC=
   export MALLOC_FASTCHK MALLOC_CLEAR_MALLOC
   ```
4. Select `Trap Manager` from the `Views` menu in Main View.
5. Type the following command in the `Trap` field of the `Trap Manager` window and click the `Add:` button:

   ```
   Stop exit cvmalloc_error
   ```

   A stop trap is set at the exit from the `malloc` library routine `cvmalloc_error`. This stops the process when a heap corruption error is detected. The Trap Manager is shown in Figure 41, page 85, with the stop trap set.



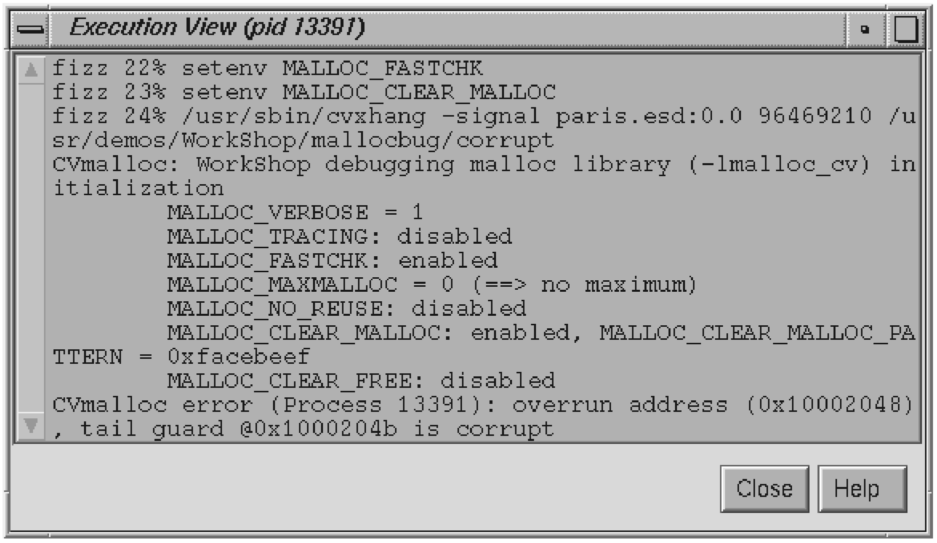
Trap: Stop exit cvmalloc_error f
Cycle Count: 1
Current Count: 0

Stop exit cvmalloc_error file /vobs/4Spe

Figure 41. Setting Traps to Detect Heap Corruption

6. Click Run in the Main View control panel to start program execution and observe Execution View.

   A heap corruption is detected and the process stops at one of the traps. The type of error and its address display in `Execution View` as shown in Figure 42, page 86.

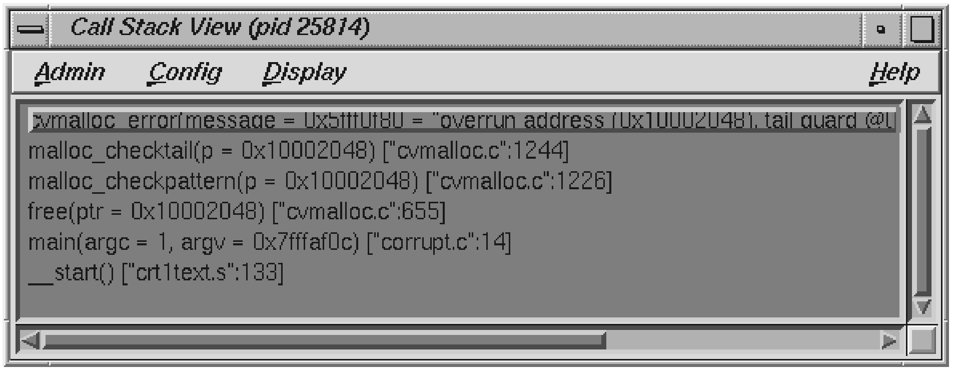


```
fizz 22% setenv MALLOC_FASTCHK
fizz 23% setenv MALLOC_CLEAR_MALLOC
fizz 24% /usr/sbin/cvxhang -signal paris.esd:0.0 96469210 /u
sr/demos/WorkShop/mallocbug/corrupt
CVmalloc: WorkShop debugging malloc library (-lmalloc_cv) in
itialization
        MALLOC_VERBOSE = 1
        MALLOC_TRACING: disabled
        MALLOC_FASTCHK: enabled
        MALLOC_MAXMALLOC = 0 (==> no maximum)
        MALLOC_NO_REUSE: disabled
        MALLOC_CLEAR_MALLOC: enabled, MALLOC_CLEAR_MALLOC_PA
TTERN = 0xfacebeef
        MALLOC_CLEAR_FREE: disabled
CVmalloc error (Process 13391): overrun address (0x10002048)
, tail guard @0x1000204b is corrupt
```

Figure 42. Heap Corruption Warning Shown in `Execution View`

7. Select `Call Stack` from the `Views` menu in Main View.

   `Call Stack View` is opened displaying the call stack frame at the time of the error (see Figure 43, page 86).



```
cvmalloc_error(message = 0x5fff0f80 = "overrun address (0x10002048), tail guard @0
malloc_checktail(p = 0x10002048) ["cvmalloc.c":1244]
malloc_checkpattern(p = 0x10002048) ["cvmalloc.c":1226]
free(ptr = 0x10002048) ["cvmalloc.c":655]
main(argc = 1, argv = 0x7fffaf0c) ["corrupt.c":14]
__start() ["crt1text.s":133]
```

Figure 43. Call Stack at Boundary Overrun Warning

8. Click the `Continue` button in the Main View control panel and watch the `Execution View` and `Call Stack View` windows.

   The process continues from the stop at the boundary overrun warning until it hits the next trap where an erroneous `free` error occurs.

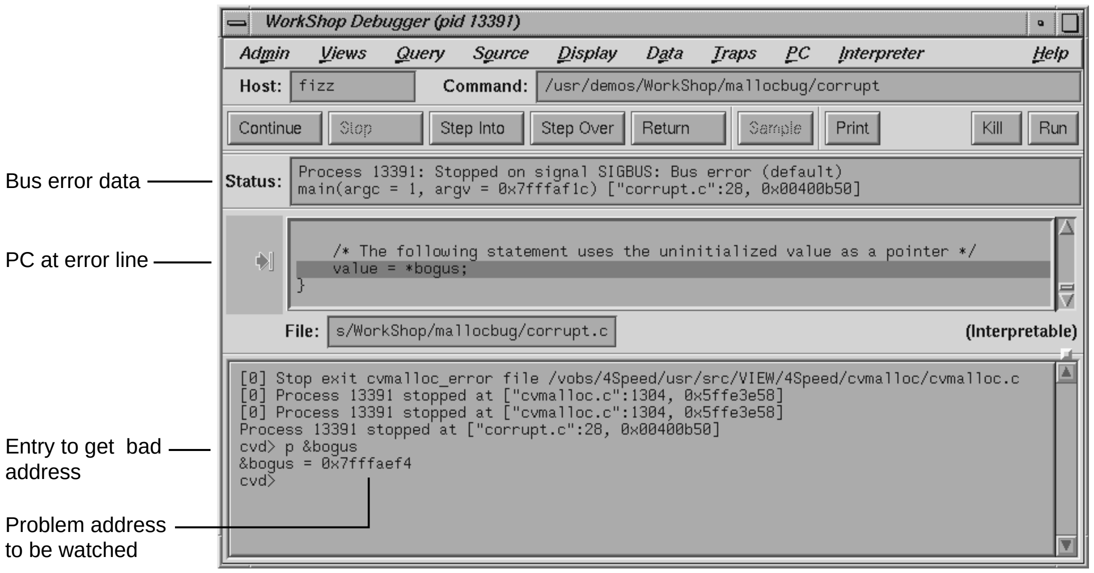
9. Click the `Continue` button again and watch the `Execution View` and `Call Stack View` windows.

   This time the process stops at a bus error. The PC stops at the following statement because `bogus` was set to an uninitialized value:

   ```
   value=*bogus
   ```

10. Enter **`p &bogus`** on the Debugger command line at the bottom of the Main View window.

    This gives us the address for the `bogus` variable and has been done in Figure 44, page 87. We need the bad address so that we can set a watchpoint to find out when it is written to. (This example has an address of `0x7fffaef4`; your address will be different.)


WorkShop Debugger (pid 13391)
Admin Views Query Source Display Data Traps PC Interpreter Help
Host: fizz
Command: /usr/demos/WorkShop/mallocbug/corrupt
Continue Stop Step Into Step Over Return Sample Print Kill Run
Status:
File: s/WorkShop/mallocbug/corrupt.c
(Interpretable)
Bus error data
PC at error line
Entry to get bad address
Problem address to be watched

```
Process 13391: Stopped on signal SIGBUS: Bus error (default)
main(argc = 1, argv = 0x7fffaf1c) ["corrupt.c":28, 0x00400b50]
```

```
    /* The following statement uses the uninitialized value as a pointer */
    value = *bogus;
}
```

```
[0] Stop exit cvmalloc_error file /vobs/4Speed/usr/src/VIEW/4Speed/cvmalloc/cvmalloc.c
[0] Process 13391 stopped at ["cvmalloc.c":1304, 0x5ffe3e58]
[0] Process 13391 stopped at ["cvmalloc.c":1304, 0x5ffe3e58]
Process 13391 stopped at ["corrupt.c":28, 0x00400b50]
cvd> p &bogus
&bogus = 0x7fffaef4
cvd>
```

Figure 44. Main View at Bus Error

11. Deactivate the stop trap by clicking the toggle button next to the trap description in the `Trap Manager` window, and click the `Kill` button in Main View to kill the process.

12. Type the following command in the `Trap` field in the `Trap Manager` window by using the address you obtained from the Debugger command line (see Figure 44, page 87) and click the `Add:` button.

    ```
    stop watch address 0x7fffaef4 for write
    ```

    Use the address from your system, not the one in the tutorial. This sets a watchpoint that is triggered if a `write` is attempted at that address.

13. Click the `Run` button and observe Main View.

    The process stops at the point where the `bogus` variable receives a bad value. Details of the error are displayed in the Main View `Status` field.

Multiple Process Debugging [9]

WorkShop supports performance analysis and debugging of multiprocess applications, including processes spawned either with `fork` or `sproc` and threaded applications. You can perform process control operations on a single process or on all members of a process group. You can attach WorkShop automatically to child processes. You can also specify spawned processes to inherit traps. The Trap Manager provides special trap commands to facilitate debugging multiple processes simultaneously.

> **Note:** The `Multiprocess View` window is for use by C, C++, and Fortran users. If you are debugging Ada code, you should use the `Task View` window available through the `View` menu of Main View (see Section A.3.1, page 149).

This chapter discusses the details of multiprocess debugging in WorkShop and includes the following topics:

- Debugging with Multiprocess View, Section 9.1, page 89
- Controlling Execution and Setting Traps in a Multiprocess Program, Section 9.2, page 92
- Debugging a Multiprocess Fortran Program, Section 9.3, page 98

9.1 Debugging with Multiprocess View

Multiprocess View operates on a process group. By default, a process group includes the parent process and all descendants spawned by `sproc`. Through a preferences option, processes spawned with `fork` during the session can be added to the process group automatically when they are created. Any process to which you have read / write access can also be added to the process group, if desired. All `sproc`'d processes must be in the same process group, since they share information.

> **Note:** Any child process that performs an `exec` with `setuid` (set user ID) enabled will not become part of the process group.

Each process in the session can have a standard Main View session associated with it. However, all processes in a process group share a single `Multiprocess View` window. Selecting `Multiprocess View...` from the `Admin` menu in Main View for any process in the group brings up the `Multiprocess View` window.

When debugging multiprocess applications, you should disable the `SIGTERM` signal by selecting the `Signal Panel` option from the `Views` menu of Main View. Although multiprocessing debugging is possible with `SIGTERM` enabled, the multiprocess application may not terminate gracefully after execution is complete.

Currently, Multiprocess View handles the following multiple process situations:

- *True multiprocess program*, which refers to a tightly integrated system of `sproc`’d processes, generated by the MIPSpro Automatic Parallelization Option. For more information on parallel processing, see the `auto_p(5)` man page, or the *MIPSpro Automatic Parallelizer Programmer’s Guide.*
- *Auto-fork application*, which is a process that spawns a child process and then runs in the background.
- *Fork application*, which is a process that spawns child processes and can interact with them. The WorkShop Performance Analyzer supports applications that `fork` but not those that `exec`.
- *Locally distributed application*, which is an application that involves two different executables running in different processes on the same host coordinated by a rendezvous mechanism. To use the Performance Analyzer, you must have a Main View for each process and enable data collection accordingly.

Multiprocess View does not support remotely distributed applications.

9.1.1 Invoking the Parent Process

The first step in debugging multiple processes is to invoke the Debugger with the parent process. Then select `Multiprocess View` from the `Admin` menu to bring up the `Multiprocess View` window. Figure 45, page 91, shows a typical `Multiprocess View` window.

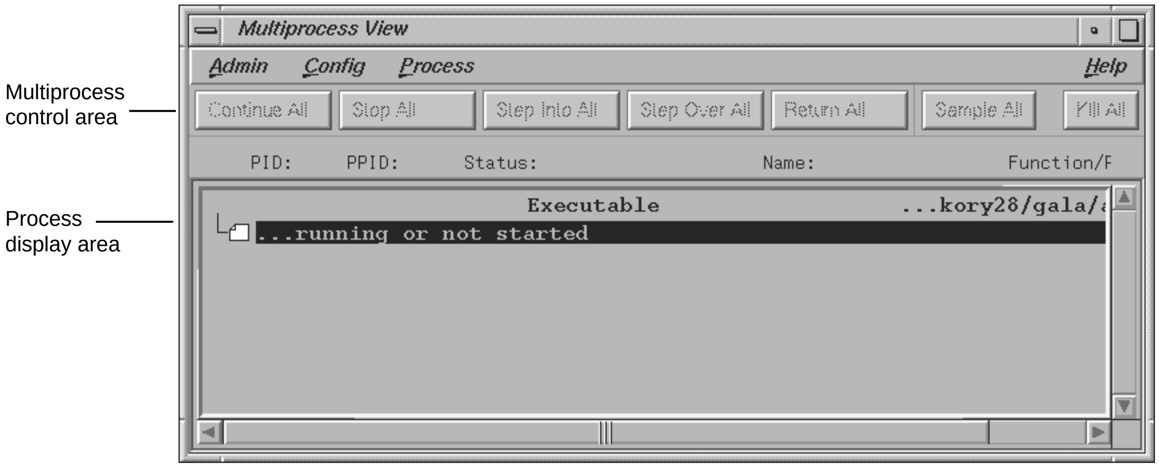


Figure 45. `Multiprocess View`

To get more information about a process or thread displayed in the process display area, use the right mouse button to click on the process or thread entry. This action pops up a `Process` menu that is applicable to the selected entry. From the `Process` menu you can change entry focus, create a new window, focus attention to a user–entered thread, and add or remove an entry. For complete details about the `Process` menu, see Section A.2.5, page 143.

9.1.2 Viewing Process Status

When the `Multiprocess View` first displays, it lists the status of all processes and threads in the process group. For definitions of the various status and states, see Section A.2.3.

9.1.3 Using Control Buttons

The `Multiprocess View` window uses the same control buttons as Main View with the following exceptions:

- Buttons are applied to all processes as a group.
- There is no separate `Run` button.

Using a control button in the Multiprocess View window has the same effect as clicking the button in the Main View window of each individual process. For definitions of the buttons, see Section A.2.4, page 142.

9.1.4 Multiprocess Traps

As discussed in Chapter 4, page 39, the trap qualifiers [all] and [pgrp] are used in multiprocess analysis. The [all] entry stops or samples all processes when a trap fires. The [pgrp] entry sets the trap in all processes within the process group containing the trap location. The qualifiers can be entered by default by using the Group Trap Default and Stop All Default selections in the Traps menu of Trap Manager.

9.1.5 Adding and Removing Processes

To add a process, select Add... from the Process menu. The Add Process dialog displays. Select one of the listed processes or enter a process ID in the Process ID field and click the OK button.

To remove a process, click on the process name and select Remove from the Process menu. Be aware that a process in a sproc process group cannot be removed from the process group. Likewise, you cannot remove a pthread from a pthread group.

9.1.6 Multiprocess Preferences

The Preferences... option in the Config menu brings up the Multiprocess View Preferences dialog. The preferences on this dialog lets you determine when a process is added to the group, specify process behavior, specify the number of call stack levels to display, and so forth.

For details about Multiprocess View Preference options, see Section A.2.6, page 144.

9.2 Controlling Execution and Setting Traps in a Multiprocess Program

This section uses a C program that generates numbers in the Fibonacci sequence to demonstrate some common tasks when using cvd to debug mp code. The following tasks are demonstrated:

- Stopping a child process on a sproc

- Using the `Multiprocess View` buttons to control all processes
- Setting traps in the parent process only
- Setting group traps

The `fibo` program uses `sproc` to split off a child process, which in turn uses `sproc` to split off a grandchild process. All three processes churn out Fibonacci numbers until stopped. If you installed the demo programs, you can find the source for `fibo.c` in the `/usr/demos/WorkShop/mp` directory. A listing of the `fibo.c` directory follows:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/prctl.h>

int NumberToCompute = 100;
int fibonacci();
void run(),run1();

int fibonacci(int n)
{
int f, f_minus_1, f_plus_1;
int i;

      f = 1;
      f_minus_1 = 0;
      i = 0;

      for (; ;) {
          if (i++ == n) return f;
          f_plus_1 = f + f_minus_1;
           f_minus_1 = f;
           f = f_plus_1;
      }
}

void run()
{
int fibon;
      for (; ;) {
          NumberToCompute = (NumberToCompute + 1) % 10;
          fibon = fibonacci(NumberToCompute);
          printf("%d'th fibonacci number is %d\n",
               NumberToCompute, fibon);
      }
}

void run1()
{
int grandChild;

      errno = 0;
      grandChild = sproc(run,PR_SADDR);

      if (grandChild == -1) {
          perror("SPROC GRANDCHILD");
      }
      else
          printf("grandchild is %d\n", grandChild);
      run();
}

void main ()
{
int second;

      second = sproc(run1,PR_SADDR);
      if (second == -1)
          perror("SPROC CHILD");
      else
          printf("child is %d\n", second);

      run();
      exit(0);
}
```

To start, compile the program and run the Debugger.

1. Compile `fibo.c` by entering the following command:

   cc -g fibo.c -o fibo

2. Invoke the Debugger on `fibo` as follows:.

   cvd fibo &

3. Bring up the `Multiprocess View` window by selecting `Multiprocess View...` from the `Admin` menu.

In the next section, you will set options to control how the process executes.

9.2.1 Using Multiprocess View to Control Execution

To examine each process as it appears, you need to stop child processes as they are created with `sproc`. You can control Debugger behavior towards `sproc` processes by setting various Multiprocess preferences.

1. Select `Preferences...` from the `Config` menu on the `Multiprocess View` window.
2. Deactivate `Resume child after attach on sproc` by toggling off the switch.

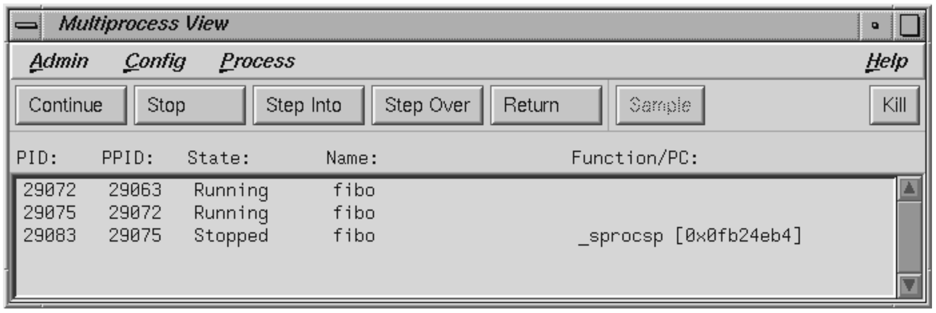
   At the same time, you can turn off trap inheritance, so you can experiment with trap setting later.
3. Click the `OK` button to accept the change.

   Now you can run the process.
4. In the Main View, click the `Run` button.

   If you watch the `Multiprocess View` window, you see the main process appear and spawn a child process. The child process stops as soon as it appears, since you turned off the `Resume child after attach on sproc` option. You can now use `Multiprocess View` to open a new main view for the child process.
5. Use the right mouse button to click on the name of the child process in the `Multiprocess View` window and select the `Create a new window` option from the `Process` menu. A new window is displayed that launches a debug session for the process.

   Use the buttons on the `Multiprocess View` window to control all processes simultaneously, or use the buttons in each of the Main Views to control each process separately.

   > **Note:** You may get a warning that `sproc.s` is missing. This is a reference to assembly code and can be ignored.
6. Click the `Continue` button in the `Multiprocess View` window. The first child now spawns a grandchild process that stops in `sprocsp`, as shown in Figure 46, page 96:



```
PID:     PPID:    State:       Name:                  Function/PC:
29072    29063    Running      fibo
29075    29072    Running      fibo
29083    29075    Stopped      fibo                   _sprocsp [0x0fb24eb4]
```

Figure 46. Examining Process State Using `Multiprocess View`

9.2.2 Using the Trap Manager to Control Trap Inheritance

This section shows using the Trap Manager to set traps that affect one or all processes in the `fibo` process group. For complete information on using the Trap Manager, refer to Chapter 4, page 39.

1. In the Main View for the parent process, select `Trap Manager` from the `Views` menu.

   Traps set using the `Traps` menu in any of the Main View windows affect only the process controlled by the individual Main View. For example, see what happens if you set a stop trap in the first executable line of `run()`, which is line 32:

   ```
   32 NumbertoCompute = (NumbertoCompute + 1) % 10;
   ```

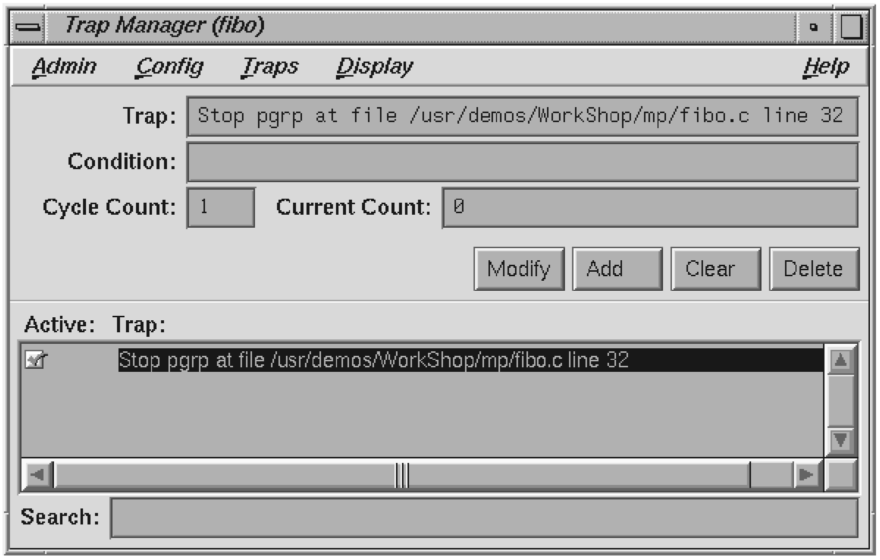
2. Using the `Traps` menu of the parent process, set a stop trap at line 32 of `fibo.c`.

   Only the parent process halts. The child processes continue running, as the `Multiprocess View` window confirms.

   You can use the Trap Manager to edit the trap so that it affects the whole process group.

3. Insert the word **`pgrp`** after the word **`Stop`**.

   The trap should read `Stop pgrp at..`, as shown in Figure 47, page 97.



Trap: Stop pgrp at file /usr/demos/WorkShop/mp/fibo.c line 32
Condition:
Cycle Count: 1
Current Count: 0

Active: Trap:
✓ Stop pgrp at file /usr/demos/WorkShop/mp/fibo.c line 32

Search:

Figure 47. Modifying a Trap to Affect a Process Group

4. Click the `Modify` button to accept your change to the trap. The trap now affects the two child processes. Watch the `Multiprocess View` window to see the entire process group stop at the trap on line 32.

5. If you set traps by using the Trap Manager, select `Group Trap Default` from the `Traps` menu. This makes the traps affect the entire process group.

6. In the Main View of the parent process, place the cursor in any executable line in the function `fibonacci` and select `At Source Line` from the `Traps` menu of the `Trap Manager` window.

   The trap you have just set includes the modifier `pgrp`. It automatically affects both child processes.

7. Select `Exit` from the `Admin` menu in each Main View to end this tutorial.

   You must explicitly close the `Multiprocess View` window. It does not close when the Main View windows do.

9.3 Debugging a Multiprocess Fortran Program

The first part of this section presents a few standard techniques to assist you in debugging a parallel program. The second part shows you how to debug the sample program.

9.3.1 General Fortran Debugging Hints

Debugging a multiprocessed program is more difficult than debugging a single-processor program. Therefore, try to isolate the problem as much as possible and try to debug as much as possible on the single-processor version. If you can, reduce the problem to a single `C$DOACROSS` loop.

Once you have isolated the problem to a specific `DO` loop, try changing the order of iterations in a single-processor version. If the loop can be multiprocessed, then the iterations can execute in any order and produce the same answer. If the loop cannot be multiprocessed, changing the order frequently causes the single-processor version to fail. If it fails, you can use standard single-process debugging techniques to find the problem.

If this technique fails, you need to debug the multiprocessed version. Compile your code with the `-g` and `-mp_keep` flags. The `-mp_keep` flag saves the file containing the multiprocessed `DO` loop Fortran code. The compiler saves the code in a file named the following:

`$TMPDIR/P`<*user_subroutine_name*><*machine_name*><*pid*>

The `user_subroutine_name` option is the name of the subroutine containing the `DOACROSS`, the `machine_name` option is your machine name, and the `pid` option is the process ID number of the compilation.

If you have not set the `TMPDIR` environment variable, `/tmp` is used.

9.3.2 Multiprocess Debugging Session

This section walks you through the process of using the Debugger to debug a small segment of incorrectly multiprocessed code.

If you installed the demo programs, you can find source for the code you will be debugging, `total.f`, in the directory `/usr/demos/WorkShop/mp`. A listing follows:

```
program driver
      implicit none
      integer iold(100,10), inew(100,10),i,j
      double precision aggregate(100, 10),result
      common /work/ aggregate
      call total(100, 10, iold, inew)
      do 20 j=1,10
        do 10 i=1,100
          result=result+aggregate(i,j)
10    continue
20  continue
      write(6,*)' result=',result
      stop
      end

      subroutine total(n, m, iold, inew)
      implicit none
      integer n, m
      integer iold(n,m), inew(n,m)
      double precision aggregate(100, 100)
      common /work/ aggregate
      integer i, j, num, ii, jj
      double precision tmp

      C$DOACROSS LOCAL(i,ii,j,jj,num)
      do j = 2, m-1
        do i = 2, n-1
          num = 1
          if (iold(i,j) .eq. 0) then
            inew(i,j) = 1
          else
          num = iold(i-1,j) +iold(i,j-1) + iold(i-1,j-1) +
&         iold(i+1,j) + iold(i,j+1) + iold(i+1,j+1)
            if (num .ge. 2) then
              inew(i,j) = iold(i,j) + 1
            else
              inew(i,j) = max(iold(i,j)-1, 0)
            end if
          end if
          ii = i/10 + 1
          jj = j/10 + 1
          aggregate(ii,jj) = aggregate(ii,jj) + inew(i,j)
        end do
      end do
      end
```

In the program, the local variables are properly declared. The `inew` always appears with `j` as its second index, so it can be a share variable when multiprocessing the `j` loop. The `iold`, `m`, and `n` are only read (not written), so they are safe. The problem is with `aggregate`. The person analyzing this code reasoned that because `j` is always different in each iteration, `j/10` will also be different. Unfortunately, since `j/10` uses integer division, it often gives the same results for different values of `j`.

While this is a fairly simple error, it is not easy to see. When run on a single processor, the program always gets the right answer. Sometimes it gets the right answer when multiprocessing. The error occurs only when different processes attempt to load from and/or store into the same location in the `aggregate` array at exactly the same time.

Here are the steps in this exercise:

1. First try reversing the order of the iterations. Replace

   ```
   do j = 2, m-1
   ```

   with

   ```
   do j = m-1, 2, -1
   ```

   This still gives the right answer when running with one process but the wrong answer when running with multiple processes. The local variables look right, there are no equivalence statements, and `inew` uses only simple indexing. The likely item to check is `aggregate`. Your next step is to look at `aggregate` with the Debugger.

2. Compile the program with the `-g -mp_keep` options:

   ```
   % f77 -g -mp -mp_keep total.f -o total
   ```

3. If your debugging session is not running on a multiprocessor machine, you can force the creation of two threads for example purposes by setting an environment variable. If you use the C shell, type:

   ```
   % setenv MP_SET_NUMTHREADS 2
   ```

   Is you use the Korn or Bourne shell, type:

   ```
   MP_SET_NUMTHREADS=2
   export MP_SET_NUMTHREADS
   ```

4. Start the Debugger, type:

   ```
   % cvd total &
   ```

The Debugger Main View window displays.

5. Choose `Go To Line...` from the `Source` menu and select line 43. This takes you to line 43, which is the following:

   ```
   aggregate(ii,jj) = aggregate(ii,jj) + inew(i,j)
   ```

   The subroutine touches `aggregate` in only one place, line 43. You want to set a stop trap at this line, so you can see what each thread is doing with `aggregate`, `ii`, and `jj`. You also want this trap to affect all threads of the process group. One way to do this is to turn on trap inheritance using the `Multiprocess View Preferences` dialog box. Another way is to use the Trap Manager to specify group traps, as follows.

6. From the `Views` menu, select `Trap Manager`.

7. In the `Trap Manager` window, pull down the `Traps` menu. Select the `Group Trap Default` option from the menu.

8. Place the cursor in line 43 in the Main View window to select the line.

9. From the `Traps` menu of the `Traps Manager` window, select `At Source Line`.

   This sets the stop trap, which should read something like the following trap:

   ```
   Stop pgrp in file /usr/demos/WorkShop/mp/total.f line 43
   ```

10. Bring up the `Multiprocess View` window to monitor status of the two processes.

    Now you are ready to run the program.

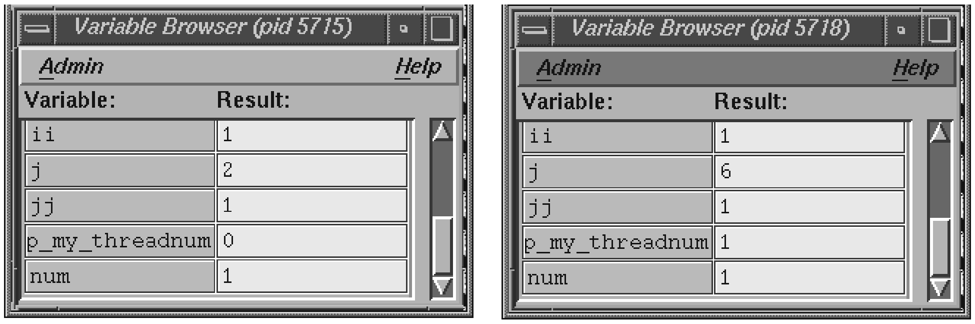
11. Click the `Run` button in the Main View window.

    As you watch the `Multiprocess View`, you see the two processes appear, run, and stop in the function `_total_25_aaaa`. The Main View window is now relative to the master process.

12. Use the right mouse button to click on the name of the slave process in the `Multiprocess View` window and select the `Create a new window` option from the `Process` menu.

    A new window is displayed that launches a debug session for the process.

    Then, invoke the Variable Browser on each process. Look at `ii` and `jj` in Figure 48, page 102.



Variable:	Result:
ii	1
j	2
jj	1
p_my_threadnum	0
num	1

Variable:	Result:
ii	1
j	6
jj	1
p_my_threadnum	1
num	1

Figure 48. Comparing Variable Values from Two Processes

They have the same values in each process; therefore, both processes may attempt to write to the same member of the array `aggregate` at the same time. So `aggregate` should not be declared as a share variable. You have found the bug in your parallel Fortran program.

Using the X/Motif Analyzer: A Tutorial [10]

This chapter provides a sample session that demonstrates most of the X/Motif Analyzer functions. The session outlines common tasks you can perform with the X/Motif Analyzer.

This chapter contains the following sections:

- Setting Up the Sample Session, Section 10.1, page 103
- Navigating the Widget Structure, Section 10.2, page 105
- Examining Widgets, Section 10.3, page 107
- Setting Callback Breakpoints, Section 10.4, page 109
- Using Additional Features of the Analyzer, Section 10.5, page 112
- Ending the Session, Section 10.6, page 116

10.1 Setting Up the Sample Session

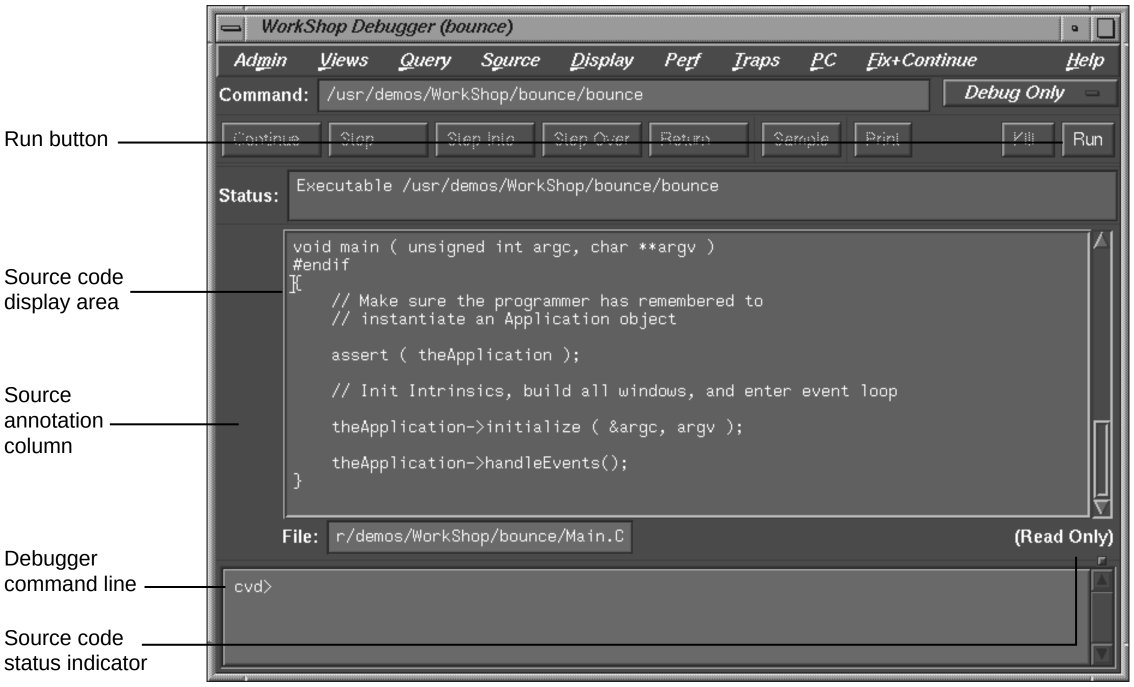
For this tutorial, use the demo files in the `/usr/demos/WorkShop/bounce` directory that contains the complete source code for the C++ `bounce` application. To prepare for the session, you first need to create the fileset, then launch the X/Motif Analyzer from the Debugger.

10.1.1 Preparing the Fileset

You must enter the commands listed below:

1. **`cd /usr/demos/WorkShop/bounce`**
2. **`make bounce`**
3. **`cvd bounce &`**

   The `cvd` command brings up the Debugger, from which you can use the X/Motif Analyzer. Upon invocation, you see the `Execution View` icon and Main View (shown in Figure 49, page 104) appear. Notice that the source code status indicator in the lower-right corner of the Debugger Main View is `(Read Only)`.



```
/usr/demos/WorkShop/bounce/bounce
```

```
Executable /usr/demos/WorkShop/bounce/bounce
```

```
void main ( unsigned int argc, char **argv )
#endif
{
    // Make sure the programmer has remembered to
    // instantiate an Application object

    assert ( theApplication );

    // Init Intrinsics, build all windows, and enter event loop

    theApplication->initialize ( &argc, argv );

    theApplication->handleEvents();
}
```

```
r/demos/WorkShop/bounce/Main.C
```

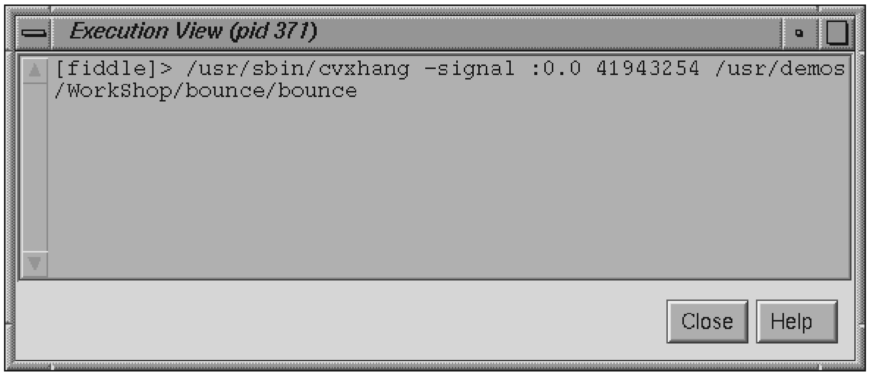
```
cvd>
```

Figure 49. Debugger Main View

4. Click the `Execution View` icon to open the window and position the window so you can see it and the Debugger Main View.

5. To see what the program does, click `Run`. The `bounce` program opens a window. Click `Run` in the new window, resize the window to make it taller, and then add balls from the `Actors` menu to see how the program executes.

6. The `Execution View` shows the program output (see Figure 50, page 105).



```
[fiddle]> /usr/sbin/cvxhang -signal :0.0 41943254 /usr/demos
/WorkShop/bounce/bounce
```

Figure 50. Program Results in `Execution View`

Note: If your screen shows different results, the program files may have been modified during a previous tutorial session.

10.1.2 Launching the X/Motif Analyzer

Once the `bounce` fileset is built and the debugger is active, you need to launch the X/Motif Analyzer with the following steps:

1. Pull down the `Views` menu in the menu bar of the debugger Main View.
2. Select `X/Motif Analyzer`.
3. Click `OK` when asked if you want to change your `$LD_LIBRARY_PATH` environment variable to include `/usr/lib/WorkShop/Motif`. These are instrumented versions of the Silicon Graphics libraries and add special support for the X/Motif Analyzer, in addition to containing symbols.
4. Click `Kill` in the Debugger Main View to kill `bounce`.

You are now ready to begin the sample session.

10.2 Navigating the Widget Structure

After being launched, the X/Motif Analyzer brings up the `X/Motif Analyzer` window with an empty `Widget` examiner tab panel. The tab panels also show the `Breakpoints`, `Trace`, and `Tree` examiner tab panels (see Figure 51, page 106).

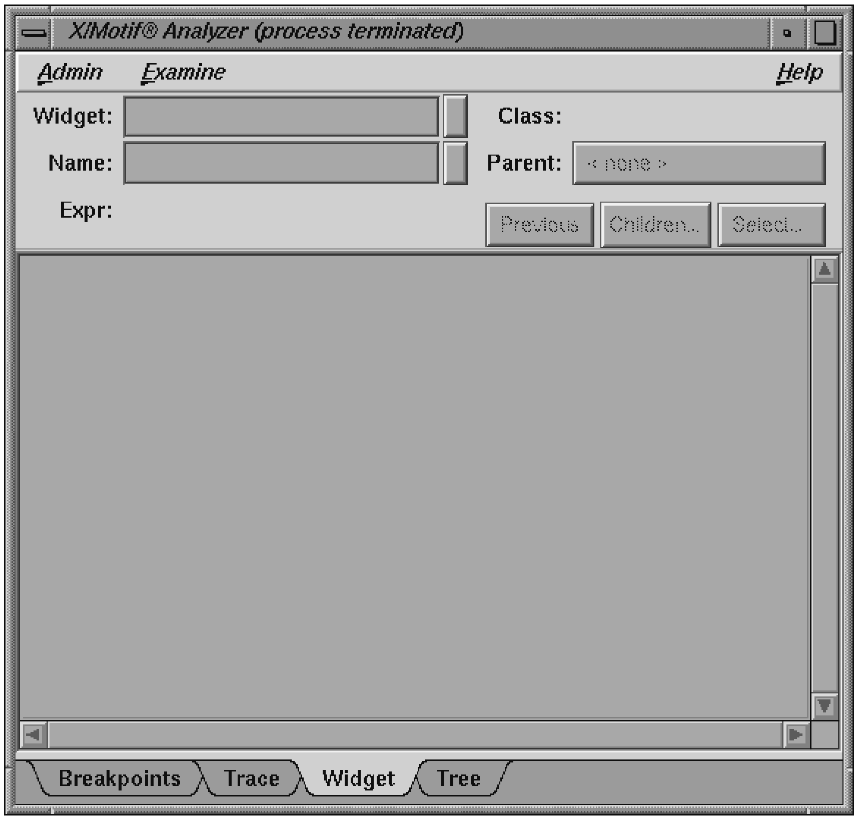


Figure 51. First View of the `X/Motif Analyzer` (`Widget` Examiner)

1. Click `Run` in the Debugger Main View to run `bounce` again (this time with the augmented versions of the libraries).
2. Click `Run` in the `bounce` window and resize the window to make it taller.
3. Click `Select`. This brings up an information dialog and changes the cursor to a plus sign (+). Click `Step` in the `bounce` window as instructed by the dialog. The `Widget` examiner displays the `Step` widget structure.
4. Click the `Tree` tab. The `Tree` examiner panel displays the widget hierarchy of the target object (see Figure 52, page 107).

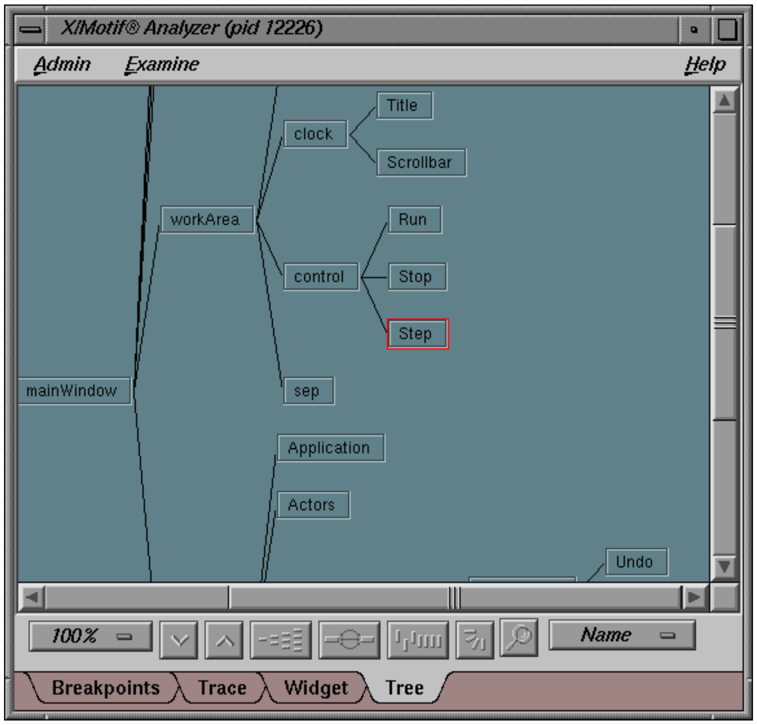


Figure 52. Widget Hierarchy Displayed by the Tree Examiner

5. Double-click the `Run` node in the tree. (`Run` is in the upper-right area of the window). This brings up the widget examiner that displays the `Run` widget structure. Notice that the `Parent` button shows the name of the current widget's parent.

6. In the `X/Motif Analyzer` window, click the `Parent` button to switch the view to the `Run` widget's parent, the `Control` object. The widget examiner now displays the `Control` widget structure. You can navigate through the widget hierarchy using either the `Widget` examiner or the `Tree` examiner.

10.3 Examining Widgets

1. In the widget examiner, pull down the `Children...` menu and select `Run`. The `Run` widget structure is now displayed in the examiner.

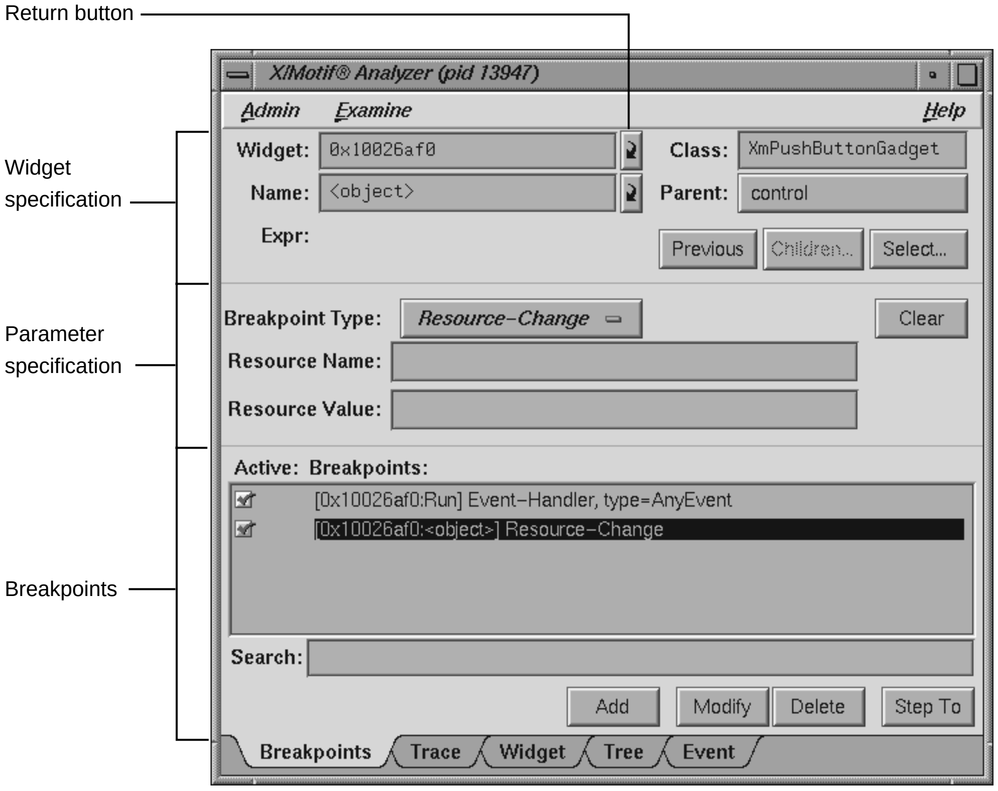
2. In the `bounce` window, pull down the `Actors...` menu and select `Add Red Ball`.

3. In the Debugger Main View, enter **`stop in Clock::timeout`** in the `cvd` command-line area to set a breakpoint in `bounce`. Notice that the `Event` tab (for the event examiner) is added to the tab list.

4. In the Debugger Main View, click `Continue` a few times to observe the behavior of `bounce` with this breakpoint added.

5. Click the `Breakpoints` tab to go to the breakpoints examiner. This examiner allows you to set widget-level breakpoints.

6. In the `Callback Name` text field, enter **`activateCallback`**, then click `Add` to add a breakpoint for the `activateCallback` object of the `Run` widget. The result is displayed in Figure 53, page 108.



Widget: 0x10026af0
Class: XmPushButtonGadget
Name: <object>
Parent: control
Expr:
Breakpoint Type: Resource-Change
Resource Name:
Resource Value:
Active: Breakpoints:
✓ [0x10026af0:Run] Event-Handler, type=AnyEvent
✓ [0x10026af0:<object>] Resource-Change
Search:

Figure 53. Adding a Breakpoint for a Widget

7. In the Debugger Main View, click the breakpoint arrow to remove the `Clock::timeout` breakpoint.

8. In the Debugger Main View, click `Continue`.

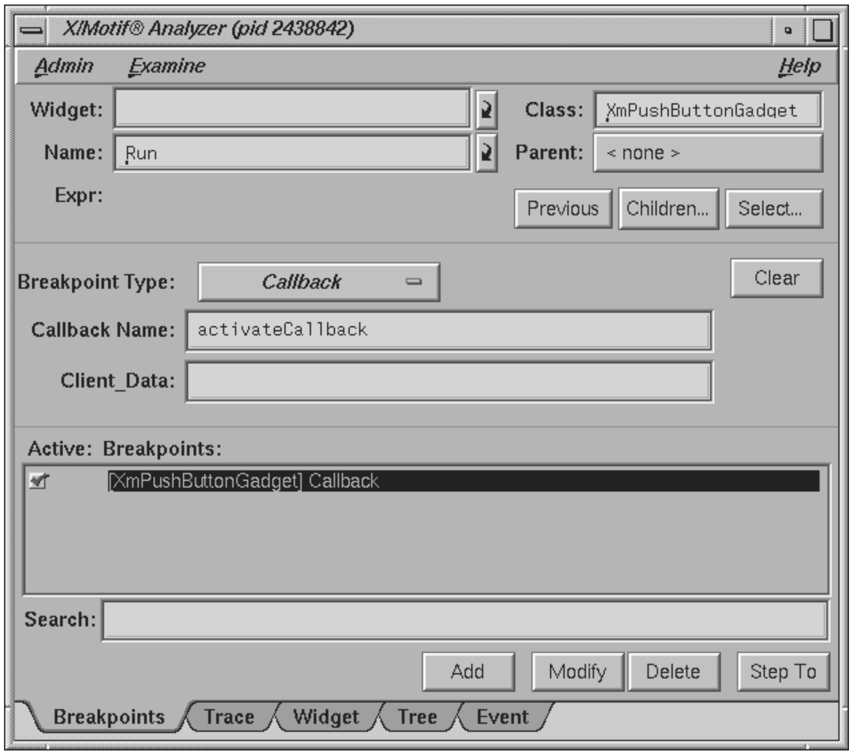
9. In the `bounce` window, click `Stop`.

10. In the `bounce` window, click `Run`. The process stops in the `Run` button's registered `activateCallback`. This is the routine that was passed to `XtAddCallback` routine. Notice that the `Callback` tab (for the callback examiner) is added to the tab list.

10.4 Setting Callback Breakpoints

1. In the `X/Motif Analyzer` window, click the `Breakpoints` list item to highlight the breakpoint.

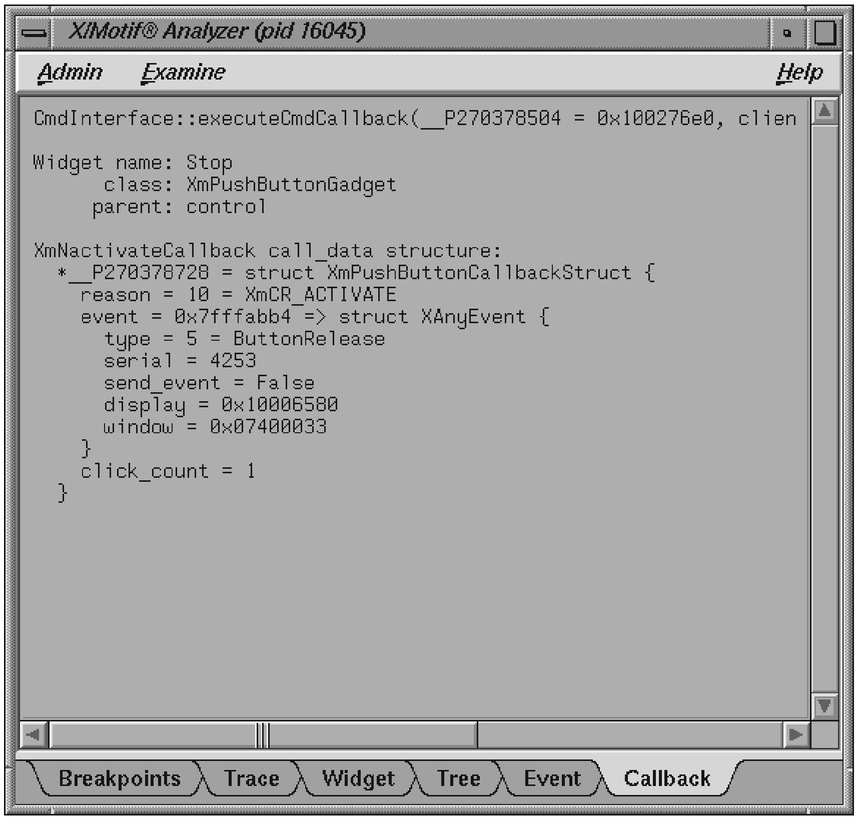
2. In the `X/Motif Analyzer` window, delete the widget address in the `Widget` text field and click `Modify`. This changes the `activateCallback` breakpoint to apply to all push button gadgets (`XmPushButtonGadget`, set in the `Class` text field) rather than just the `Run` button (see Figure 54, page 110).



Name: Run
Class: XmPushButtonGadget
Parent: < none >
Breakpoint Type: Callback
Callback Name: activateCallback
[XmPushButtonGadget] Callback

Figure 54. Setting Breakpoints for a Widget Class

3. In the Debugger Main View, click `Continue`.
4. In the `bounce` window, click `Stop`. The process now stops in the `Stop` button's `activateCallback` routine.
5. In the `X/Motif Analyzer` window, click the `Callback` tab to go to the callback examiner. This examiner displays the callback context and the appropriate `call_data` structure (see Figure 55, page 111).



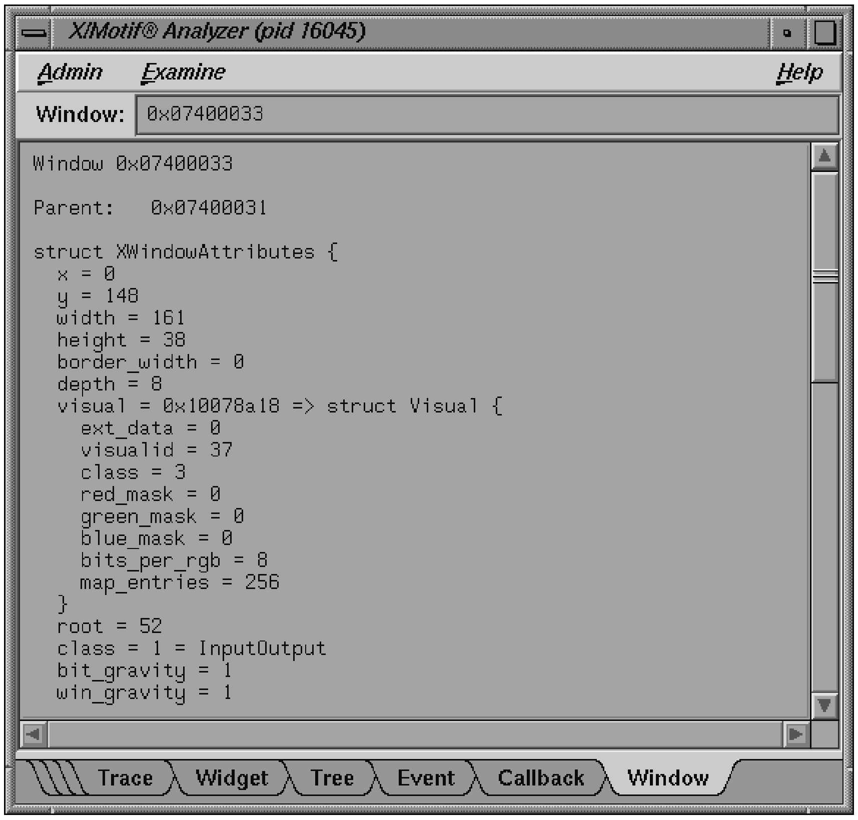
```
CmdInterface::executeCmdCallback(__P270378504 = 0x100276e0, clien

Widget name: Stop
      class: XmPushButtonGadget
     parent: control

XmNactivateCallback call_data structure:
  *__P270378728 = struct XmPushButtonCallbackStruct {
    reason = 10 = XmCR_ACTIVATE
    event = 0x7fffabb4 => struct XAnyEvent {
      type = 5 = ButtonRelease
      serial = 4253
      send_event = False
      display = 0x10006580
      window = 0x07400033
    }
    click_count = 1
  }
```

Figure 55. Callback Context Displayed by the Callback Examiner

6. Double-click the window value in the callback structure, fourth line from bottom.

7. Pull down the `Examine` menu and select `Window`. The X/Motif Analyzer displays the window attributes for that window (the window of the `Stop` button). Notice that the `Window` tab (for the window examiner) is added to the tab list. See Figure 56, page 112.

   You can also accomplish the same action by triple-clicking the window value in the callback structure of the callback examiner. In general, triple-clicking on an address brings you to that object in the appropriate examiner.



```
Window 0x07400033

Parent:   0x07400031

struct XWindowAttributes {
  x = 0
  y = 148
  width = 161
  height = 38
  border_width = 0
  depth = 8
  visual = 0x10078a18 => struct Visual {
    ext_data = 0
    visualid = 37
    class = 3
    red_mask = 0
    green_mask = 0
    blue_mask = 0
    bits_per_rgb = 8
    map_entries = 256
  }
  root = 52
  class = 1 = InputOutput
  bit_gravity = 1
  win_gravity = 1
```

Figure 56. Window Attributes Displayed by the Window Examiner

10.5 Using Additional Features of the Analyzer

1. In the `X/Motif Analyzer` window, click the `Widget` tab.
2. Double-click the `widget_class` value on the fourth line.
3. Pull down the `Examine` menu and select `Widget Class`. The `X/Motif Analyzer` window displays the class record for the `XmPushButtonGadget` routine. Notice that the `Widget Class` tab (for the widget class examiner) is added to the tab list.

   (Again, the same action can be accomplished by triple-clicking the `widget_class` value in the widget examiner.)

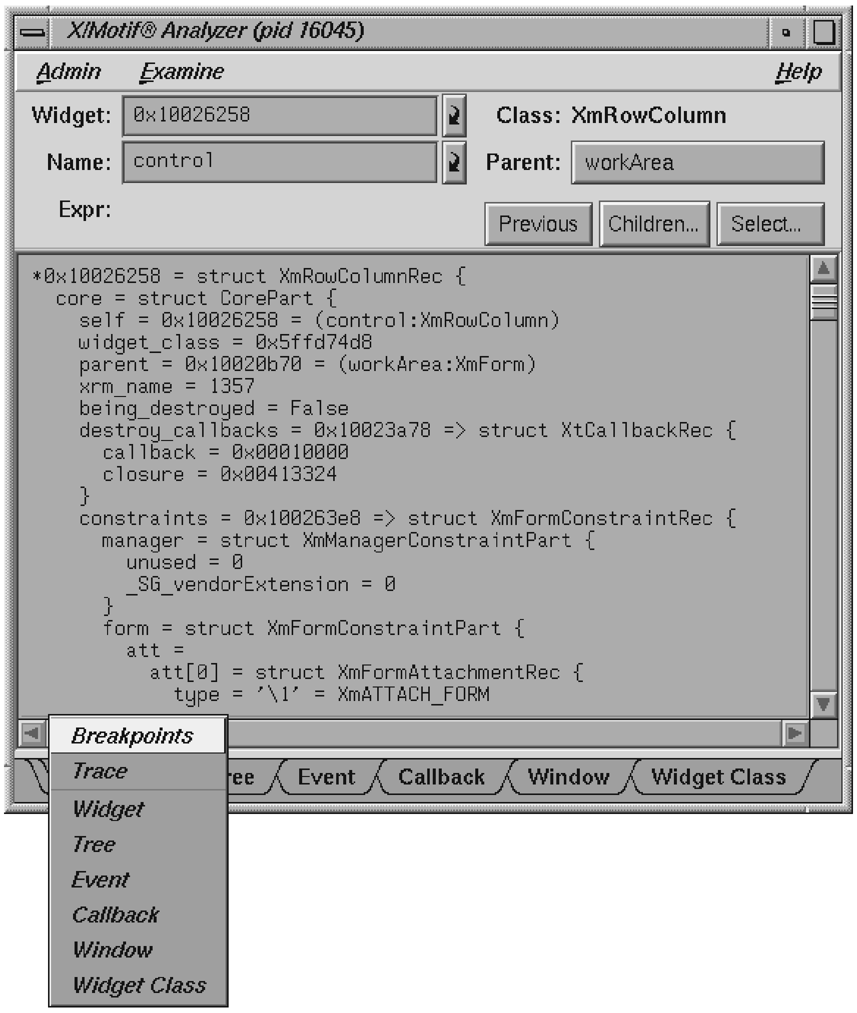
4. Triple-click the superclass value on the third line. The `X/Motif Analyzer` window displays the class record for `XmLabelGadget`, the superclass of `XmPushButtonGadget`. (Triple-clicking is a shortcut for automatically selecting the correct examiner.)

5. Triple-click the superclass value on the third line. The `X/Motif Analyzer` window displays the class record for `XmGadget`, the superclass of `XmLabelGadget`.

6. Click the `Widget` tab to change to the widget examiner.

7. Triple-click the parent value on the fifth line. The `X/Motif Analyzer` window displays the `control` widget, the parent of `Run`. This action produces the same results as clicking the `Parent` button.

8. In the `X/Motif Analyzer` window, click the tab overflow area (the area where the tabs overlap, to the far left of the tab list) and select the `Breakpoints` tab (see Figure 57).



Widget: 0x10026258
Name: control
Parent: workArea

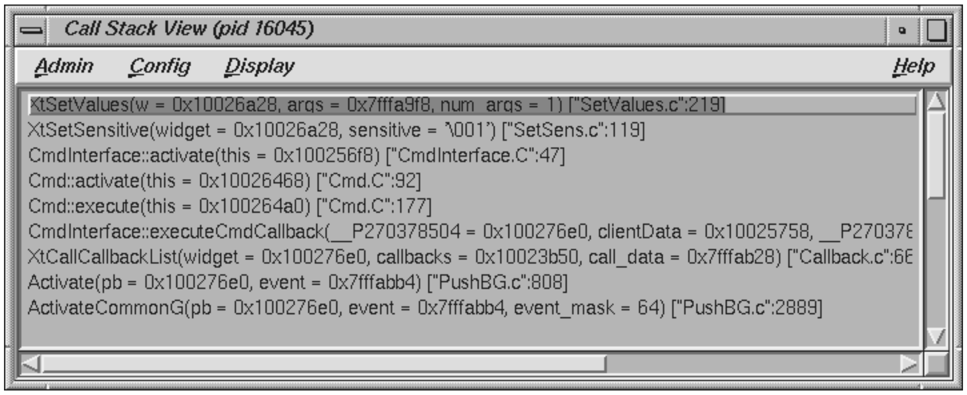
```
*0x10026258 = struct XmRowColumnRec {
  core = struct CorePart {
    self = 0x10026258 = (control:XmRowColumn)
    widget_class = 0x5ffd74d8
    parent = 0x10020b70 = (workArea:XmForm)
    xrm_name = 1357
    being_destroyed = False
    destroy_callbacks = 0x10023a78 => struct XtCallbackRec {
      callback = 0x00010000
      closure = 0x00413324
    }
    constraints = 0x100263e8 => struct XmFormConstraintRec {
      manager = struct XmManagerConstraintPart {
        unused = 0
        _SG_vendorExtension = 0
      }
      form = struct XmFormConstraintPart {
        att =
          att[0] = struct XmFormAttachmentRec {
            type = '\1' = XmATTACH_FORM
```

Figure 57. Selecting the Breakpoints Tab From the Overflow Area

9. Change the breakpoint type from `Callback` to `Resource-Change`.
10. In the Class text field, enter: **`Any`**.
11. In the `Resource Name` text field, enter: **`sensitive`**.
12. Click `Add`. This adds a breakpoint.

13. In the Debugger Main View, click `Continue`. The resource change breakpoint was reached, stopping the process in the `XtSetValues` routine.

14. In the Debugger Main View, pull down the `Views` menu and select `Call Stack`. Notice the call to `XtSetValues` on the second line (see Figure 58, page 115).



```
XtSetValues(w = 0x10026a28, args = 0x7fffa9f8, num_args = 1) ["SetValues.c":219]
XtSetSensitive(widget = 0x10026a28, sensitive = '\001') ["SetSens.c":119]
CmdInterface::activate(this = 0x100256f8) ["CmdInterface.C":47]
Cmd::activate(this = 0x10026468) ["Cmd.C":92]
Cmd::execute(this = 0x100264a0) ["Cmd.C":177]
CmdInterface::executeCmdCallback(__P270378504 = 0x100276e0, clientData = 0x10025758, __P27037
XtCallCallbackList(widget = 0x100276e0, callbacks = 0x10023b50, call_data = 0x7fffab28) ["Callback.c":6
Activate(pb = 0x100276e0, event = 0x7fffabb4) ["PushBG.c":808]
ActivateCommonG(pb = 0x100276e0, event = 0x7fffabb4, event_mask = 64) ["PushBG.c":2889]
```

Figure 58. Breakpoint Results Displayed by the `Call Stack View`

15. In the `Call Stack View`, double-click the `Cmdinterface::activate` frame (just below `XtSetSensitive`). This is where the sensitive resource was changed.

16. In the `X/Motif Analyzer` window, click the `Widget` tab.

17. In the `X/Motif Analyzer` window, double-click the widget address in the `Widget` text field, press backspace, enter `_w`, and press **`Enter`**. The X/Motif Analyzer displays the `Run` widget, which is the widget currently being changed.

18. In the Debugger Main View, click `Continue`. The process stops again in the `XtSetValues` routine, which is another sensitivity change.

19. Double-click the `Cmdinterface::active` frame (just below `XtSetSensitive`).

20. Double-click in `Widget` field, press backspace, enter `_w`, and press **`Enter`**. The `X/Motif Analyzer` window displays the `Step` widget, which is the widget now being changed.

10.6 Ending the Session

Exit the X/Motif Analyzer by pulling down the `Admin` menu and choosing `Close`. Exit the Debugger by pulling down the `Admin` menu and choosing `Exit`.

Note: If you exit the Debugger, you automatically exit the X/Motif Analyzer.

Debugger Reference [A]

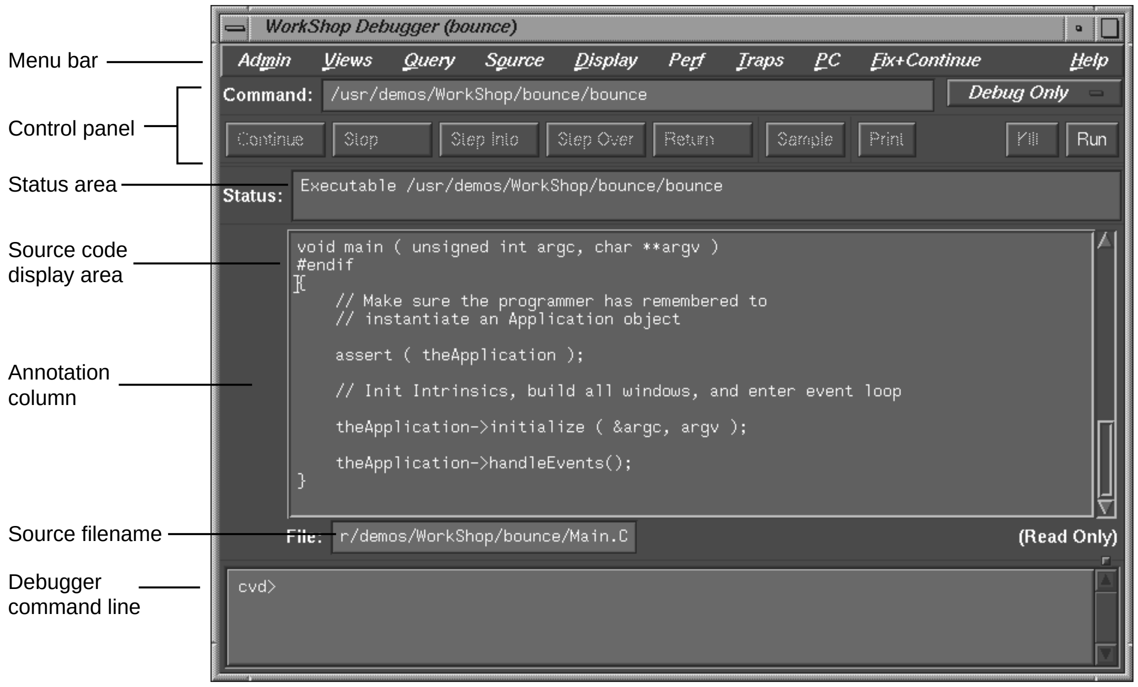
This chapter describes the function of each window, menu, and display in the Debugger's graphical user interface and describes the commands available on the Debugger command line (see Section A.10, page 244).

This chapter contains the following sections:

- Main Window, Section A.1, page 117
- Basic Windows, Section A.2, page 140
- Ada-Specific Windows, Section A.3, page 149
- X/Motif Analyzer Windows, Section A.4, page 155
- Trap Management Windows, Section A.6, page 186
- Data Examination Windows, Section A.7, page 190
- Machine-Level Debugging Windows, Section A.8, page 221
- Fix+Continue Windows, Section A.9, page 232
- Debugger Command Line, Section A.10, page 244

A.1 Main View

The major areas of the Main View window are shown in Figure 59, page 118.



Command: /usr/demos/WorkShop/bounce/bounce

Status: Executable /usr/demos/WorkShop/bounce/bounce

```
void main ( unsigned int argc, char **argv )
#endif
{
    // Make sure the programmer has remembered to
    // instantiate an Application object

    assert ( theApplication );

    // Init Intrinsics, build all windows, and enter event loop

    theApplication->initialize ( &argc, argv );

    theApplication->handleEvents();
}
```

File: r/demos/WorkShop/bounce/Main.C

```
cvd>
```

Figure 59. Major Areas of the Main View Window

The Main View contains a menu bar, from which you can perform a number of functions and launch windows. The menu bar contains the following menus, which are discussed in detail in later pages:

- `Admin`, Section A.1.1, page 123
- `Views`, Section A.1.2, page 126
- `Query`, Section A.1.3, page 127
- `Source`, Section A.1.4, page 128
- `Display`, Section A.1.5, page 130
- `Perf`, Section A.1.6, page 131
- `Traps`, Section A.1.7, page 133
- `PC`, Section A.1.8, page 134

- `Fix+Continue`, Section A.1.9, page 134
- `Help`, Section A.1.10, page 139

Main View also contains several input fields, a source code display area, and buttons that trigger commonly used actions.

In Main View, actions can be applied to a single process or to all processes. To force an action to apply to all processes, type the command on the Debugger command line or click the right mouse button over the `Run`, `Continue`, or `Stop` button to set the *all* mode.

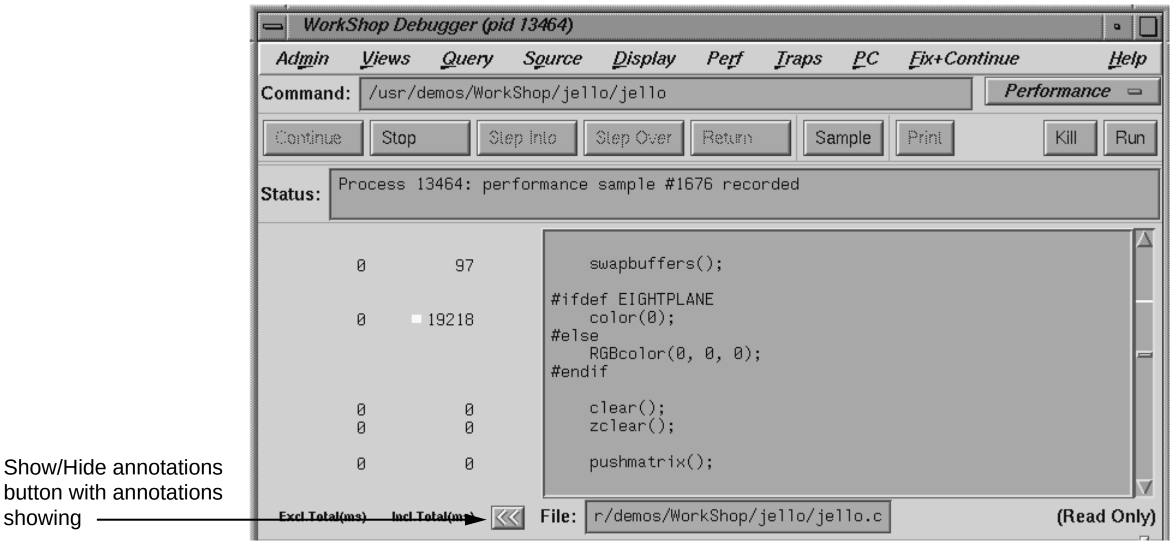
The Main View contains the following items:

`Command` text field	Displays full pathname of the executable file that you are currently debugging.
Debug option menu	Allows you to conduct performance experiments using the WorkShop performance tools. The following menu choices are available: • `Debug Only` runs the Debugger in Debug mode with no performance tools enabled. • `Performance` causes performance data to be gathered and instrumented code to be generated for performance analysis while using the Debugger. • `Purify` activates the Purify memory corruption analysis tool. The code displayed in Main View, `Source View`, and so forth will be code generated by Purify. (This option appears only if Purify is installed on your system. Purify is not a Silicon Graphics product nor is it part of the WorkShop package. It is a product of Rational Software and is not orderable from nor supported by SGI.)
`Continue` [`all`]	Continues execution of the current process or all processes. This command can be run only after the running process(es) has stopped. If the program has not been run or has been killed, the `Continue` button is grayed out. If the target program has not yet started executing, use the `Run` command to start execution.

`Stop` [`all`]	Stops execution of the current process or all processes while it is running. This command is valid only when a process(es) is running; otherwise the command button is grayed out. Traps can also be planted to stop the program at a specific location or on a particular condition.
`Step Into`	Executes a source line single step of the current process. If a function call is encountered, it is stepped into. That is, the current process continues to the next source statement, even if that statement is encountered in a function that is called. The `Step Over` command can be used to step over function calls, then stop. If a trap is encountered while executing `Step Into`, the command is canceled and the process is stopped where the trap was fired. This command can be run only after the running process(es) has stopped; otherwise the command button is grayed out. When you press the right mouse button over the `Step Into` button, a menu pops up to allow you to choose the number of source lines to be stepped. If you choose the `N...` menu entry, a dialog window is opened to allow you to enter a step value.
`Step Over`	Executes source line single step of the current process. If a function call is encountered, it is stepped over. That is, the current process continues to the next source statement, but does not count statements in functions that are called while stepping. `Step Into` can be used to step into function calls, then stop. If a trap is encountered while executing `Step Over`, the command is canceled and the process is stopped where the trap was fired. When you press the right mouse button over `Step Over`, a menu pops up to allow you to choose the number of source lines to be stepped. If you choose the `N...` menu entry, a dialog window is opened to allow you to enter a step value.

`Return`	Continues execution of the process until the current function that is being executed returns. The process is stopped immediately upon returning to the calling function. All code within the current function is executed as usual. If a trap is encountered while executing the `Return` command, the command is canceled and the process is stopped where the trap was fired. This command can be run only after the running process(es) has stopped; otherwise the command button is grayed out. This command is not allowed if the executable is instrumented for performance analysis.
`Sample`	Allows you to manually sample the state of a process for evaluation by the Performance Analyzer. This command can be run only when the process(es) is running and the `Enable Data Collection` mode is set on the `Performance` panel; otherwise the command button is grayed out.
`Print`	Prints the value of the currently selected expression, which is highlighted in `Source View`.
`Kill` [`all`]	Kills the currently running process or all running processes that you are debugging by sending it the equivalent of a `kill -9` signal. This command can be run if the process(es) is running or stopped; otherwise the command button is grayed out.
`Run` [`all`]	Runs the program that you are currently debugging or all programs. After the initial run, allows you to rerun the program(s) while maintaining the traps you have set.
`Status:` area	Displays information about the process that you are debugging.
Source Code area	Displays the source code that your are currently debugging.

Annotation column	Displays information specific to a line number, such as breakpoints, location of the PC, and so forth.
`File:` text field	Displays the name of the file shown in the source code area.
Command line area	Area of the Main View where you can enter Debugger commands.
Show / Hide annotations button	This button (see Figure 60, page 123) is visible only when you run or load a performance experiment (see the *Developer Magic: Performance Analyzer User's Guide* for more information on the performance tools). This is a toggle button that shows or hides performance related annotations.

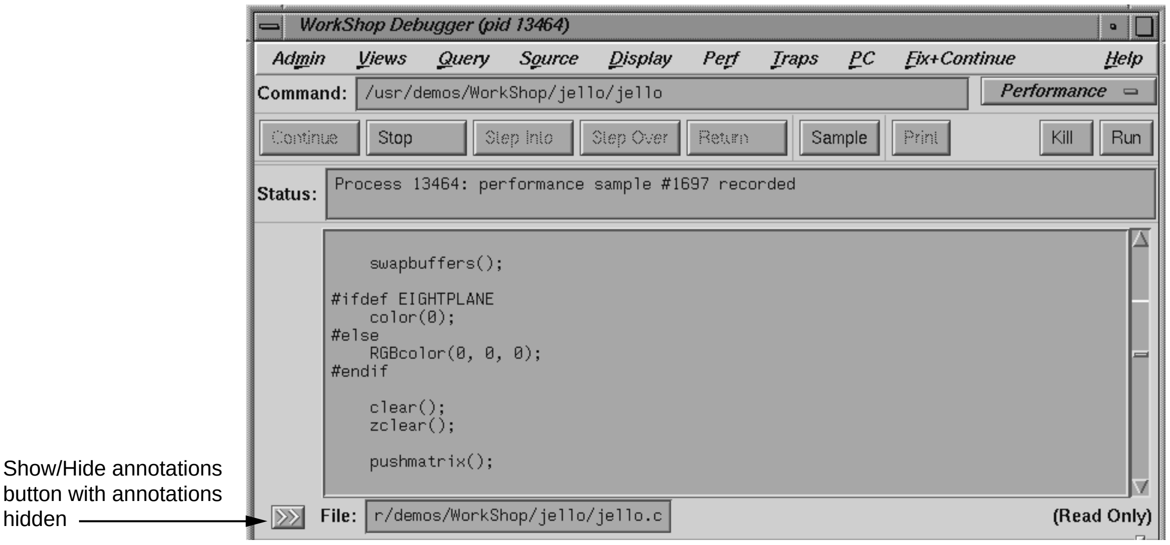


Command: /usr/demos/WorkShop/jello/jello

Status: Process 13464: performance sample #1676 recorded

```
      0        97      swapbuffers();

                   #ifdef EIGHTPLANE
      0     19218      color(0);
                   #else
                       RGBcolor(0, 0, 0);
                   #endif

      0         0      clear();
      0         0      zclear();

      0         0      pushmatrix();
```

File: r/demos/WorkShop/jello/jello.c



Command: /usr/demos/WorkShop/jello/jello

Status: Process 13464: performance sample #1697 recorded

```
    swapbuffers();

#ifdef EIGHTPLANE
    color(0);
#else
    RGBcolor(0, 0, 0);
#endif

    clear();
    zclear();

    pushmatrix();
```

File: r/demos/WorkShop/jello/jello.c

Figure 60. Show / Hide Annotations Button in Main View

A.1.1 `Admin` Menu

The `Admin` menu in Main View performs general management functions dealing with processes, windows, and user preferences. The `Admin` menu provides the following selections:

`Library Search Path...`	Controls where the Debugger looks for DSOs when you invoke the Debugger on an executable or core file. The `Library Search Path` dialog allows you to reset the `LD_LIBRARY_PATH` and `_RLD_ROOT` environment variables. You can also reset `_RLD_LIST` to control the set of DSOs that will be used by the program. See the `rld`(1) man page for more information on these variables. Any changes you make to these variables are propagated into `Execution View` when you run the program. The `Library Search Path` dialog opens automatically if you invoke the Debugger on an executable or core file and the Debugger is unable to find all of the required DSOs. You can also open the `Library Search Path` dialog by selecting `Library Search Path...` from the `Admin` Menu. The list of required DSOs displays at the top of the dialog box, annotated by the status of each DSO. The status can be `OK`, `Error: Cannot find library`, or `Error:  Core file and library mismatch`. The status `Error:  Core file and library mismatch` indicates that the Debugger found a DSO that did not match the core file. There are three fields for the variables below the list area where you can modify their values. `Insert Before` and `Insert After` move the shared object specified in the `Value` field before or after the selected object in the list. `Modify` replaces the selected object in the list with the file entered in the `Value` field. `Remove` deletes the selected shared object from the list.
`Multiprocess View...`	Displays the `Multiprocess View` window, which allows you to control processes and threads. You should note that if you exit from `Multiprocess View`, you will exit from your debugging session.
`GLdebug`	Provides a toggle to turn on `GLdebug`. `GLdebug` is a graphical software tool for debugging

	application programs that use the IRIS Graphics Library (GL). `GLdebug` locates programming errors in executables when GL calls are used incorrectly. For more information, refer to the *GLdebug User's Guide*.
`Switch Process...`	Changes the current process. You will be queried for the new process ID. You can type it in or paste it from another window, if desired. Switching processes changes the session.
`Switch Executable...`	Changes the current executable. This option also lets you debug a different core file.
`Detach`	Releases the process from the Debugger. This allows you to make changes to the source code. You must detach the process before you recompile the program.
`Load Settings...`	Allows you to use the previously saved preference settings to an initialization file used when the Debugger is first started.
`Save Settings...`	Allows you to save the current preference settings to an initialization file used when the Debugger is first started. These can include such items as window sizes, current views, window configurations, and so on.
`Iconify`	Iconifies all session views.
`Raise`	Brings all session view windows to the foreground and redisplays any iconified windows.
`Launch Tool`	Lets you run other WorkShop tools. You can switch to the other tools by selecting `Build Manager`, `Static Analyzer`, `Performance Analyzer`, or `Tester`. Selecting `Debugger` lets you start another debugging session. If you have WorkShop ProMPF installed on your system, the `Parallel Analyzer` selection is also available.

`Exit`	Exits all views in the session and terminates the session.

A.1.2 Views Menu

The `Views` menu in Main View provides the following selections for viewing the process(es) and their corresponding data:

`Array Browser`	Displays values from an array or array-slice in a two-dimensional spreadsheet and optionally in a three-dimensional representation; that is, a bar graph, surface, multiple lines, or points in space. These help you pick out bad data more readily. Arrays can contain up to 100 x 100 elements.
`Call Stack`	Displays the call stack along with parameters to the calls. If you double-click an entry in the stack, you switch the current context to that entry and you can check the state of variables.
`Disassembly View`	Displays assembly code corresponding to the source code.
`Exception View`	Displays the `Exception View`, an Ada-specific window used for exception handling.
`Execution View`	Displays the `Execution View` window for handling the target process's input and output.
`Expression View`	Evaluates expressions in Fortran, C, or C++. To enter an expression, select it in the source code display and paste it into the `Expression View` field, using the middle mouse button.
`File Browser`	Displays a scrollable list of source files used by the current executable. Double-click a file in the list to load it directly into the source display area in Main View or Source View. The `Search field` lets you find files in the list quickly.
`Memory View`	Displays the value at a given memory address.
`Process Meter`	Monitors the resource usage of a running process without saving the data. (Used with the Performance Analyzer.)

`Register View`	Displays the values stored in the hardware registers for the target process.
`Signal Panel`	Displays the signals that can occur. You can specify which signals trigger traps and which are to be ignored.
`Source View`	Displays source code. Lets you set traps, perform searches, and inspect source code without losing information in Main View.
`Structure Browser`	Displays data structures in a graphical format. You can dereference pointers by double-clicking.
`Syscall Panel`	Lets you set traps at the entry to or exit from system calls.
`Task View`	Brings up the `Task View`, an Ada-specific view that provides task and callstack information for processes.
`Trap Manager`	Allows you to set, edit, and manage traps. The Trap Manager is used by both the Debugger and the Performance Analyzer.)
`Variable Browser`	Displays values of local variables and parameters for the current context.
`X/Motif Analyzer`	Provides you with specific debugging support for X/Motif applications. There are various examiners for different X/Motif objects, such as widgets and X graphics contexts, that might be difficult or impossible to inspect using ordinary Debugger functionality.

A.1.3 `Query` Menu

The `Query` menu lets you perform some of the queries available in the Static Analyzer. If you have previously built a `cvstatic` fileset, this is rather convenient. However, if you need to build the fileset from scratch, the process becomes more involved.

With a current fileset, you can double-click any defined entity in the source code, select the `Where Defined?` option appropriate to its type, and the source code display area will scroll to the location where the item is defined.

A.1.4 Source Menu

The `Source` menu in Main View provides the following selections to manage source code files:

`Open...`	Loads a source file.
`Save`	Records changes made during the debugging session to the source file. You must first select `Make Editable`, which appears in the `Source` menu when the file is read-only.
`Save As...`	Records changes made during the debugging session to the source file under a different filename. . You must first select `Make Editable`, which appears in the `Source` menu when the file is read-only.
`Save As Text...`	Records the information in the display area as a text file.
`Insert Source...`	Inserts the text of a file within your current file.
`Fork Editor`	Starts your default editor on the current file. The default editor is determined by the `editorCommand` resource in the `app-defaults` file. The value of this resource defaults to `wsh -c vi +%d,` which means run `vi` in a `wsh` window and scroll to the current line. If the editor lets you specify a starting line, enter **`%d`** in the resource to indicate the new line number.
`Recompile`	Displays the `Build View` window, which lets you compile the source code associated with the current executable.
`Make Read Only` / `Make Editable`	Toggles the source code displayed between read-only and writable states so that you can edit your code.
`Search...`	Searches for a literal case-sensitive, literal case-insensitive, or regular expression (see Figure 61, page 129). After you have set your target and clicked `Apply` (or pressed `Enter`), each instance is marked by a search target indicator in the scroll bar. You can search forward or backward in the file by clicking the `Next` and `Prev` buttons. You can also click an indicator with the middle mouse

button to scroll Main View to that point. Clicking `Reset` removes the search target indicators.

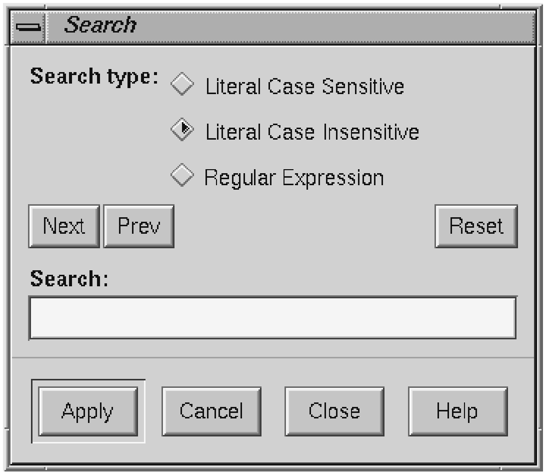


Figure 61. The `Search` Dialog

`Go to Line...` Lets you scroll to a position in the source code by specifying a line number. `Go to Line...` brings up a dialog box similar to the one shown in Figure 62, page 129.

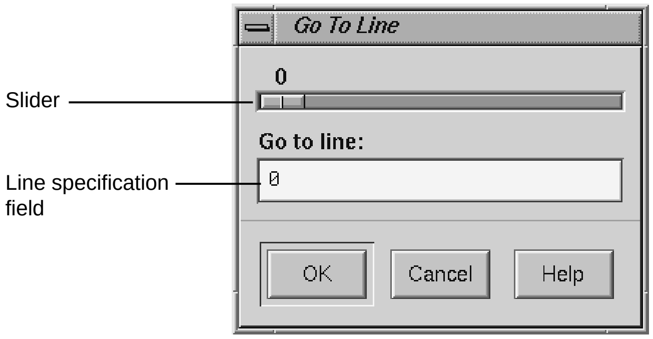


Figure 62. `Go to Line` Dialog

	You can enter a line number or use the slider at the top of the box to select a line number. You do not have to display line numbers to use this feature.
`Versioning`	Provides access to the configuration management tool, if you have designated one. The `cvconfig` script lets you designate ClearCase, RCS, or SCCS. Type the following: **`cvconfig [clearcase	rcs	sccs]`**  You must have `root` permissions to run `cvconfig`.  The `Versioning` submenu appears.  Selecting any submenu option displays a shell in which you can access the configuration management tool. The following selections are available on the submenu:  • `CheckIn` — Saves the source file and checks it into the database as a new version.  • `CheckOut` — Recalls the source file from the tool's database if you have the proper authority, locks it, and makes it editable.  • `UncheckOut` — Cancels the checkout, with no changes registered.

A.1.5 `Display` Menu

The `Display` menu in Main View provides the following selections to annotate the displayed source code:

`Show Line Numbers/Hide Line Numbers`	Displays or hides line numbers in the annotation column corresponding to the source code.
`Preferences...`	Displays the `Annotations Preferences` dialog box (see Figure 63, page 131), which lets you show or hide column annotations and menus specific to the different WorkShop tools. In the Debugger, you can display trap, PC, and context

icons. If you have purchased WorkShop Pro MPF, you can display and manipulate loop indicators. The `Performance` toggle displays experiment statistics. The Tester module lets you see coverage statistics. Turning off the `Performance` toggle deletes the performance annotations from the `Source View`.

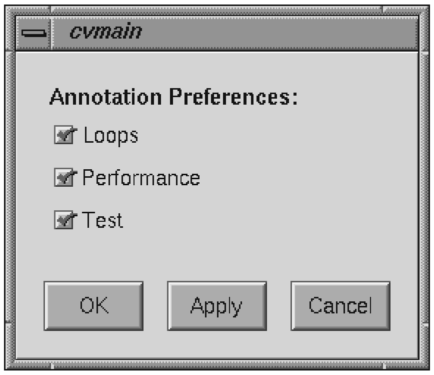


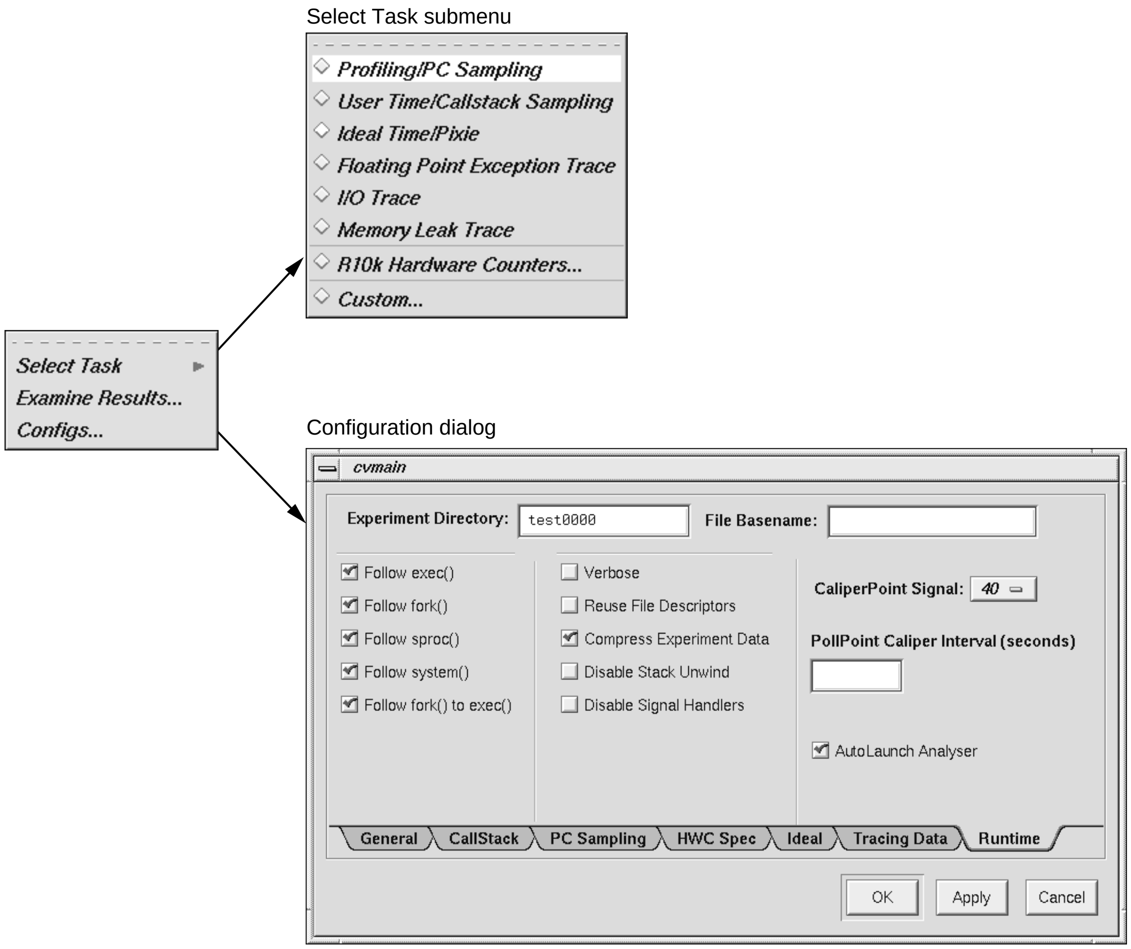
Figure 63. Preferences Dialog

`Hide Icons/Show Icons`	Hides or displays the annotation column next to the source code display area.

A.1.6 `Perf` Menu

The `Perf` (Performance) menu (see Figure 64, page 132) offers the following menu selections:

`Select Task` submenu	Allows you to choose the task for your performance analysis. The choices available are shown in Figure 64, page 132. You may select only one task per performance analysis run. If none of the given tasks satisfy your requirements, you can choose `Custom...`, which brings up the configuration dialog open to the `General` tab. From here, you can design your own task requirements.



Experiment Directory: test0000
File Basename:

[✓] Follow exec()
[✓] Follow fork()
[✓] Follow sproc()
[✓] Follow system()
[✓] Follow fork() to exec()

[ ] Verbose
[ ] Reuse File Descriptors
[✓] Compress Experiment Data
[ ] Disable Stack Unwind
[ ] Disable Signal Handlers

CaliperPoint Signal: 40
PollPoint Caliper Interval (seconds)
[✓] AutoLaunch Analyser

Figure 64. `Perf` Menu and Subwindows

`Examine Results...`	Launches the Performance Analyzer. For complete information about the Performance Analyzer, see the *Developer Magic: Performance Analyzer User's Guide*.
`Configs...`	Brings up the configuration dialog open to the `Runtime` tab as shown in Figure 64, page 132. The dialog opens with the `Experiment Directory` text field filled in with a default

value. The Performance Analyzer provides a default directory named `test0000`. If you use the default or any other name that ends in four digits, the four digits are used as a counter and will be incremented automatically for each subsequent experiment

A.1.7 `Traps` Menu

The `Set Trap` submenu contains selections that allow you to set various types of traps. The following submenu selections are available:

`Stop`	Sets a breakpoint at a designated line in your source code. To set a breakpoint at a line displayed in Main View or `Source View`, highlight the the appropriate line in the source code display area and select the `Set Trap` submenu, then choose the `Stop` option.
`Stop At Function Entry`	Sets a breakpoint at the beginning of a function. To set a breakpoint at a function, highlight the function name in the source code display area and select the `Set Trap` submenu, then choose the `Stop At Function Entry` option.
`Stop At Function Exit`	Sets a breakpoint at the end of a function. To set a breakpoint at a function exit, highlight the function name in the source code display area and select the `Set Trap` submenu, then choose the `Stop At Function Exit` option.
`Sample`	Sets a sample trap at a designated line in your source code to collect performance data. To set a trap, highlight the appropriate line in the source code display area, pull down the `Set Trap` submenu, then select the `Sample` option.
`Sample At Function Entry`	Sets a sample trap at the beginning of a function. To set a trap, highlight the function name in the source code display area and select `Set Trap`, then `Sample At Function Entry`.
`Sample At Function Exit`	Sets a sample trap at the end of a function. To set, highlight the function name in the source

code display area and select `Set Trap`, then `Sample At Function Exit`.

The `Clear Trap` menu contains selections that allow you to delete a trap on the line containing the cursor. You must designate `Stop` or `Sample` trap type, since both types can exist at the same location, appearing superimposed on each other. The following submenu selections are available:

`Stop`	Designates the stop trap type.
`Sample`	Designates the sample trap type.

A.1.8 PC Menu

The PC (program counter) menu in Main View provides the following selections for controlling the execution of a process:

`Continue To`	Continues the process to the selected point in the program unless some other event interrupts. You select a line by placing the cursor in it.
`Jump To`	Goes directly to a selected point within the same function, jumping over intervening code. Waits for command to resume execution. You select a line by placing the cursor in it.

A.1.9 `Fix+Continue` Menu

The `Fix+Continue` menu offers the following menu selections:

`Edit`	Allows you to edit functions using the Debugger editor.
`External Edit`	Allows you to edit functions using an external editor. The default editor is `vi`, but can be changed by using the `Set Edit Tool...` pop-up menu in the `Admin` menu of the `Status` window. See Section A.9.1, page 234, for further information.
`Parse and Load`	Compiles your modified function and loads it for execution. You can execute the modified function by clicking on the `Run` or `Continue` buttons in the Debugger main view.

`Show Difference` submenu	Allows you to see the difference between the original code and your modifications. See Section A.1.9.1, page 135, for further information.
`Edited<-->Compiled`	Enables or disables your changes. This switch allows you to see how your application executed before and after the changes you made.
`Save As...`	Allows you to save your changes to a file. You can save changes to the current source file (the default) or to a separate file.
`Save All Files...`	Launches the `Save File+Fixes As...` dialog that allows you to update the current session, saving all the modified functions to the appropriate files.
`View` submenu	Allows you to change to different views. Fix+Continue supports status, message, and build environment windows. See Section A.1.9.2, page 136, for further information.
`Preferences` submenu	Allows you to set your Fix+Continue preferences. See Section A.1.9.3, page 136, for further information.
`Cancel Edit`	Takes you out of edit mode and cancels any changes you have made.
`Delete Edits`	Deletes any changes that you made to functions.

A.1.9.1 `Show Difference` Submenu

This submenu allows you to view differences between your original and your modified code. It contains the following options:

`For Function`	Opens a window that shows you the differences between the original function source and your modified source.
`For File`	Opens a window that shows you the differences between the original source file and your modified version.
`Set Diff Tool ...`	Launches the `Fix+Continue Preferences Dialog` (see Figure 65, page 137) that allows you to set the tool that displays code differences. The default is `xdiff`(1). For further information on

the `Fix+Continue Preferences Dialog`, see Section A.1.9.3, page 136.

A.1.9.2 `View` Submenu

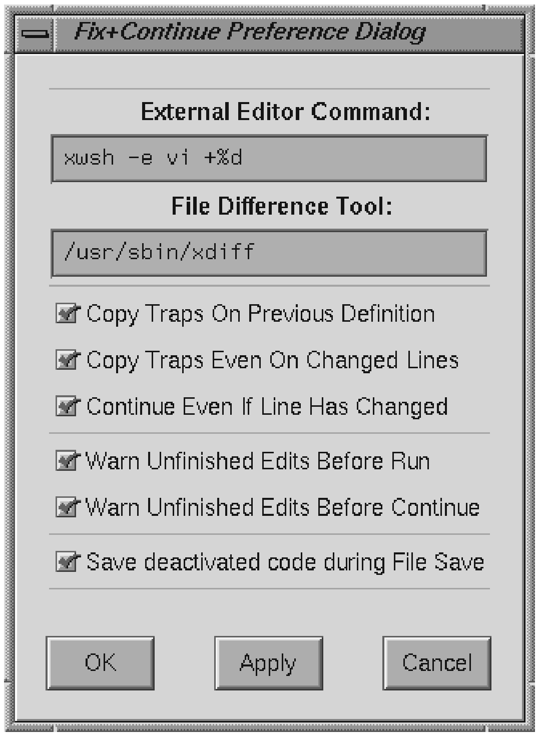
This submenu allows you to open different Fix+Continue view windows. It contains the following options:

`Status Window`	Launches the `Fix+Continue Status` window. See Section A.9.1, page 234, for more information.
`Message Window`	Launches the `Fix+Continue Message` window. See Section A.9.2, page 238, for more information.
`Build Environment Window`	Launches the `Fix+Continue Build Environment` window. See Section A.9.3, page 239, for more information.

A.1.9.3 `Preferences` Submenu

This submenu allows you to set various options for the Fix+Continue environment, such as the difference tool, the external editor command, and so on. The menu contains the following options:

`Show Preferences`	Launches the `Fix+Continue Preference Dialog` (see Figure 65, page 137) that displays preferences currently enabled for the session, and allows you to change the settings.



External Editor Command:

xwsh -e vi +%d

File Difference Tool:

/usr/sbin/xdiff

✓ Copy Traps On Previous Definition

✓ Copy Traps Even On Changed Lines

✓ Continue Even If Line Has Changed

✓ Warn Unfinished Edits Before Run

✓ Warn Unfinished Edits Before Continue

✓ Save deactivated code during File Save

Figure 65. `Fix+Continue Preferences Dialog`

The following preferences are available through the dialog:

- `External Editor Command:` text field that allows you to choose your text editor. The default is `vi`.
- `File Difference Tool:` text field that allows you to choose the tool to use when comparing code. The default is `xdiff`.
- `Copy Traps On Previous Definition` toggle. When you edit and parse a function, Fix+Continue copies traps from the old definition to the new one by mapping old lines to new lines. (This mapping is the same as what can be generated using the UNIX

`diff` utility.) If `Copy Traps On Previous Definition` is on and the mapped line the new definition is modified, then Fix+Continue will look at the switch.

- `Copy Traps Even On Changed Lines` toggle that causes the debugger to copy traps onto a mapped line.
- `Continue Even If Line Has Changed` toggle. When you edit and compile a function in which your program is currently stopped, Fix+Continue can continue in the new definition provided some conditions are satisfied. The line from which the program continues depending on the mapping from the line in which it stopped. In case it can continue in the new definition from a line which you have modified, Fix+Continue consults this toggle to determine whether to continue in the new or old definition. This toggle allows you to override the default behavior.
- `Warn Unfinished Edits Before Run` toggle that pops up a warning dialog before a run if you have unfinished edits.
- `Warn Unfinished Edits Before Continue` toggle that pops up a warning dialog before a continue if you have unfinished edits.
- `Save deactivated code during File Save` toggle. The Fix+Continue file save substitutes new definitions in place of old ones. If you want to save your original functions in the same file, this switch allows you to save the old (original or compiled) code under an `#ifdef`. When you compile, the old code will not get compiled. You can manually edit the source to use the old definition in any way you desire.

`Reset Factory Defaults`	Sets preferences to the installed defaults.
`Save Preferences`	Allows you to save your preferences to a file. This item brings up the `File` dialog.
`Load Preferences...`	Allows you to load preferences from a file. This item brings up the `File` dialog.

A.1.9.4 Keyboard Accelerators

Use the accelerators in Table 4, page 139, to issue Fix+Continue commands directly from the keyboard. The accelerators are listed alphabetically by command.

Table 4. Fix+Continue Keyboard Accelerators

Command	Key Sequence
Cancel Edit	`Ctrl-u`
Edit	`Ctrl-e`
External Edit	`Ctrl-x`
Parse And Load	`Ctrl-p`

A.1.10 `Help` Menu

The `Help` menu provides the following options:

`Click for Help`	Provides information on the selected window or menu.
`Overview`	Provides general information on the current tool.
`Index...`	Displays the entire list of help topics, alphabetically, hierarchically, or graphically.
`Keys & Shortcuts`	Lists the keys and shortcuts for the current tool.

`Product Information`	Provides copyright and version number information on the tool.

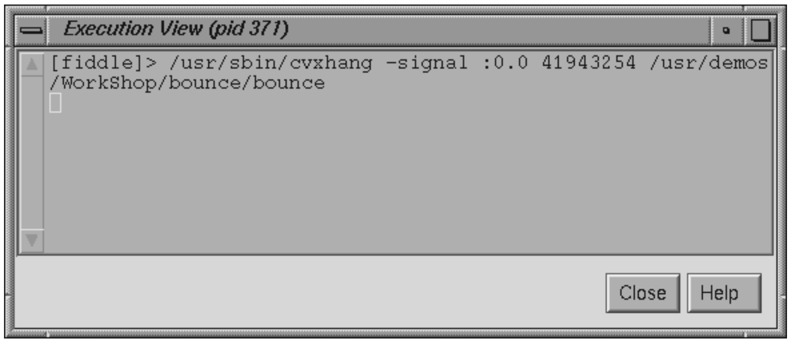
A.2 Basic Windows

This section discusses additional views available through the Debugger: the `Execution View`, `Multiprocess View`, `Source View`, and `Process Meter`.

A.2.1 Execution View

The `Execution View` window is a simple shell that lets you set environment variables and inspect error messages. Your target program I/O, if any, will be displayed in the `Execution View` window. If the program is I/O-based, then all interaction takes place in `Execution View`.

The `Execution View` (see Figure 66, page 140) is launched automatically with the Debugger.



```
[fiddle]> /usr/sbin/cvxhang -signal :0.0 41943254 /usr/demos
/WorkShop/bounce/bounce
```

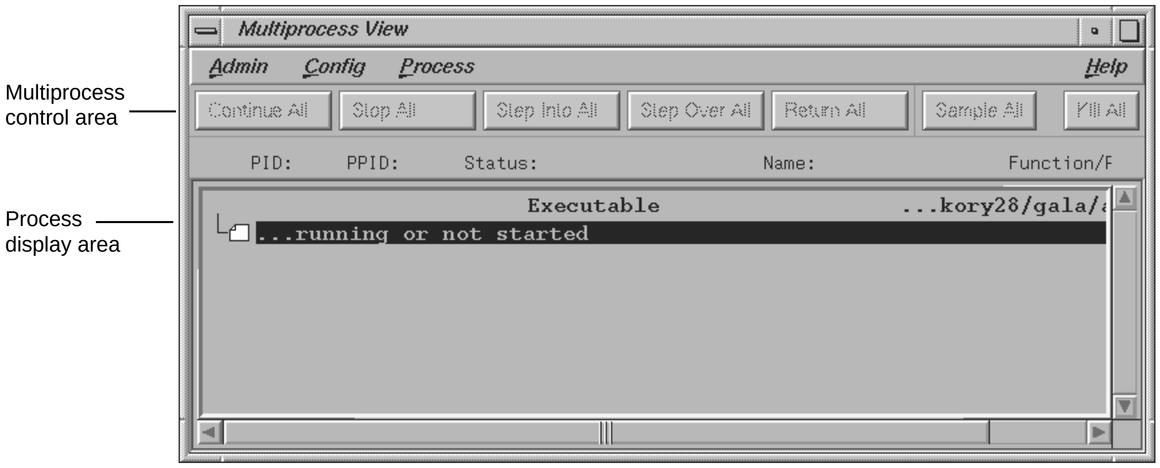
Figure 66. `Execution View`

A.2.2 Multiprocess View

WorkShop supports debugging of multiprocess applications, including processes spawned with either `fork` or `sproc` commands.

Multiprocess debugging is supported primarily through the `Multiprocess View` window. To display this window, select `Multiprocess View` from the `Admin` menu. `Multiprocess View` displays a hierarchical view of your pthreaded application. Pthreaded processes are marked with a folder icon. Clicking the folder changes the view to show that process's pthreads. Clicking on a thread opens the call stack for the thread.

Figure 67, page 141, shows a typical `Multiprocess View`.



```
PID:    PPID:    Status:                    Name:             Function/F
                        Executable                  ...kory28/gala/
...running or not started
```

Figure 67. `Multiprocess View`

Note: Main View is attached to the parent process.

A.2.3 Viewing Status

When `Multiprocess View` comes up, it lists the status of all processes in the process group. This view includes the following information:

`PID:`	Shows the process identifier (PID).
`PPID:`	Lists the parent process PIDs. Notice in Figure 67, page 141, that the first process `PID#7748` is the parent process of the second.
`Status/State:`	Shows `Status` or `State` depending on how you have set your preferences on the `Preferences` menu. `Status` is user–level status and `State` is the kernel–level status.
`Name:`	Identifies the process by file name.

`Function/PC:`	Indicates the current function and program counter (PC) for any stopped processes.

The following `Status` and `State` conditions are possible:

Status	State
`Running`	`RUNNING`
`Stopped`	`RUNNING`
`Stopped on breakpoint`	`RUNNING` (but at a trap pc)
`Waiting to terminate`	`JOIN`
`Thread terminated`	`DEAD`
`Waiting on kernel`	`READY`
`Waiting on mutex`	`MUTEX-WAIT`

A.2.4 Multiprocess Control Buttons

`Multiprocess View` uses the same control buttons as Main View with two exceptions. The buttons are applied to all processes as a group. There is no separate `Run` button. Using a control button in `Multiprocess View` has the same effect as clicking the button in each process's Main View window. The following buttons are available:

`Continue [all]`	Resumes program execution after a halt and continues until a stop trap or other event stops execution.
`Stop [all]`	Stops execution of all processes. When program execution stops, the current source line of each process is highlighted in its Main View, if one is active, and annotated with an arrow indicating the PC.
`Step Into [all]`	Steps to the next source line and into function calls. To step a specific number of lines, hold down the right mouse button over the `Step Into` button. A pop-up menu displays that lets you select one of the fixed values or a specified number of steps.

`Step Over [all]`	Steps to the next source line and over function calls. To step a specific number of lines, hold down the right button over the `Step Over` button. A pop-up menu displays that lets you select one of the fixed values or a specified number of steps.
`Return [all]`	Executes the remaining instructions in the current function. Program execution stops upon return from that procedure.
`Sample [all]`	Collects performance data for each process (if performance data collection is enabled).
`Kill [all]`	Terminates all processes in the group.

A.2.5 Controlling Processes

The `Process` menu lets you control processes and threads.

The `Process` menu has the following options:

`Change focus to this entry...`	Opens a dialog that allows you to switch the process or thread currently focused on in the `cvmain` window to the process or thread selected in the `Multiprocess View` window. Selecting a call stack entry changes `cvmain`'s focus to that process or thread and positions the `cvmain` window at the offset of the selected call stack.
`Create a new window...`	Opens a dialog that allows you to bring up a new `cvmain` window on the selected process or thread.
`Goto...`	Opens a dialog box that allows you to enter the name of a thread on which focus should be switched. This is useful when multiple threads, all at the same location, are collapsed into a single line. While `Change focus to this entry...` always takes you to the first thread, `Goto...` lets you jump to any thread.
`Details...`	Shows thread–specific details.
`Add...`	Opens a dialog in which you can select from a list of process ids. Selected processes ids are added to the `Multiprocess View` group.

`Remove`	Removes the highlighted process.

A process in a `sproc` share group cannot be removed from the process group.

A.2.6 Controlling Preferences

The `Preferences...` option in the `Config` menu brings up the `Multiprocess View Preferences` dialog . It lets you control when processes are added to the group and specifies their behavior.

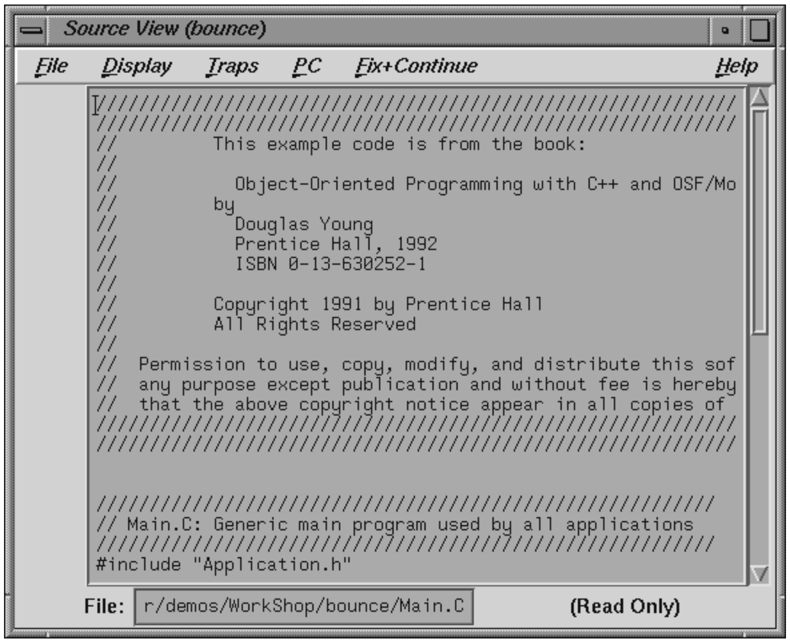
The `Multiprocess View` preference options are:

`Stack Depth`	Allows you to set how many lines of the call stack should be displayed when opening the call stack. Default is 10 lines.
`Levels to open`	Allows you to specify the number of levels of hierarchy to be displayed. By default, only the top level of the hierarchy is displayed.
`Attach to forked processes`	Automatically attaches new processes spawned by the `fork` command to the group. (Note that processes spawned by `sproc` are always attached.)
`Copy traps to forked processes`	Copies traps you have set in the parent process to new `forked` processes automatically. If you create parent traps with `Trap Manager` and specify `pgrp`, then the children inherit these traps automatically, regardless of the state of this flag.
`Copy traps to sproc'd processes`	Copies traps you have set in the parent process to new `sproc'd` processes automatically. As in the previous option, if you create parent traps with the `Trap Manager` and specify `pgrp`, the children inherit these traps automatically, whether this flag is set or not.
`Resume parent after fork`	Restarts the parent process automatically when a child is `forked`.
`Resume child after attach on fork`	Restarts the new `forked` process automatically when it is attached. If this option is left off, a new process will stop as soon as it is attached.
`Resume parent after sproc`	Restarts the parent process automatically when a child is `sproc'd`.

`Resume child after attach on sproc`	Restarts the new `sproc'd` process automatically when it is attached. If this option is left off, a new process will stop as soon as it is attached.
`Combine threads at same location`	Displays threads stopped at the same location at the same time. (It is possible for threads to arrive at the same location through different logical routes.)
`Show Thread Status vs Thread State`	Displays thread status, which is the user–level status as opposed to showing thread state, which is the kernel–level status.
`Follow highest priority thread`	Specifies that window focus shift after each event to the thread that is considered to be the most interesting thread. Events that may make a thread interesting include the following: an error condition, being stopped at a breakpoint, being stopped waiting on a mutex, and so forth.

A.2.7 Source View

The `Source View` (see Figure 68, page 146) displays your source, opening your `Main` program file by default.



```
///////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////
//         This example code is from the book:
//
//           Object-Oriented Programming with C++ and OSF/Mo
//         by
//           Douglas Young
//           Prentice Hall, 1992
//           ISBN 0-13-630252-1
//
//         Copyright 1991 by Prentice Hall
//         All Rights Reserved
//
//  Permission to use, copy, modify, and distribute this sof
//  any purpose except publication and without fee is hereby
//  that the above copyright notice appear in all copies of
///////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////


/////////////////////////////////////////////////////////////
// Main.C: Generic main program used by all applications
/////////////////////////////////////////////////////////////
#include "Application.h"
```

```
r/demos/WorkShop/bounce/Main.C
```

Figure 68. `Source View`

The `Source View` menu bar contains selections duplicated from the Main View: `Display`, `Traps`, `PC`, and `Fix+Continue`. Each of these menus has the same functionality as its counterpart in the Main View (see Section A.9.4.1, page 241). The only new menu selection is the `File` menu described below:

`Open...`	Launches the file dialog that allows you to choose a file to load into `Source View`.
`Save`	Records changes made to the file during the current debugging session. You must first select `Make Editable` from the `Source` menu when the file is read only.
`Save As...`	Records changes made during the debugging session to the source file under a different filename.
`Save As Text...`	Records information in the display area as a text file.

`Open Separate...`	Launches the `File` dialog that allows you to create a new `Source View` with the contents of a different source file.
`Insert File...`	Inserts the text of a file within your current file.
`Clone`	Clones the current window.
`Fork Editor`	Starts your default editor on the current file. The default editor is determined by the `editorCommand` resource in the `app-defaults` file. The value of this resource defaults to `wsh -c vi +%d`, which means run `vi` in a `wsh` window and scroll to the current line. If the editor lets you specify a starting line, enter **`%d`** in the resource to indicate the new line number.
`Recompile`	Displays the `Build View` window that lets you compile the source code associated with the current executable.
`Make Editable`	Toggles the source code displayed between read-only and writeable states so that you can edit your code.
`Search`	Searches for a literal case-sensitive, literal case-insensitive, or regular expression. After you have set your target and clicked `Apply` (or pressed `Enter`), each instance is marked by a search target indicator in the scroll bar. You can search forward or backward in the file by clicking the `Next` or the `Prev` button. You can also click an indicator with the middle mouse button to scroll Main View to that point. Clicking `Reset` removes the search target indicators.
`Go To Line...`	Launches the `Go To Line` dialog that allows you to go to a specific line in the source. You can type in the line, or select the line number via the slider bar.
`Versioning`	Provides access to the configuration management tool, if you have designated one. The `cvconfig` script lets you designate ClearCase, RCS or SCCS. Type the following: **`cvconfig [clearcase	rcs	sccs]`**

	You must have `root` permissions to run `cvconfig`. Selecting any option displays a shell in which you can access the configuration management tool. The selections in the submenu are:
`Versioning` submenu	Contains the following options: • `CheckIn` — Saves the source file and checks it into the database as a new version. • `CheckOut` — Recalls the source file from the tool's database if you have the proper authority, locks it, and makes it editable. • `UncheckOut` — Cancels the checkout, with no changes registered.
`Close`	Dismisses the `Source View` window.

A.2.8 Process Meter

The `Process Meter` monitors resource usage of a running process without saving the data. Figure 69, page 148, shows the Process Meter in its default configuration (with only the `User Time` and `Sys Time` fields active).

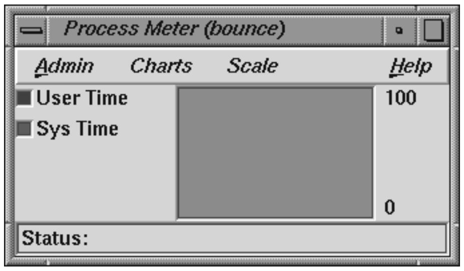


Figure 69. `Process Meter`

The `Process Meter` contains its own menu bar that contains the `Admin`, `Charts`, `Scale`, and `Help` menus. The `Admin` menu is the same as that described in Section A.3.1.1, page 150. The `Help` menu is the same as that described in Section A.1.10, page 139. The other menus are described in the following sections.

A.2.8.1 `Charts` Menu

The `Charts` menu contains a set of toggles that allow you to choose which charts are displayed in the `Process Meter` window. You can display as many charts simultaneously as you wish. The following choices are available:

- `User/Sys Time` (the default)
- `Major/Minor Faults`
- `Context Switches`
- `Bytes Read/Written`
- `Read/Write Sys Calls`
- `Other Sys Calls`
- `Total Sys Calls`
- `Signals`
- `Process Size`

A.2.8.2 `Scale` Menu

The `Scale` menu allows you to set the time scale for the processes displayed in the `Process Meter` window. You can choose a time scale from 2 seconds to 10 minutes.

A.3 Ada-specific Windows

This section discusses the Debugger `Task View` and `Exception View` windows that are specific to Ada code.

A.3.1 Task View

The `Task View` is an Ada-specific view that provides you with task and call stack information. If you do not have Ada installed on your system, the `Task View` menu option of the `Views` menu is grayed out.

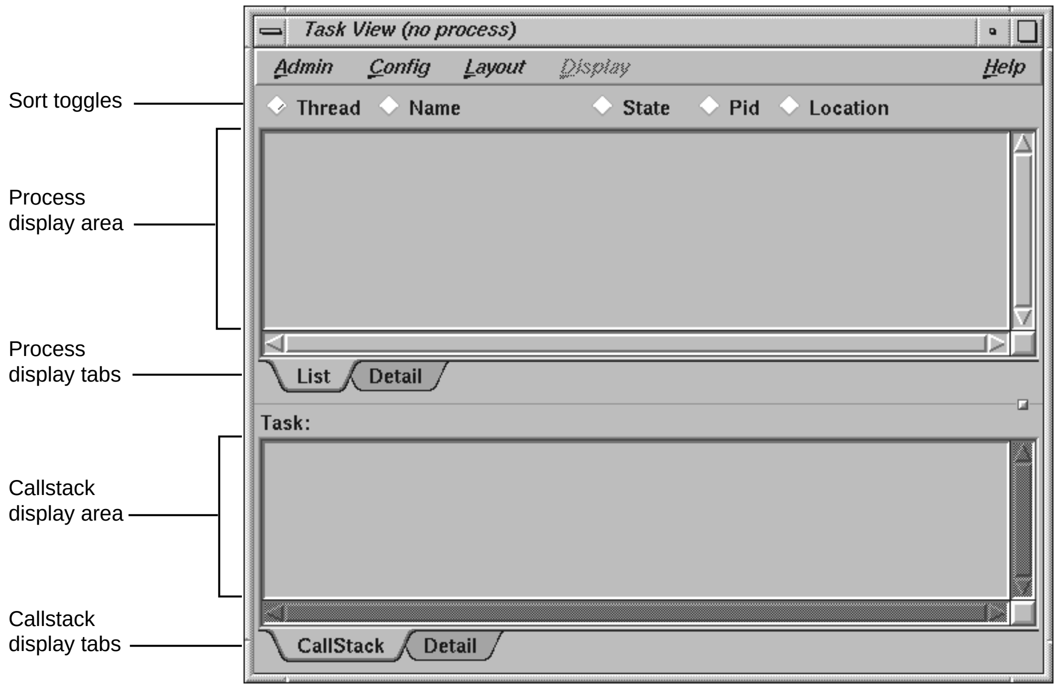


Figure 70. Task View

The Task View menu bar contains the Admin, Config, Layout, Display, and Help menus. The Help menu is the same as that described in Section A.1.10, page 139. Other menus are described in the following sections.

A.3.1.1 Admin Menu

The Admin menu contains the following options:

Active	This toggle activates the current window in a set of cloned windows.
Clone	Creates a clone of the current window. This function is not supported in the current release, and the option is grayed out.
Save As Text...	Launches the Save Text dialog. This dialog allows you to save your current session as text in a file you designate.

`Close`	Closes the current window.

A.3.1.2 `Config` Menu

The `Config` menu contains the following item:

`Preferences...`	Launches the `Task View Preference` dialog that allows you to set maximum call stack depth shown in `Task View`. Default depth is 32 frames.

A.3.1.3 `Layout` Menu

The `Layout` menu contains the following toggles:

`Task List`	Causes only the `CallStack View` to be shown.
`Single Task`	Causes only the `Process Display` to be shown.

A.3.1.4 `Display` Menu

The `Display` menu is divided into the `Task List Format` and `Callstack Format` sections. The `Callstack Format` toggles match the toggles that are contained in the `Callstack View Display` menu. The `Task List Format` toggle buttons are made available in the toggle sort list, as well as what information is displayed in the `Process Display` area.

The `Task View Display` menu contains the following toggles:

`Status`	Displays process status. This toggle is active by default.
`Priority`	Displays process priority.
`Sproc`	Displays the sproc value of the process. This toggle is active by default.
`Resource Vector`	Displays process resource vector value.
`Arg Values`	Allows you to set the argument values in `Task View`. This toggle is active by default.
`Arg Names`	Allows you to set the argument names in `Task View`. This toggle is active by default.
`Arg Types`	Allows you to set the argument types in `Task View`.
`Location`	Allows you to set the function location in `Task View`. This toggle is active by default.

`PC`	Allows you to set the program counter (PC) in `Task View`.

In addition to menus, `Task View` also contains the following items from which you can select to vary the display:

Sort toggles	Allow you to sort the process list by `Thread`, `Name`, `State`, `Pid`, or `Location`, depending on which of the buttons is active. Default selection is `Thread`.
Process display tabs	Allow you to view a list of tasks or details of the currently running (highlighted) task.
Callstack display tabs	Allow you to view all call stack information or call stack details of the currently selected process.

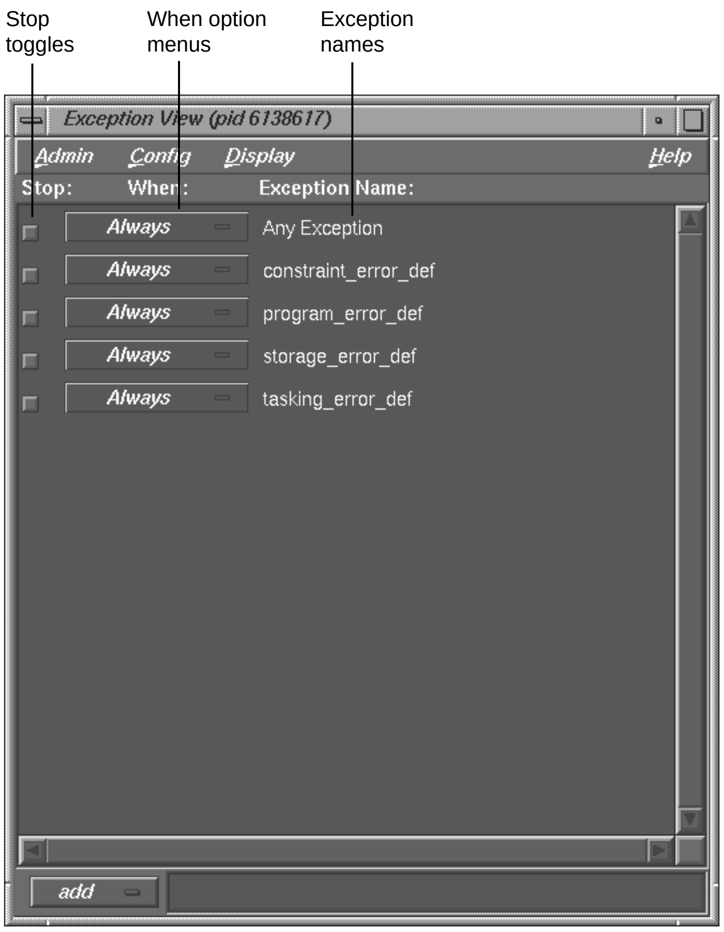
A.3.2 Exception View

`Exception View` is an Ada-specific view that allows you to set traps on exceptions and control exception handling. This view functions only if Ada is installed. By default, this view displays only the following predefined Ada exceptions:

- Constraint errors
- Program errors
- Storage errors
- Tasking errors

In addition, a single breakpoint is set on any unhandled exception.

Figure 71 shows a typical `Exception View` window.



Any Exception
constraint_error_def
program_error_def
storage_error_def
tasking_error_def

Figure 71. Exception View

The Admin menu has the following options:

Active	Activates the current window in a set of cloned windows.
Clone	Creates a clone of the current window.
Save As Text...	Launches the Save Text dialog. This dialog allows you to save your current session as text in a file you designate.

`Close`	Closes the current window.

The `Config` menu has the following options:

`Load Exceptions...`	Opens the `Load User Defined Exceptions` dialog that allows you to add additional exceptions to the predefined Ada exceptions.
`Save Exceptions...`	Opens the `Save User Defined Exceptions` dialog that allows you to save any user-defined exceptions to the predefined Ada exceptions.

The `Display` menu has the following options:

`Delete All`	Deletes all exception traps.
`Clear All Traps`	Clears all exception traps. Clearing traps is not the same as deleting traps. Clearing only temporarily affects traps while deleting removes them permanently.
`Reset All Buttons`	Resets all button functions.

The `Stop:` boxes toggle on and off to indicate whether a trap is active.

The `When:` control menus allow you to select when an exception trap fires. The following choices are available:

`Always`	Stops any time the exception is raised.
`WhenOthers`	Stops when caught by a `when others` handler rather than an explicit handler or when unhandled.
`Unhandled`	Stops when the exception is unhandled.

In the unlabled text field at the bottom right of the window you can enter a single, fully qualified Ada exception name or a single, fully qualified Ada unit name. Depending on whether the `add`, `remove`, or `find` mode is active; pressing `Enter` will cause one of the following actions to occur:

- `add` mode:
  - Single exception: Adds single exception to the exception list
  - Library unit name: Adds all exceptions found in that library unit name to the exception list
- `remove` mode:

  - Single exception: Removes single exception from the exception list
  - Library unit name: removes all exceptions found in that library unit name from the exception list
- `find` mode
  - Single exception: positions top of the exception list to single exception
  - Library unit name: positions top of the exception list to the first exception found in given library unit name

A.4 X/Motif Analyzer Windows

The X/Motif Analyzer provides specific debugging support for X/Motif applications. There are various examiners for different X/Motif objects, such as widgets and X Window System graphics context, that might be difficult or impossible to inspect using ordinary debugger functionality.

To access the `X/Motif Analyzer` window, pull down the `Views` menu and select `X/Motif Analyzer` (see Figure 72, page 156).

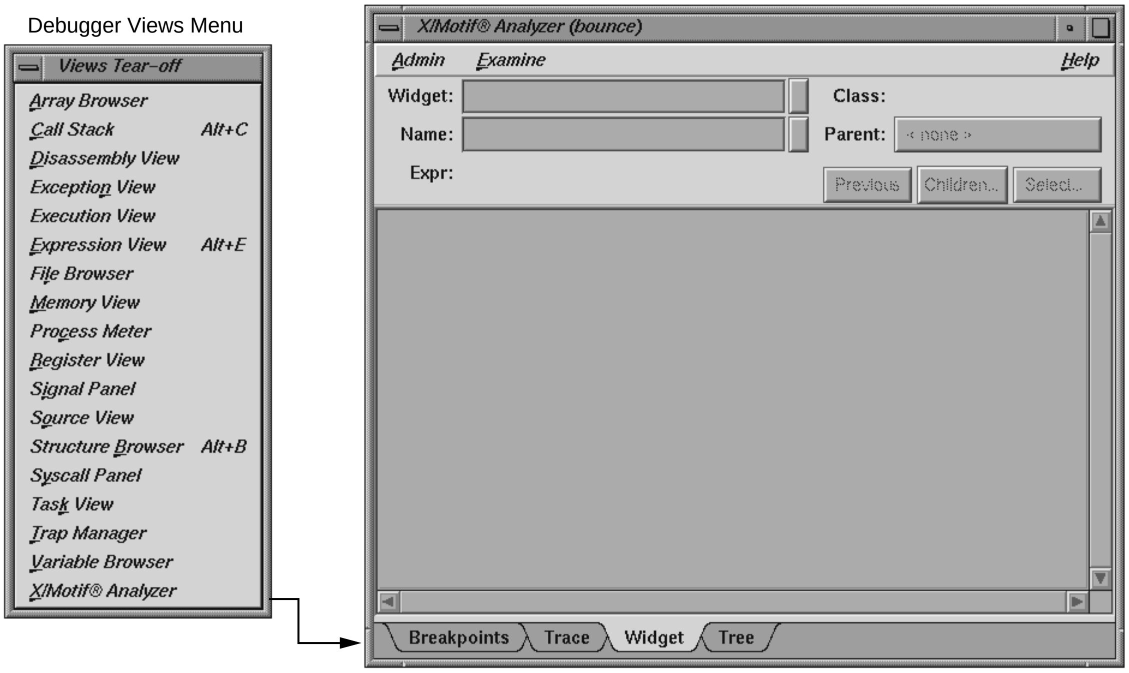


Figure 72. Launching the `X/Motif Analyzer` Window

A.4.1 Global Objects

Though the X/Motif Analyzer is made up of several different examiner windows, a number of objects remain constant throughout window changes. The following examiners are available:

- Breakpoints Examiner, Section A.4.2, page 158
- Trace Examiner, Section A.4.3, page 175
- Widget Examiner, Section A.4.4, page 177
- Tree Examiner, Section A.4.5, page 178
- Callback Examiner, Section A.4.6, page 179
- Window Examiner, Section A.4.7, page 180
- Event Examiner, Section A.4.8, page 181
- Graphics Context Examiner, Section A.4.9, page 182

- Pixmap Examiner, Section A.4.10, page 183
- Widget Class Examiner, Section A.4.11, page 184

A.4.1.1 `Admin` Menu

The `Admin` menu offers the following menu selections:

`Active`	Activates the current window in a set of cloned windows. In the current release, this toggle is always active.
`Clone`	Creates a clone of the current window. This function is not supported in the current release and the option is grayed out.
`Save As Text...`	Launches the `Save Text` dialog. This dialog allows you to save your current session as text in a file you designate.
`Close`	Closes the current window.

A.4.1.2 `Examine` Menu

The `Examine` menu offers the following options:

`Selection`	Selects the currently highlighted object for examination.
`Widget`	Uses the current selection as input to the widget examiner, then opens that examiner (see Section A.4.4, page 177, for information).
`Widget Tree`	Switches the window view to the widget tree examiner (see Section A.4.5, page 178, for information).
`Widget Class`	Switches the window view to the widget class examiner (see Section A.4.11, page 184, for information).
`Window`	Switches the window view to the window examiner (see Section A.4.7, page 180, for information).
`X Event`	Switches the window view to the X Event examiner (see Section A.4.8, page 181, for information).

`X Graphics Context`	Switches the window view to the X graphics context examiner (see Section A.4.9, page 182, for information).
`X Pixmap`	Switches the window view to the X pixmap examiner (see Section A.4.10, page 183, for information).

A.4.1.3 Examiner Tabs

In addition to access through the `Examine` menu, each examiner can be accessed through a tab at the bottom of each view (see Figure 73, page 158).

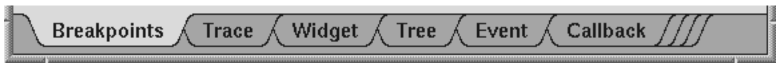


Figure 73. Examiner Tabs

When first launched, the X/Motif Analyzer has the following four tabs: `Breakpoints`, `Trace`, `Widget`, and `Tree`. As you select other examiners through the `Examine` menu, new tabs are added for the new examiners. Any of these new tabs may be deleted at any time by selecting the tab, clicking the right mouse button, and then selecting `Remove Examiner`. The initial four tabs may not be removed.

A.4.1.4 `Return` Button

Both the `Widget:` and `Name:` text fields have return buttons (see Figure 74, page 159) just to the right. Clicking these buttons causes the X/Motif Analyzer to respond exactly as if you had pressed `Return` on your keyboard.

A.4.2 Breakpoints Examiner

The Breakpoints examiner is not really an examiner, but a control area where you can set widget-level breakpoints. The breakpoints examiner is divided into three areas (see Figure 74, page 159):

- The widget specification area that contains the same information as that in the Widget examiner. You can select a widget address, name, or class in this area, as well as move to the widgets parents or children, or select a widget in the application. In cases where the breakpoint type does not apply to widgets (for example, input-handler breakpoints), this area is blank.

- The parameter specification area, the contents of which vary according to the type of breakpoint you are setting. For example, for callback breakpoints, this area contains the callback name and client data; for event-handler breakpoints, it contains the event type and the client data, and so on.
- The breakpoint area, which contains the breakpoint name, a search field, and the `Add`, `Modify`, `Delete`, and `Step To` buttons.

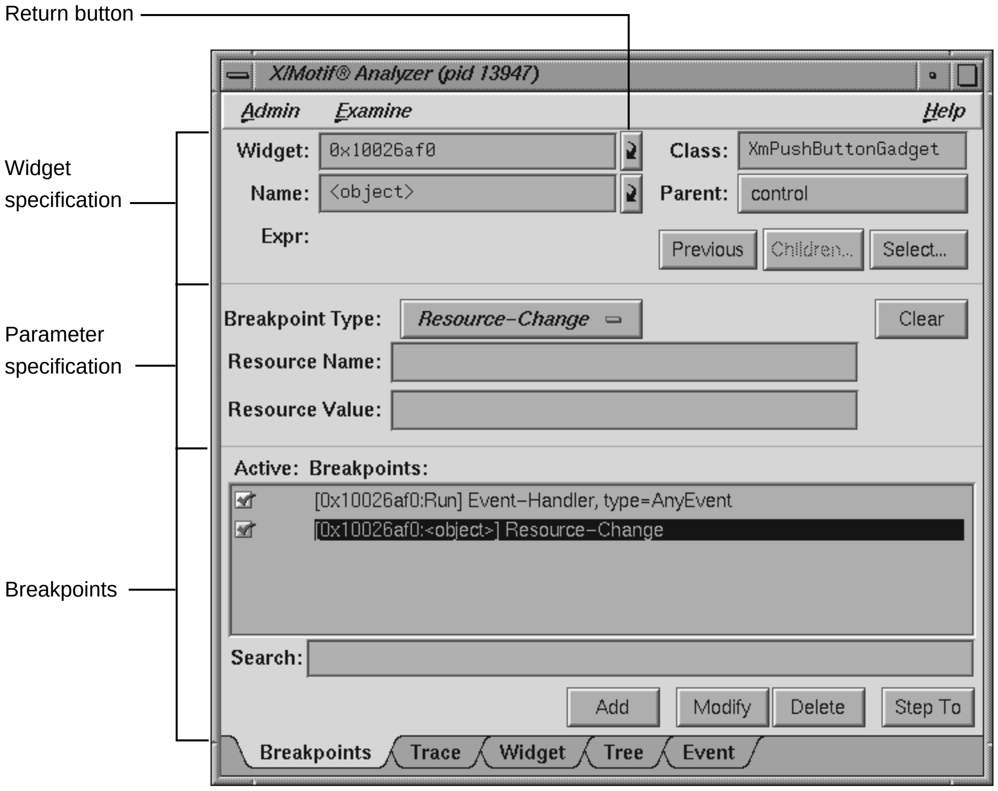


Figure 74. Breakpoints Examiner Display in the `X/Motif Analyzer` Window

The control area has eight different breakpoint types that it can examine. These types are set through the `Breakpoint Type:` options. The following `Breakpoint Type:` options are available:

`Callback`	Widget callback installed by `XtAddCallback`. Parameters include callback name and `client_data` `XtPointer` value. See Section A.4.2.1, page 160, for more information.
`Event-Handler`	Widget event handler installed by `XtAddEventHandler`. Parameters include X event type and `client_data` `XtPointer` value. See Section A.4.2.1.1, page 162, for more information.
`Resource-Change`	Resource change caused by `XtSetValues` or `XtVaSetValues`. Parameters include resource name and resource value, both strings. See Section A.4.2.1.2, page 164, for more information.
`Timeout-Procedure`	Timeout callback installed by `XtAppAddTimeOut`. Parameters include `client_data` `XtPointer` value. See Section A.4.2.1.3, page 165, for more information.
`Input-Handler`	Input callback installed by `XtAppAddInput`. Parameters include `client_data` `XtPointer` value. See Section A.4.2.1.4, page 167, for more information.
`State-Change`	Various widget state changes (for example, managed or realized). Parameters include widget state. See Section A.4.2.1.5, page 168, for more information.
`X-Event`	X event received by target application. Parameters include X event type. See Section A.4.2.1.6, page 170, for more information.
`X-Request`	X request received by target application. Parameters include X request type. See Section A.4.2.1.7, page 172, for more information.

A.4.2.1 Callback Breakpoints Examiner

When the `Callback` option of the `Breakpoint Type` option button in the `Breakpoints Examiner` is selected, the Callback Breakpoints Examiner is displayed.

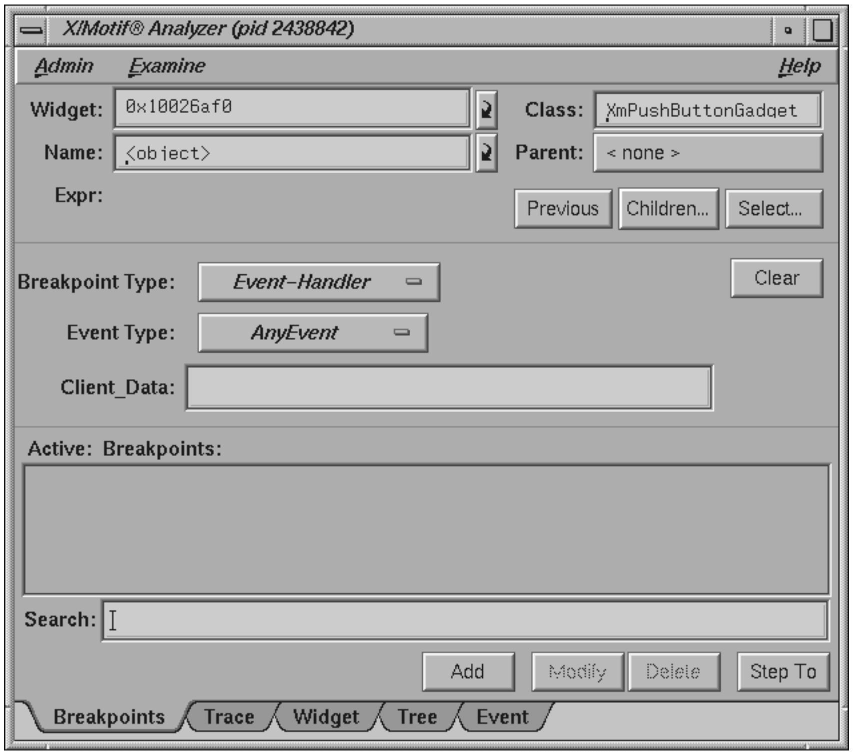
The Callback Breakpoints examiner contains the following items:

`Widget:` text field	Allows you to choose a widget to examine by entering the widget address.
`Name:` text field	Allows you to choose a widget to examine by entering the widget name.
`Class:` text field	Allows you to choose a widget to examine by entering the widget's class. Leave the field blank or enter **Any** to select all widgets.
`Parent:` text field	Allows you to move the parent of the currently selected widget.
`Previous` button	Moves you to the previously selected widget.
`Children...` button	Shows you the widget's children (it is grayed out if the selected widget cannot have children).
`Select...` button	Allows you to select the widget in the target process.
`Breakpoint Type:` option button	Allows you to select the type of breakpoint you wish to set.
`Clear` button	Clears all the current breakpoint selections and text fields.
`Callback Name` text field	Allows you to set the name of the callback for the breakpoint.
`Client_Data` text field	Allows you to pass and get back pointer values for `Client_Data:`.
`Search:` text field	Allows you to perform a text search through your breakpoints.
`Add` button	Allows you to add a new breakpoint.
`Modify` button	Allows you to change the selected breakpoint's settings.
`Delete` button	Deletes the selected breakpoint.
`Step To` button	Allows you to step to the next condition. `Step To` creates a temporary breakpoint, resumes the process, and waits until the process stops. This temporary breakpoint acts exactly like an ordinary breakpoint, save that the `Step To` button

automatically resumes the process and puts up a busy cursor until the condition becomes true.

A.4.2.1.1 Event-Handler Breakpoints Examiner

When the `Event-Handler` option of the `Breakpoint Type:` option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure 75, page 162.



Widget: 0x10026af0
Class: XmPushButtonGadget
Name: <object>
Parent: < none >
Breakpoint Type: Event-Handler
Event Type: AnyEvent

Figure 75. Event-Handler Breakpoints Examiner

The Event-Handler Breakpoints examiner contains the following items:

`Widget:` text field	Allows you to choose a widget to examine by entering the widget address.
`Name:` text field	Allows you to choose a widget to examine by entering the widget name.

`Class:` text field	Allows you to choose a widget to examine by entering the widget's class. Leave the field blank or enter **Any** to select all widgets.
`Parent:` text field	Allows you to move the parent of the currently selected widget.
`Previous` button	Moves you to the previously selected widget.
`Children...` button	Shows you the widget's children (it is grayed out if the selected widget cannot have children).
`Select...` button	Allows you to select the widget in the target process.
`Breakpoint Type:` option button	Allows you to select the type of breakpoint you wish to set.
`Clear` button	Clears all the current breakpoint selections and text fields.
`Event Type`: option button	Takes the place of the `Callback Name:` text field in the Callback Breakpoints examiner. Allows you to set the event type for a given breakpoint.
`Client_Data:` text field	Allows you to pass and get back pointer values for the `Client_Data:`.
`Search:` text field	Allows you to perform a text search through your breakpoints.
`Add` button	Allows you to add a new breakpoint.
`Modify` button	Allows you to change the selected breakpoint's settings.
`Delete` button	Deletes the selected breakpoint.
`Step To` button	Allows you to step to the next condition. `Step To` creates a temporary breakpoint, resumes the process, and waits until the process stops. This temporary breakpoint acts exactly like an ordinary breakpoint, save that the `Step To` button

automatically resumes the process and puts up a busy cursor until the condition becomes true.

A.4.2.1.2 Resource-Change Breakpoints Examiner

When the `Resource-Change` option of the `Breakpoint Type:` option button in the Breakpoint Examiner is selected, the examiner appears.

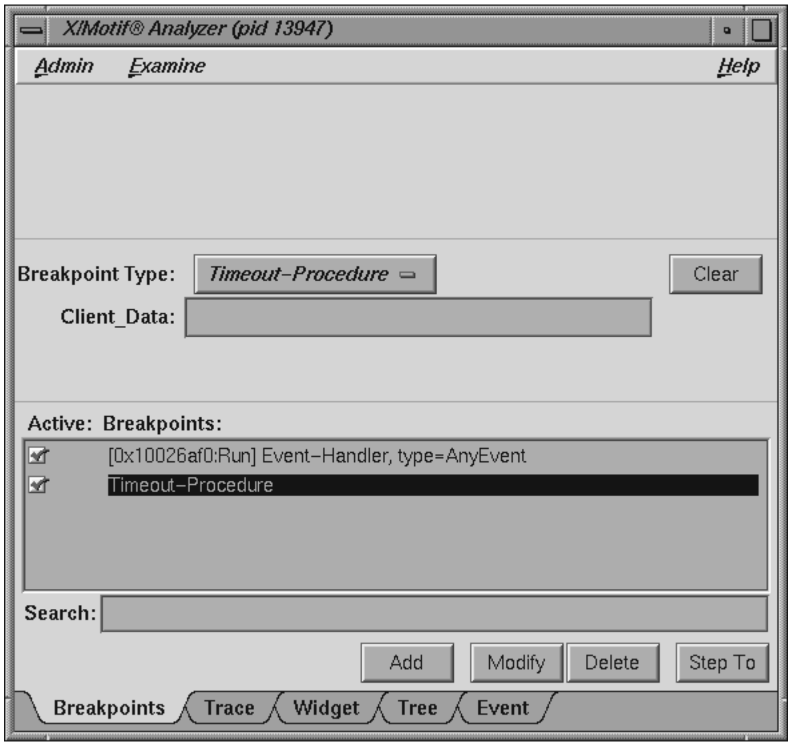
The Resource-Change Breakpoints examiner contains the following items:

`Widget:` text field	Allows you to choose a widget to examine by entering the widget address.
`Name:` text field	Allows you to choose a widget to examine by entering the widget name.
`Class:` text field	Allows you to choose a widget to examine by entering the widget's class. Leave the field blank or enter **Any** to select all widgets.
`Parent:` text field	Allows you to move the parent of the currently selected widget.
`Previous` button	Moves you to the previously selected widget.
`Children...` button	Shows you the widget's children (it is grayed out if the selected widget cannot have children).
`Select...` button	Allows you to select the widget in the target process.
`Breakpoint Type:` option button	Allows you to select the type of breakpoint you wish to set.
`Clear` button	Clears all the current breakpoint selections and text fields.
`Resource Name:` text field	Takes the place of the `Callback Name:` text field. Allows you to set the resource name for the breakpoint.
`Resource Value:` text field	Takes the place of the `Client Data:` text field. Allows you to set the resource value for the breakpoint.
`Search:` text field	Allows you to perform a text search through your breakpoints.
`Add` button	Allows you to add a new breakpoint.

`Modify` button	Allows you to change the selected breakpoint’s settings.
`Delete button`	Deletes the selected breakpoint.
`Step To` button	Allows you to step to the next condition. `Step To` creates a temporary breakpoint, resumes the process, and waits until the process stops. This temporary breakpoint acts exactly like an ordinary breakpoint, save that the `Step To` button automatically resumes the process and puts up a busy cursor until the condition becomes true.

A.4.2.1.3 Timeout-Procedure Breakpoints Examiner

When the `Timeout Procedure` option of the `Breakpoint Type` option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure 76, page 166.



[0x10026af0:Run] Event–Handler, type=AnyEvent
Timeout–Procedure

Figure 76. Timeout-Procedure Breakpoints Examiner

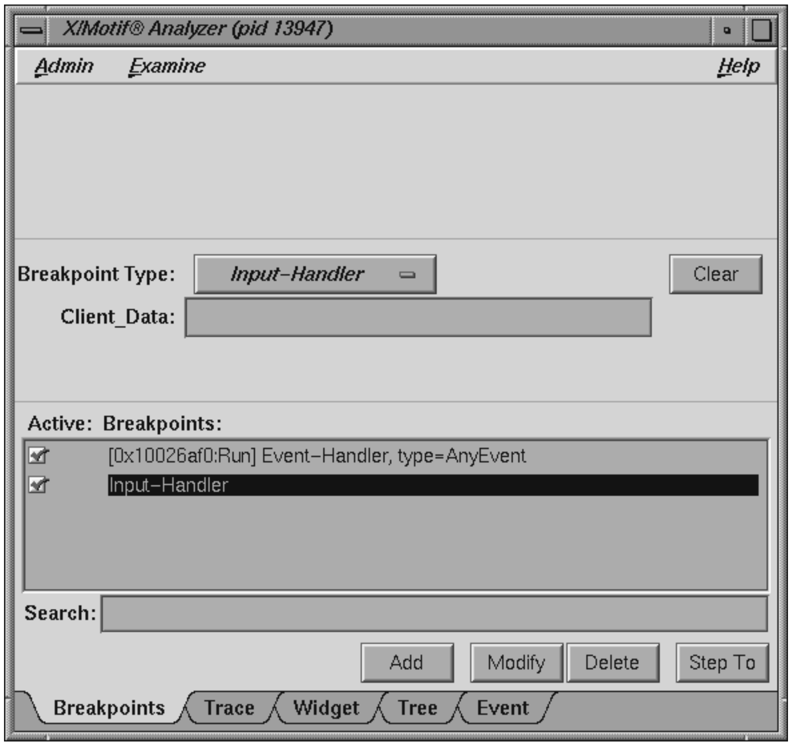
The Resource-Change Breakpoints examiner contains the following items:

`Breakpoint Type`: option button	Allows you to select the type of breakpoint you wish to set.
`Clear` button	Clears all the current breakpoint selections and text fields.
`Client_Data:` text field	Allows you to pass in and get back pointer values for the `Client_Data`.
`Search`: text field	Allows you to perform a text search through your breakpoints.
`Add` button	Allows you to add a new breakpoint.
`Modify` button	Allows you to change the selected breakpoint's settings.

`Delete` button	Deletes the selected breakpoint.
`Step To` button	Allows you to step to the next condition. `Step To` creates a temporary breakpoint, resumes the process, and waits until the process stops. This temporary breakpoint acts exactly like an ordinary breakpoint, save that the `Step To` button automatically resumes the process and puts up a busy cursor until the condition becomes true.

A.4.2.1.4 Input-Handler Breakpoints Examiner

When the `Input-Handler` option of the `Breakpoint Type:` option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure 77, page 167.



[0x10026af0:Run] Event–Handler, type=AnyEvent
Input–Handler

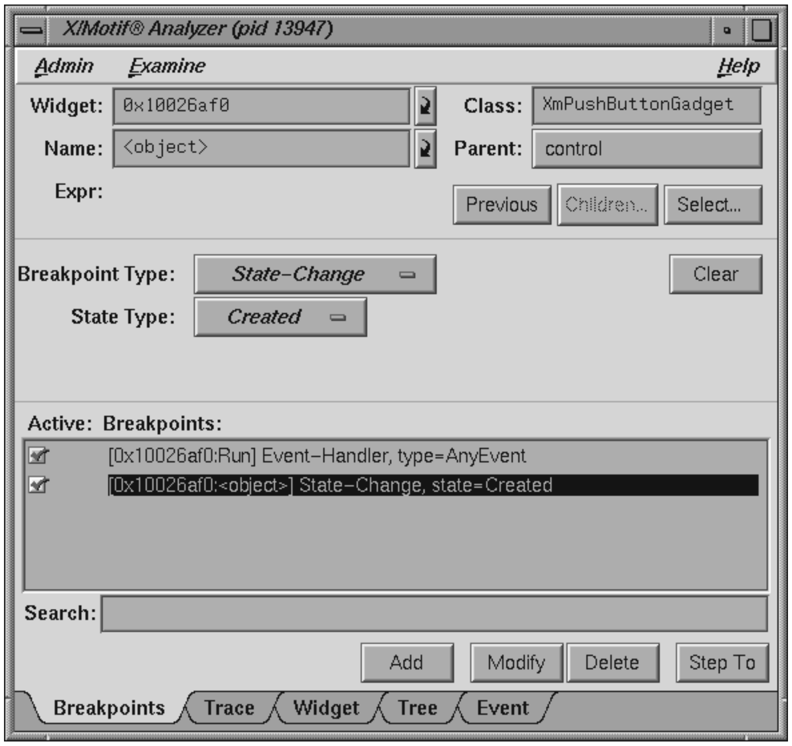
Figure 77. Input-Handler Breakpoints Examiner

The Input-Handler Breakpoints examiner contains the following items:

`Breakpoint Type:` option button	Allows you to select the type of breakpoint you wish to set.
`Clear` button	Clears all the current breakpoint selections and text fields.
`Client_Data:` text field	Allows you to pass in and get back pointer values for the `Client_Data`.
`Search:` text field	Allows you to perform a text search through your breakpoints.
`Add` button	Allows you to add a new breakpoint.
`Modify` button	Allows you to change the selected breakpoint's settings.
`Delete` button	Deletes the selected breakpoint.
`Step To` button	Allows you to step to the next condition. `Step To` creates a temporary breakpoint, resumes the process, and waits until the process stops. This temporary breakpoint acts exactly like an ordinary breakpoint, save that the `Step To` button automatically resumes the process and puts up a busy cursor until the condition becomes true.

A.4.2.1.5 State-Change Breakpoints Examiner

When the `State-Change` option of the `Breakpoint Type:` option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure 78, page 169.



Widget: 0x10026af0

Name: <object>

Class: XmPushButtonGadget

Parent: control

Expr:

Breakpoint Type: State-Change

State Type: Created

Active: Breakpoints:

✓ [0x10026af0:Run] Event–Handler, type=AnyEvent

✓ [0x10026af0:<object>] State–Change, state=Created

Search:

Figure 78. State-Change Breakpoints Examiner

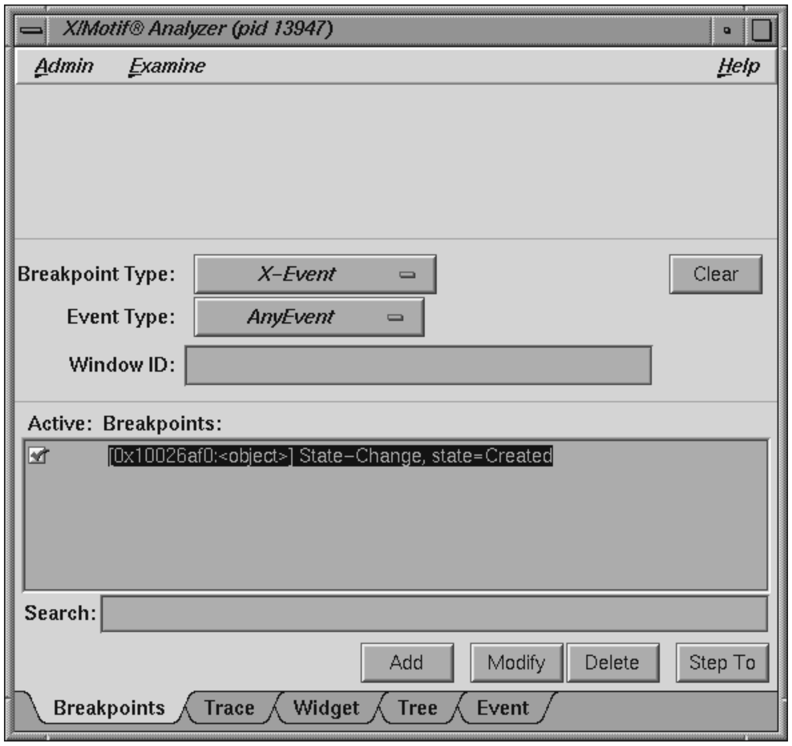
The Resource-Change Breakpoints examiner contains the following items:

`Widget:` text field	Allows you to choose a widget to examine by entering the widget address.
`Name:` text field	Allows you to choose a widget to examine by entering the widget name.
`Class:` text field	Allows you to choose a widget to examine by entering the widget's class. Leave the field blank or enter **Any** to select all widgets.
`Parent` button	Allows you to move the parent of the currently selected widget.
`Previous` button	Moves you to the previously selected widget.
`Children...` button	Shows you the widget's children (it is grayed out if the selected widget cannot have children).

`Select...` button	Allows you to select the widget in the target process.
`Breakpoint Type:` option button	Allows you to select the type of breakpoint you wish to set.
`Clear` button	Clears all the current breakpoint selections and text fields.
`State Type:` option button	Takes the place of the `Callback Name:` text field in the Callback Breakpoints examiner. Allows you to set the state change type for a given breakpoint.
`Search`: text field	Allows you to perform a text search through your breakpoints.
`Add` button	Allows you to add a new breakpoint.
`Modify` button	Allows you to change the selected breakpoint's settings.
`Delete` button	Deletes the selected breakpoint.
`Step To` button	Allows you to step to the next condition. `Step To` creates a temporary breakpoint, resumes the process, and waits until the process stops. This temporary breakpoint acts exactly like an ordinary breakpoint, save that the `Step To` button automatically resumes the process and puts up a busy cursor until the condition becomes true.

A.4.2.1.6 X-Event Breakpoints Examiner

When you select the `X-Event` option of the `Breakpoint Type:` option button in the Breakpoint Examiner, the examiner appears as shown in Figure 79, page 171.



```
[0x10026af0:<object>] State-Change, state=Created
```

Figure 79. X-Event Breakpoints Examiner

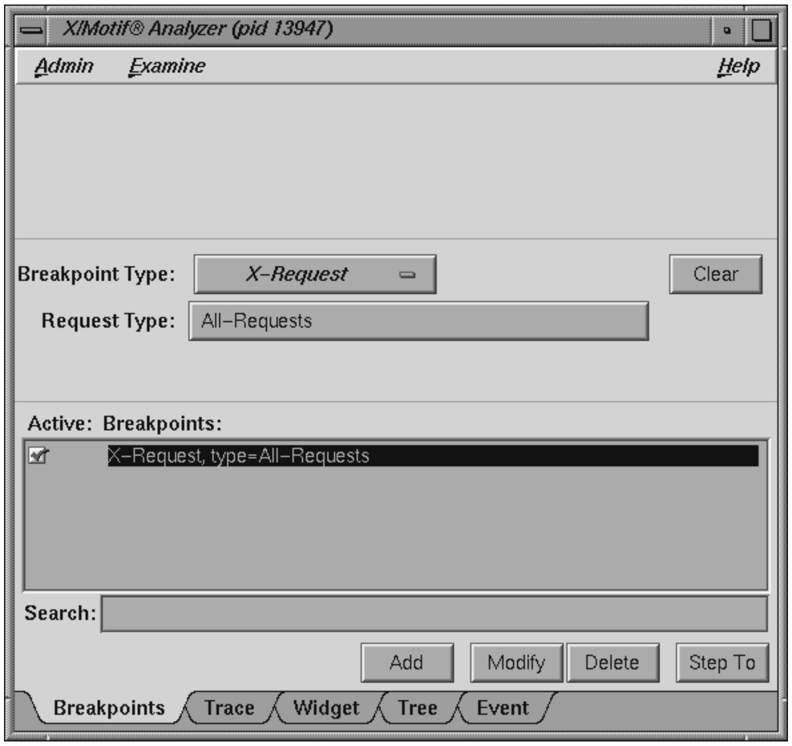
The X-Event Breakpoints examiner contains the following items:

`Breakpoint Type:` option button	Allows you to select the type of breakpoint you wish to set.
`Clear` button	Clears all the current breakpoint selections and text fields.
`Event Type:` option button	Takes the place of the `Callback Name` text field in the Callback Breakpoints examiner. Allows you to set the event type for a given breakpoint.
`Window ID:` text field	Takes the place of the `Client_Data:` text field. Allows you to set the Window ID value for the breakpoint.
`Search:` text field	Allows you to perform a text search through your breakpoints.

`Add` button	Allows you to add a new breakpoint.
`Modify` button	Allows you to change the selected breakpoint's settings.
`Delete` button	Deletes the selected breakpoint.
`Step To button`	Allows you to step to the next condition. `Step To` creates a temporary breakpoint, resumes the process, and waits until the process stops. This temporary breakpoint acts exactly like an ordinary breakpoint, save that the `Step To` button automatically resumes the process and puts up a busy cursor until the condition becomes true.

A.4.2.1.7 X-Request Breakpoints Examiner

When the `X-Request` option of the `Breakpoint Type:` option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure 80, page 173.



Breakpoint Type: X-Request
Clear
Request Type: All-Requests
Active: Breakpoints:
✓ X-Request, type=All-Requests
Search:

Figure 80. X-Request Breakpoints Examiner

The X-Request Breakpoints examiner contains the following items:

`Breakpoint Type:` option button	Allows you to select the type of breakpoint you wish to set.
`Clear` button	Clears all the current breakpoint selections and text fields.
`Request Type` button	Launches the `Request Type Selection` dialog (see Figure 81, page 174). This dialog allows you to select the type of X-Request used for your breakpoint. The information displayed is in outline form; selecting a given item selects all its subitems. For example, if you select `Window-Category`, `CreateWindow`, `ChangeWindowAttributes`,

GetWindowAttributes, and so on are also selected.

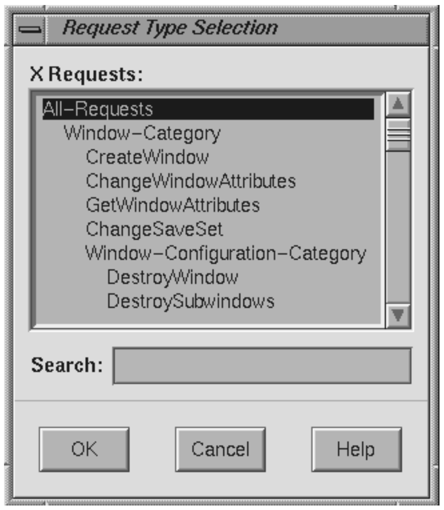


Figure 81. Request Type Selection Dialog

Search: text field	Allows you to perform a text search through your breakpoints.
Add button	Allows you to add a new breakpoint.
Modify button	Allows you to change the selected breakpoint's settings.
Delete button	Deletes the selected breakpoint.
Step To button	Allows you to step to the next condition. Step To creates a temporary breakpoint, resumes the process, and waits until the process stops. This temporary breakpoint acts exactly like an ordinary breakpoint, save that the Step To button

automatically resumes the process and puts up a busy cursor until the condition becomes true.

A.4.3 Trace Examiner

The Trace examiner (see Figure 82, page 176) is a control area where you can trace the execution of your application and collect various forms of data. The following data is collected:

- X Server Events
- X Server Requests
- Widget Event Dispatch Information
- Widget Resource Changes (through `XtSetValues`)
- Widget State Changes (`create`, `destroy`, `manage`, `realize`, and `unmanage`)
- `Xt` Callbacks (`widget`, `event handler`, `work proc`, `timeout`, `input`, and `signal`)

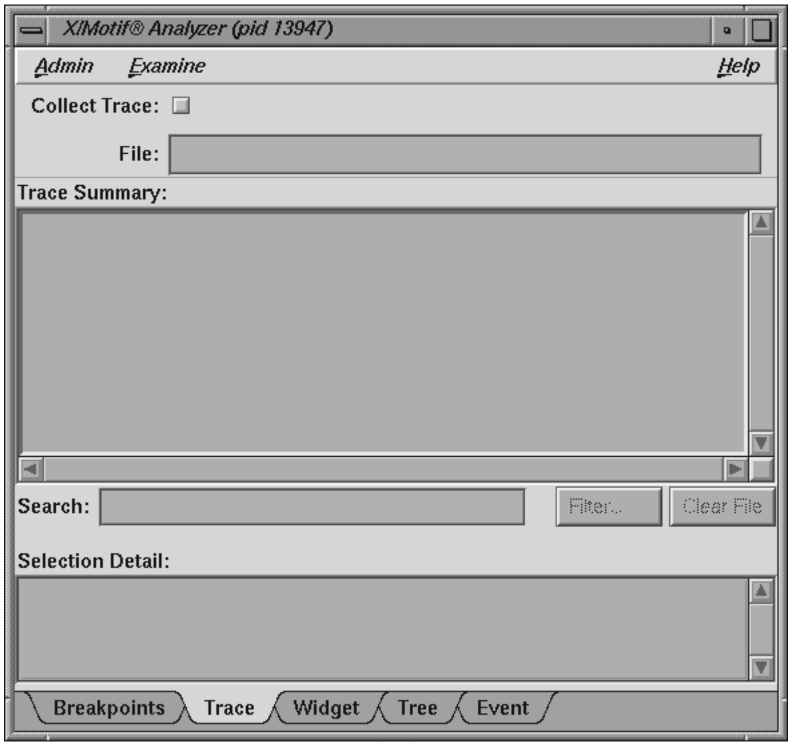


Figure 82. Trace Examiner

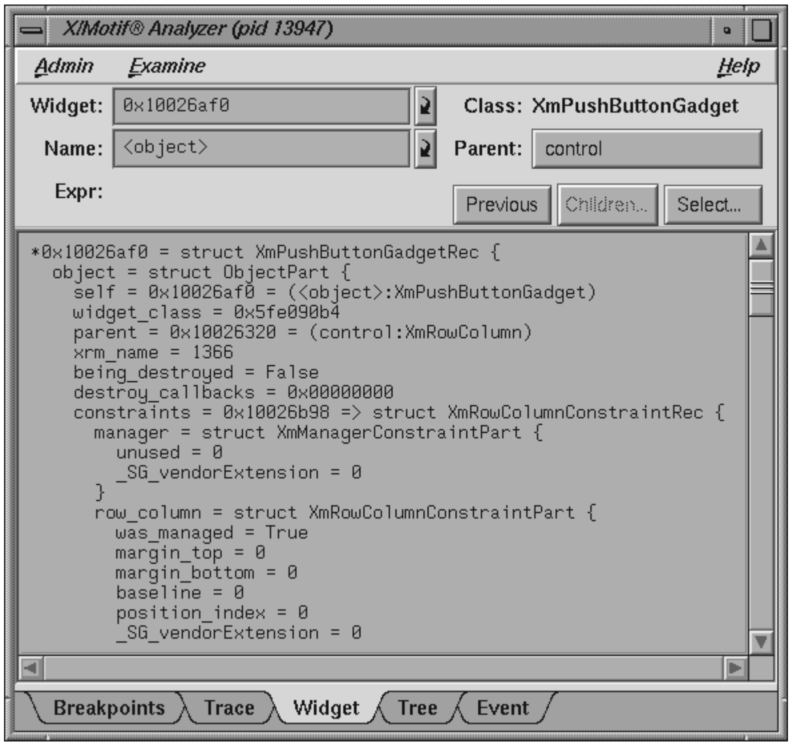
The Trace examiner contains the following items:

`Collect Trace:` toggle	Allows you to turn the tracing on and off.
`File:` text field	Allows you to select the filename for the trace. If no file is selected, a default filename for the trace is chosen.
`Search:` text field	Allows you to perform an incremental, textural search for the trace list.
`Filter...` button	Launches a dialog that allows you to select the trace entry types you want displayed in the list.

`Clear File` button	Erases the trace file. Any subsequent trace information goes to the beginning of the file.

A.4.4 Widget Examiner

The Widget examiner (see Figure 83, page 177) displays the internal `Xt` widget structure, as well as the `Xt` inheritance implementation using nested C constructs.



```
*0x10026af0 = struct XmPushButtonGadgetRec {
  object = struct ObjectPart {
    self = 0x10026af0 = (<object>:XmPushButtonGadget)
    widget_class = 0x5fe090b4
    parent = 0x10026320 = (control:XmRowColumn)
    xrm_name = 1366
    being_destroyed = False
    destroy_callbacks = 0x00000000
    constraints = 0x10026b98 => struct XmRowColumnConstraintRec {
      manager = struct XmManagerConstraintPart {
        unused = 0
        _SG_vendorExtension = 0
      }
      row_column = struct XmRowColumnConstraintPart {
        was_managed = True
        margin_top = 0
        margin_bottom = 0
        baseline = 0
        position_index = 0
        _SG_vendorExtension = 0
```

Figure 83. Widget Examiner

The Widget examiner contains the following items:

`Widget:` text field	Allows you to choose a widget to examine by entering the widget address.
`Name:` text field	Allows you to choose a widget to examine by entering the widget name.

`Parent button`	Allows you to move the parent of the currently selected widget.
`Previous` button	Moves you to the previously selected widget.
`Children...` button	Shows you the widget's children. (It is grayed out if the selected widget cannot have children.)
`Select...` button	Allows you to select the widget in the target process.

A.4.5 Tree Examiner

The Tree examiner (see Figure 84, page 178) displays the widget hierarchy.

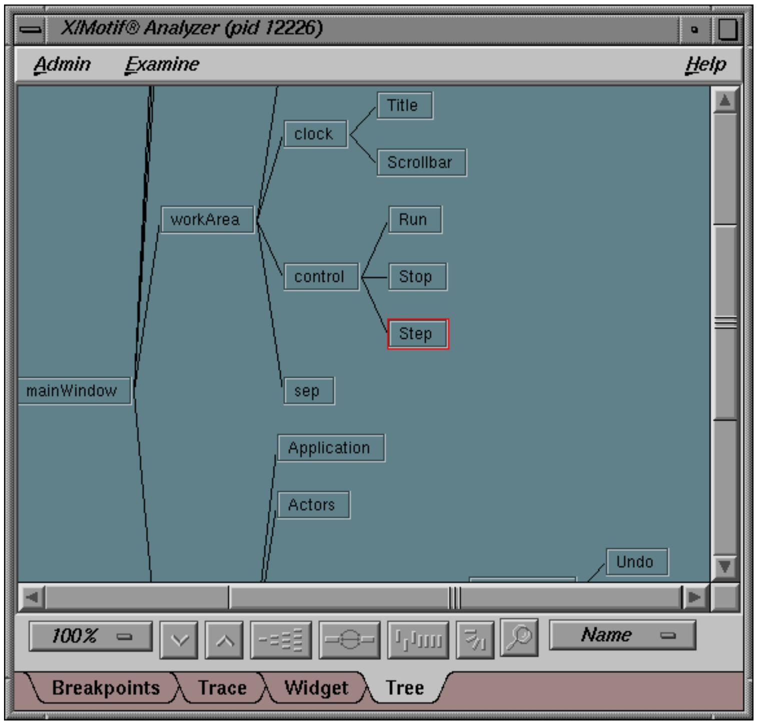


Figure 84. Tree Examiner

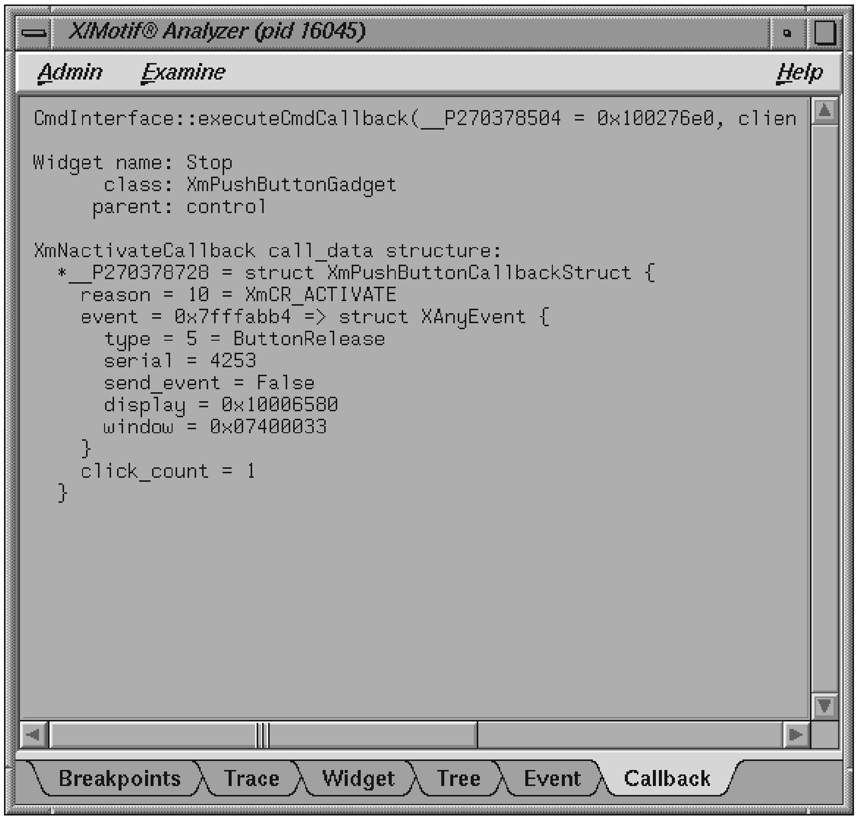
You may double-click a node to view that widget in the Widget examiner.

If the Tree examiner is currently selected, it will not automatically fetch the current widget tree each time the process stops. To force retrieval of the widget tree, select another examiner and then go back to the Tree examiner. Or, click on the `Tree` tab.

You may display the tree according to widget name, widget class, or widget ID. You can select this by choosing the appropriate option from the widget view type option button in the lower-right portion of the examiner.

A.4.6 Callback Examiner

The Callback examiner (see Figure 85, page 180) automatically appears when the process is stopped somewhere in a callback. It first displays the callstack frame for the callback function. Next, it displays information about the widget in the callback. Finally, it displays the proper callback structure contained in the `call_data` argument to the callback procedure, based on the widget type and the callback name.

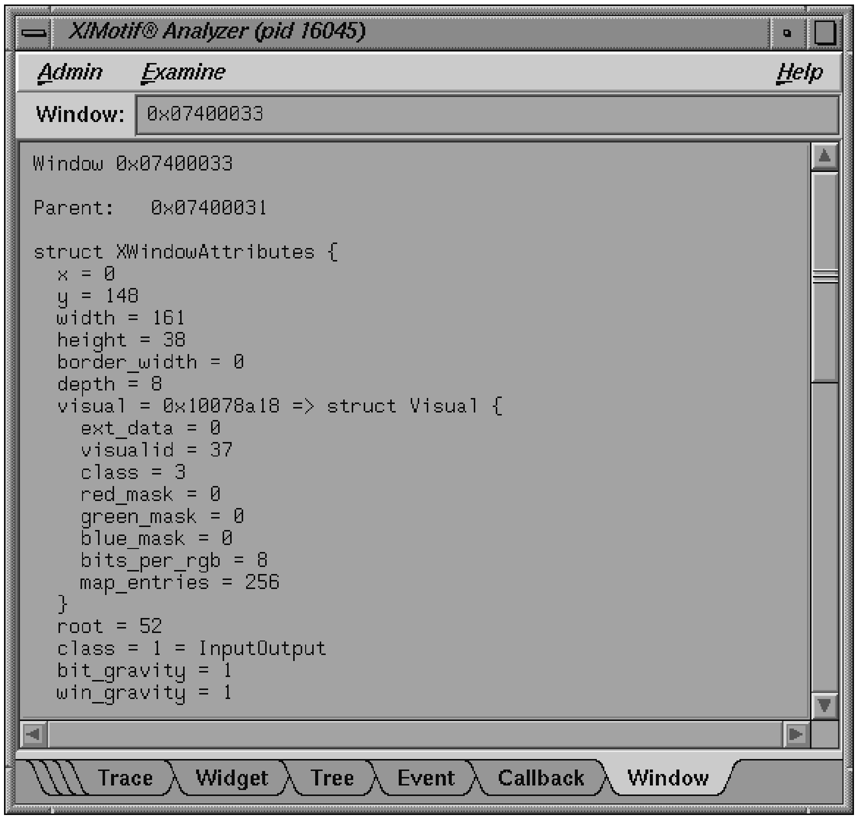


```
CmdInterface::executeCmdCallback(__P270378504 = 0x100276e0, clien

Widget name: Stop
      class: XmPushButtonGadget
     parent: control

XmNactivateCallback call_data structure:
  *__P270378728 = struct XmPushButtonCallbackStruct {
    reason = 10 = XmCR_ACTIVATE
    event = 0x7fffabb4 => struct XAnyEvent {
      type = 5 = ButtonRelease
      serial = 4253
      send_event = False
      display = 0x10006580
      window = 0x07400033
    }
    click_count = 1
  }
```

Figure 85. Callback Examiner

A.4.7 Window Examiner

The Window examiner (see Figure 86, page 181) displays window attributes for an X window. These are the attributes returned by `XGetWindowAttributes`, with decoding of the visual structure and enums and masks decoded. Additionally, the Window examiner shows the parent and children window IDs.



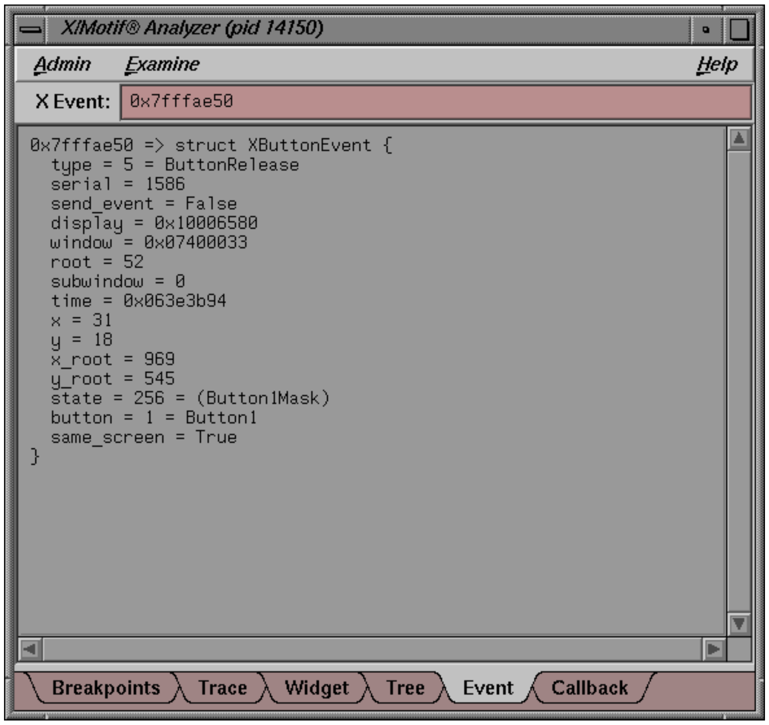
```
Window 0x07400033

Parent:   0x07400031

struct XWindowAttributes {
  x = 0
  y = 148
  width = 161
  height = 38
  border_width = 0
  depth = 8
  visual = 0x10078a18 => struct Visual {
    ext_data = 0
    visualid = 37
    class = 3
    red_mask = 0
    green_mask = 0
    blue_mask = 0
    bits_per_rgb = 8
    map_entries = 256
  }
  root = 52
  class = 1 = InputOutput
  bit_gravity = 1
  win_gravity = 1
```

Figure 86. Window Examiner

The Window examiner contains the `Window:` text field that displays the address of the window that is being examined. You may change to a different window by entering a new address and pressing `Enter`.

A.4.8 Event Examiner

The Event examiner (see Figure 87, page 182) displays the event structure for an `XEvent` pointer. The proper `XEvent` union member is used, and enums and masks are decoded.



```
0x7fffae50
```

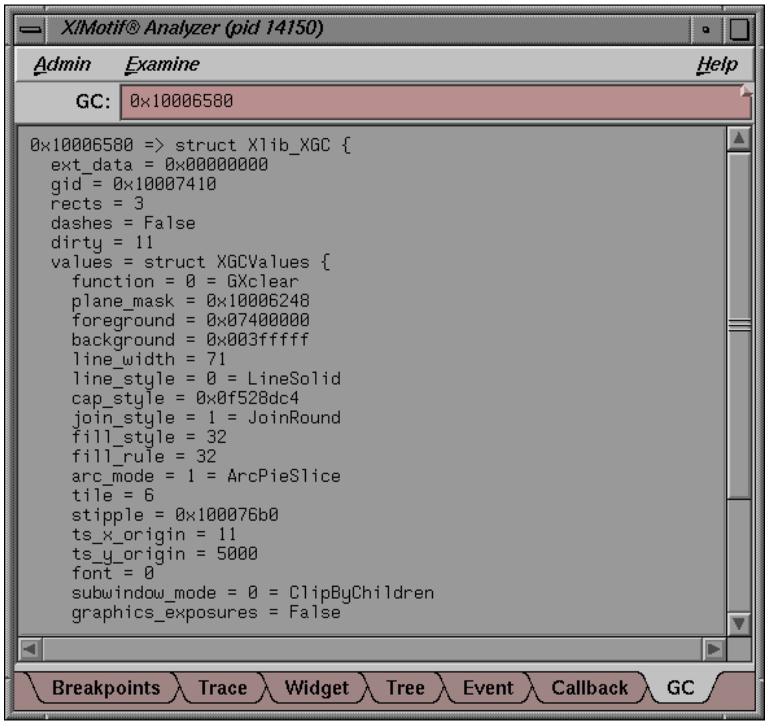
```
0x7fffae50 => struct XButtonEvent {
  type = 5 = ButtonRelease
  serial = 1586
  send_event = False
  display = 0x10006580
  window = 0x07400033
  root = 52
  subwindow = 0
  time = 0x063e3b94
  x = 31
  y = 18
  x_root = 969
  y_root = 545
  state = 256 = (Button1Mask)
  button = 1 = Button1
  same_screen = True
}
```

Figure 87. Event Examiner

The Event examiner contains the `X Event:` text field, which displays the address of the X event that is being examined. You may change to a different X event by entering a new address and pressing `Enter`.

A.4.9 Graphics Context Examiner

The Graphics Context examiner (see Figure 88, page 183) displays the X graphics context attributes that are cached by `Xlib` in the form of an `XGCValues` structure. Enums and masks are decoded.



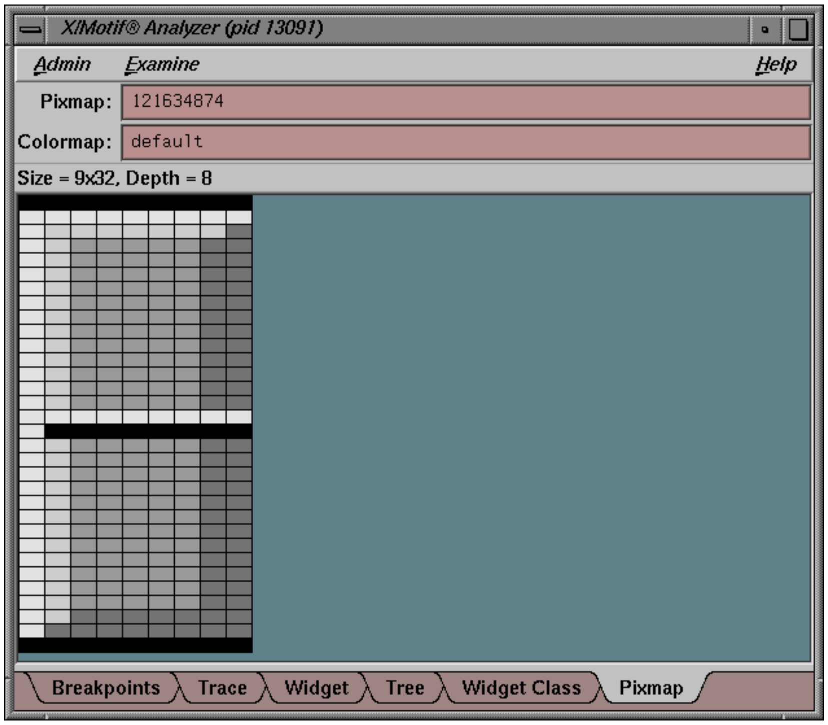
```
0x10006580 => struct Xlib_XGC {
  ext_data = 0x00000000
  gid = 0x10007410
  rects = 3
  dashes = False
  dirty = 11
  values = struct XGCValues {
    function = 0 = GXclear
    plane_mask = 0x10006248
    foreground = 0x07400000
    background = 0x003fffff
    line_width = 71
    line_style = 0 = LineSolid
    cap_style = 0x0f528dc4
    join_style = 1 = JoinRound
    fill_style = 32
    fill_rule = 32
    arc_mode = 1 = ArcPieSlice
    tile = 6
    stipple = 0x100076b0
    ts_x_origin = 11
    ts_y_origin = 5000
    font = 0
    subwindow_mode = 0 = ClipByChildren
    graphics_exposures = False
```

Figure 88. Graphics Context Examiner

The Graphics Context examiner contains the `GC:` text field that displays the address of the graphics context that is being examined. You may change to a different context by entering a new address and pressing `Enter`.

A.4.10 Pixmap Examiner

The Pixmap examiner (see Figure 89, page 184) displays basic attributes of an X pixmap, like size and depth. It also attempts to provide an ASCII display of small pixmaps, using the units digit of the pixel values.



Pixmap: 121634874

Colormap: default

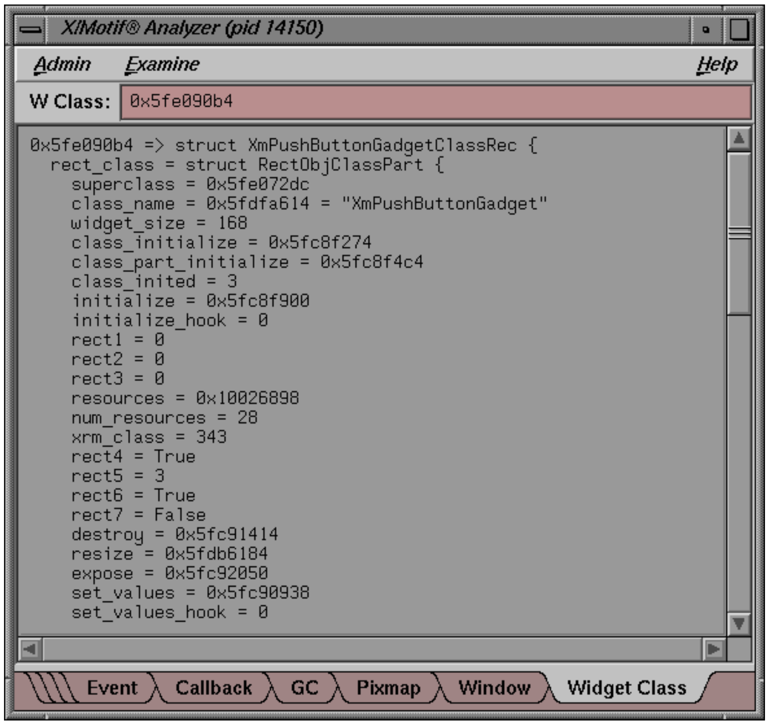
Size = 9x32, Depth = 8

Figure 89. Pixmap Examiner

The Pixmap examiner displays the contents of an X pixmap. Specify the X pixmap identifier and optionally, the X colormap identifier, by entering expressions in the two text areas at the top of the window. Use `default` as the colormap identifier to specify the default X colormap for your screen. In the actual pixmap display, left-click on a pixel to see the pixel value, position, and red-green-blue intensities.

A.4.11 Widget Class Examiner

The Widget Class examiner (see Figure 90, page 185) displays the `Xt` widget class structure, as well as the `Xt` inheritance implementation using nested C constructs.



```
0x5fe090b4
```

```
0x5fe090b4 => struct XmPushButtonGadgetClassRec {
  rect_class = struct RectObjClassPart {
    superclass = 0x5fe072dc
    class_name = 0x5fdfa614 = "XmPushButtonGadget"
    widget_size = 168
    class_initialize = 0x5fc8f274
    class_part_initialize = 0x5fc8f4c4
    class_inited = 3
    initialize = 0x5fc8f900
    initialize_hook = 0
    rect1 = 0
    rect2 = 0
    rect3 = 0
    resources = 0x10026898
    num_resources = 28
    xrm_class = 343
    rect4 = True
    rect5 = 3
    rect6 = True
    rect7 = False
    destroy = 0x5fc91414
    resize = 0x5fdb6184
    expose = 0x5fc92050
    set_values = 0x5fc90938
    set_values_hook = 0
```

Figure 90. Widget Class Examiner

The Widget Class examiner contains the `W Class:` text field, which displays the address of the widget class that is being examined. You may change to a different widget class by entering a new address and pressing `Enter`.

A.5 Project Management Window

In situations where you have many windows open from one or more projects, it is easy to lose track of projects. The `Project View` window can help you keep track of your projects.

To display the `Project View` window, pull down the `Admin` menu, select the `Project` submenu, and then select `Project View...` The `Project View` window opens. This window represents the components of a project as buttons. Elements from the same project are grouped within a rectangle. A dashed-line rectangle indicates the currently selected project. When you select a project, you can change its name or change the command.

A.5.1 Project View Admin Menu

The Show Applications and Show Window toggles in the Admin menu determine whether applications or individual windows display as project element buttons. The Rescan selection reevaluates the state of your projects and redisplays current elements. Exit closes the window.

A.5.2 Text Fields

The Project: field lets you enter a name to identify the curent project. The Command: field lets you invoke other tools to be included in the project.

A.5.3 Project Display Area

Elements of a project are represented as buttons. When a button protrudes from the screen, the item is currently iconified; when it is recessed, the item is displayed. You can toggle between the two modes.

A.5.4 Project Pop-up Menu

When you hold down the right mouse button inside a rectangle, the project pop-up menu displays. This menu lets you iconify, raise, or quit the item under the cursor or all items in the proects as a whole if the cursor is within the rectangle but not on any individual item.

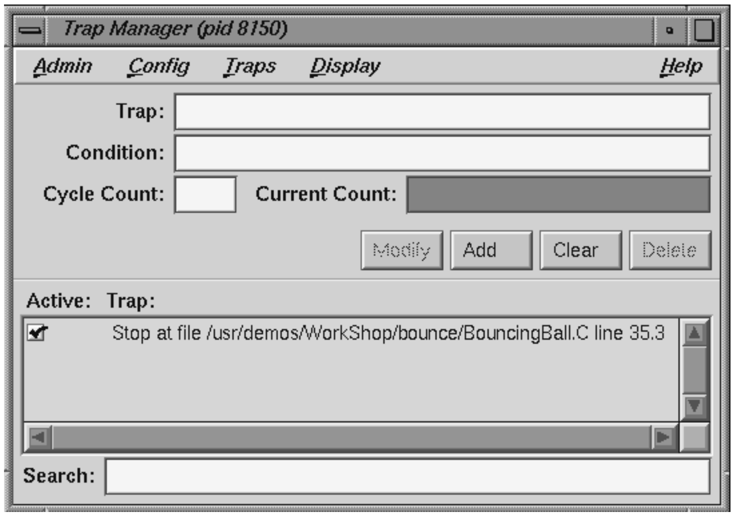
A.6 Trap Management Windows

In addition to setting traps through the Main View and the command line, the debugger provides you with three views specific to trap management:

- Trap Manager view
- Signal Panel view
- Syscall Panel view

A.6.1 Trap Manager

The Trap Manager allows you to set, edit, and manage traps (used in both the Debugger and Performance Analyzer). The X window is shown in Figure 91, page 187.



Stop at file /usr/demos/WorkShop/bounce/BouncingBall.C line 35.3

Figure 91. `Trap Manager` Window

The `Trap Manager` window contains the following items (besides the menu bar, which is discussed below):

`Trap:` text field	Contains a description of the trap.
`Condition:` text field	Contains the condition of the trap.
`Cycle Count:` text field	Displays the current cycle count.
`Current Count:` text field	Displays the current trap count.
`Modify` button	Allows you to change the selected breakpoint's settings.
`Add` button	Allows you to add a new breakpoint.
`Clear` button	Clears all the current breakpoint selections and text fields.
`Delete` button	Deletes the selected breakpoint.

`Trap:` display area	Contains a description of each trap, and a toggle to indicate whether or not the trap is active.
`Search:` text field	Allows you to perform an incremental, textural search for the trap list.

The `Trap Manager` window has a menu bar which contains the `Admin`, `Config`, `Traps`, `Display`, and `Help` menus. The `Admin` menu is the same as that described in Section A.3.1.1, page 150. The `Help` menu is the same as that described in Section A.1.10, page 139. The other menus are described in the following sections.

A.6.2 `Config` Menu

The `Config` menu contains the following items:

`Load Traps...`	Brings up the `File` dialog allowing you to load the traps from a file.
`Save Traps...`	Brings up the `File` dialog allowing you to save the current traps to a file.

A.6.3 `Traps` Menu

The `Traps` menu has options that allow you to set traps under a number of conditions. The following conditions are available:

- `At Source Line`
- `Entry Function`
- `Exit Function`
- `Stop Trap Default`
- `Sample Trap Default`
- `Group Trap Default`
- `Stop All Default`

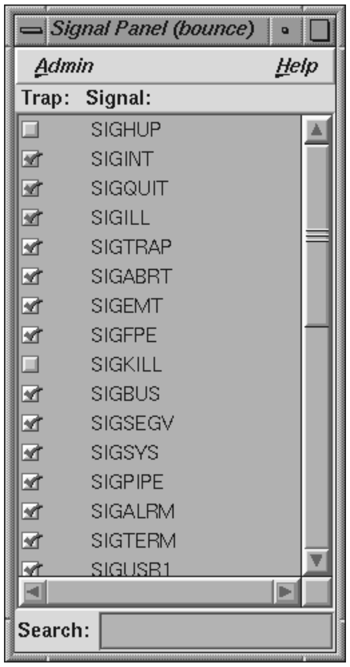
A.6.4 Display Menu

The Display menu contains the following item:

Delete All	Deletes all traps from the trap list.

A.6.5 Signal Panel

The Signal Panel displays the signals that can occur. You can specify which signals trigger traps and which are to be ignored. The Signal Panel is shown in Figure 92, page 189.



Trap:	Signal:
☐	SIGHUP
✓	SIGINT
✓	SIGQUIT
✓	SIGILL
✓	SIGTRAP
✓	SIGABRT
✓	SIGEMT
✓	SIGFPE
☐	SIGKILL
✓	SIGBUS
✓	SIGSEGV
✓	SIGSYS
✓	SIGPIPE
✓	SIGALRM
✓	SIGTERM
✓	SIGUSR1

Figure 92. Signal Panel

The Signal Panel contains an Admin menu (described in Section A.3.1.1, page 150) and a Help menu (described in Section A.1.10, page 139). Each signal trigger trap in the display has a toggle associated with it. In addition, the panel has a Search: field.

A.6.6 Syscall Panel

The Syscall Panel lets you set traps at the entry to or exit from system calls. The Syscall Panel is shown in Figure 93, page 190.

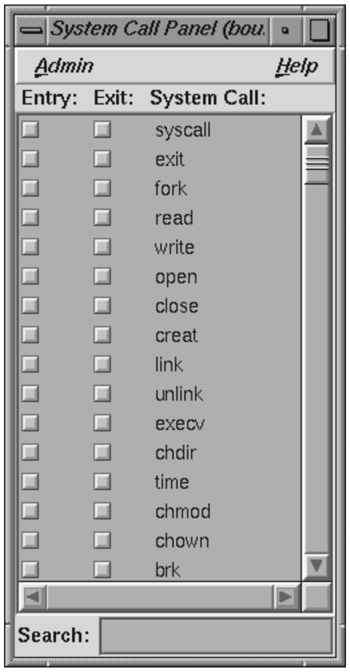


Figure 93. Syscall Panel

The Syscall Panel contains an Admin menu (described in Section A.3.1.1, page 150) and a Help menu (described in Section A.1.10, page 139). Each system call in the display has two toggle associated with it: one to set a trap on entry, one to set a trap on exit. In addition, the panel has a Search field.

A.7 Data Examination Windows

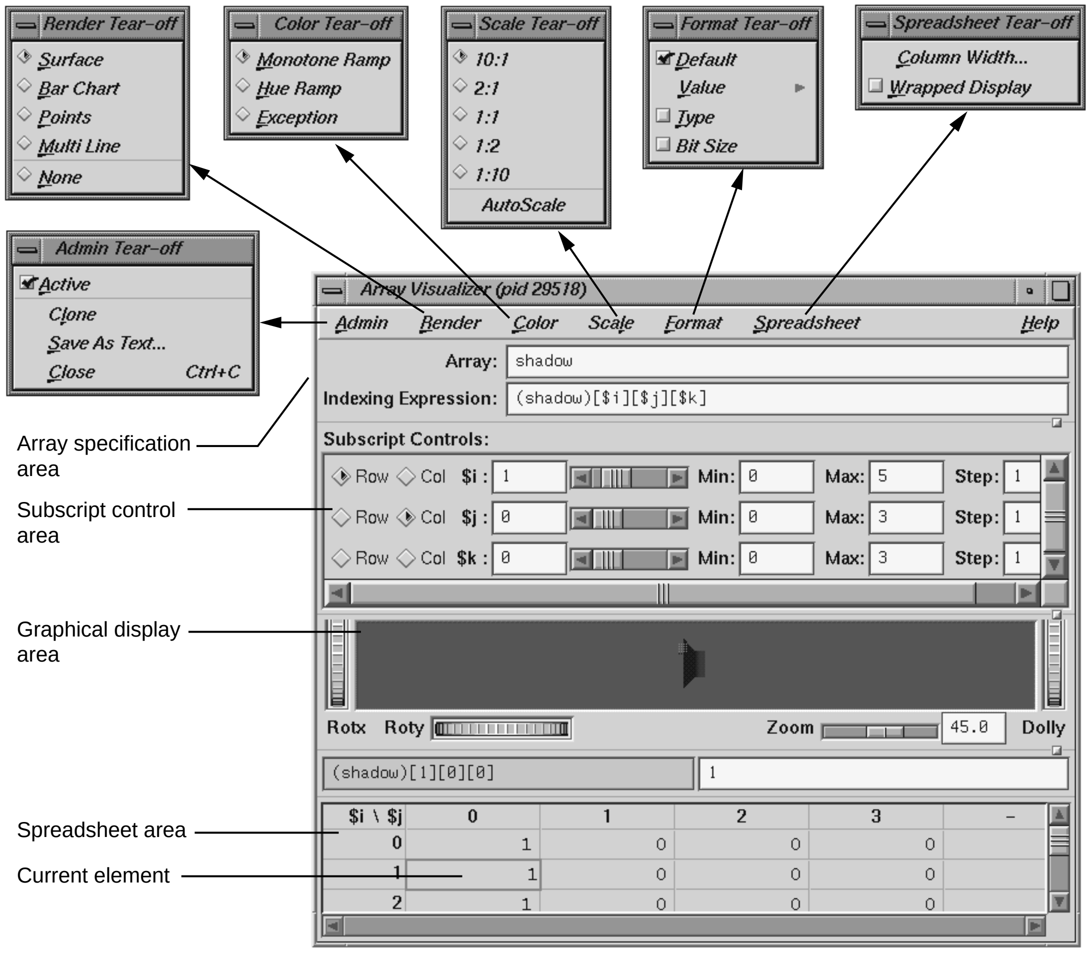
There are several windows that are used primarily to examine your debugging data:

- Array Browser Window, Section A.7.1, page 191

- `Call Stack View` Window, Section A.7.2, page 204
- `Expression View` Window, Section A.7.3, page 206
- `File Browser` Window, Section A.7.4, page 208
- `Structure Browser` Window, Section A.7.5, page 209
- `Variable Browser` Window, Section A.7.6, page 219

A.7.1 `Array Browser` Window

To examine data in an array variable, select `Array Browser` from the `Views` menu at a point in the process where the variable is present. The `Array Browser` lets you view elements in a multi-dimensional array (up to 100 x 100 elements), presented in a spreadsheet and graphically, if desired.



Array: shadow

Indexing Expression: (shadow)[$i][$j][$k]

[✓ Row] Col	$i :	1	Min:	0	Max:	5	Step: 1
Row [✓ Col]	$j :	0	Min:	0	Max:	3	Step: 1
Row Col	$k :	0	Min:	0	Max:	3	Step: 1

Zoom 45.0

(shadow)[1][0][0] 1

$i \ $j	0	1	2	3	–
0	1	0	0	0	
1	1	0	0	0	
2	1	0	0	0	

Figure 94. `Array Browser` with `Display` Menu Options

Note: The `Render`, `Color`, and `Scale` tear-off menus are available only if you have installed Open Inventor.

The *array specification area* lets you specify the variable and its dimensions. It consists of the following fields:

`Array:`	Lets you enter the name of the array variable. This entry is language-dependent.

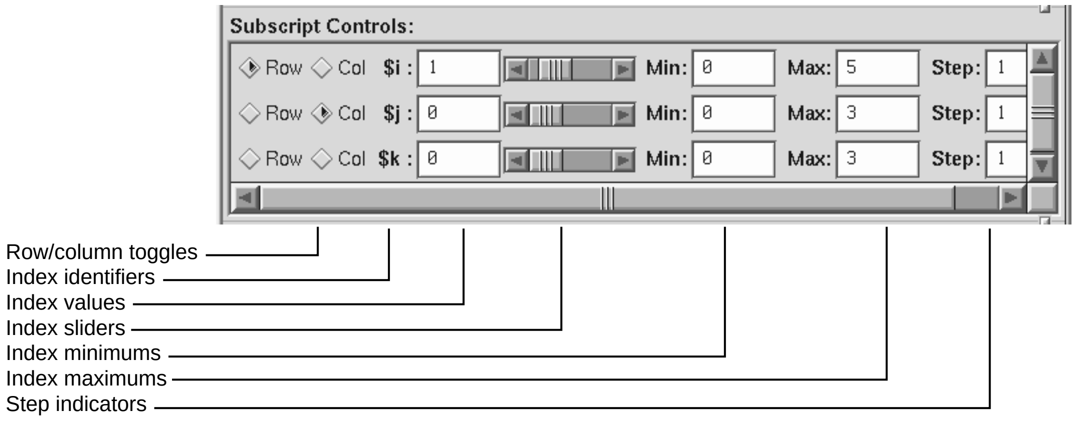
For Fortran, the expression may be an array or a dummy array variable name. If the last dimension of the array is unspecified (`*`), a subscript value of 1 is assumed initially.

For C and C++, the entry may be an array, a pointer, or an array pointer. If pointers are used, the expression is treated as though it were a single element, in which case you need to use the subscript controls to see more than the first element.

`Indexing Expression:` The expression used to view an element in the array. It is filled in automatically when you specify an array to view.

The expression supplied is language-specific. It represents the indexing expression used in the language to access a particular element. The subscripts are specified by special indexing variables (`$i`, `$j`, `$k`, and so forth) that can be manipulated in the subscript controls area.

The `Subscript Controls:` area serves two functions: it lets you control which elements in the variable are to be displayed, and it lets you shift the current element. The number of dimensions in the array governs the number of controls that are displayed. A close-up view of the subscript controls area appears in Figure 95, page 193.


Subscript Controls:

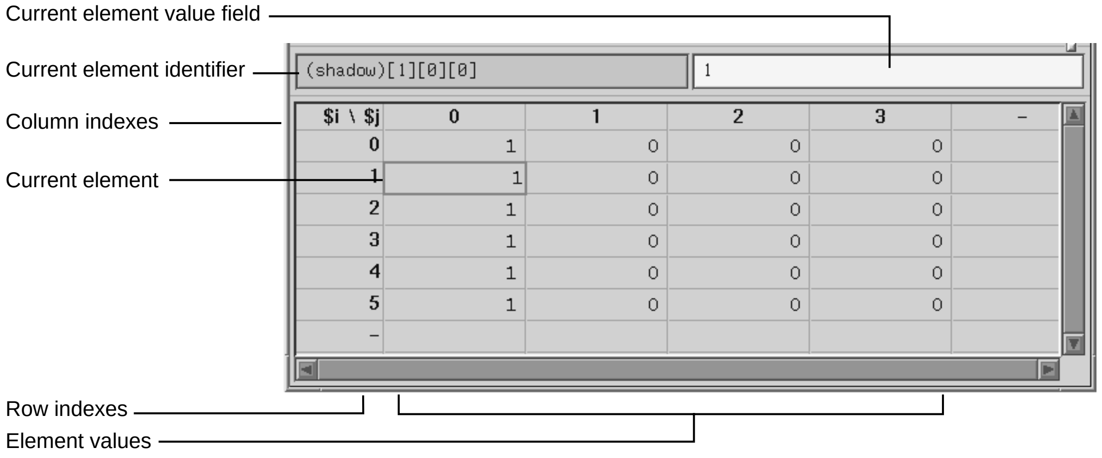
[✓ Row] Col	$i :	1	Min: 0	Max: 5	Step: 1
Row [✓ Col]	$j :	0	Min: 0	Max: 3	Step: 1
Row Col	$k :	0	Min: 0	Max: 3	Step: 1

Figure 95. `Subscript Controls:` Area in the `Array Browser`

The `Subscript Controls:` area provides the following features:

Row/column toggles	Control whether an index variable represents rows or columns (or neither) in the spreadsheet area. You are not limited by the number of vectors in an array, but you can only view a two-dimensional orthogonal slice of the array at a time.
Index identifiers	Indicate to which subscript the controls in the row refer.
Index values	Show the value of the subscript for the element currently in the focus cell. You can enter a different value if you wish.
Index sliders	Let you move the focus cell along the particular vector.
Index minimums	Identify the beginning visible element in a vector.
Index maximums	Identify the last visible element in a vector. If you have an unspecified array, you can use this field to specify the last element in the vector to be displayed in the spreadsheet.
Step indicators	Specifies the increment between adjacent elements in a vector to be displayed. A value of 1 displays consecutive data. Specifying some *n* greater than 1 lets you display every *n* element in a vector.
Control area scroll bars	Let you expose hidden portions of the subscript control area if your window is not large enough for viewing all of the controls.

The spreadsheet area is where numeric data is displayed. It can show two dimensions at a time (indicated in the upper left corner of the matrix). The column indexes run along the top of the matrix and the row indexes are displayed along the left column. The spreadsheet area has scroll bars for viewing data elements not currently visible in the viewing area. Figure 96, page 195, shows a closeup of the spreadsheet area.



(shadow)[1][0][0]

1

$i \ $j	0	1	2	3	–
0	1	0	0	0	
1	1	0	0	0	
2	1	0	0	0	
3	1	0	0	0	
4	1	0	0	0	
5	1	0	0	0	
–					

Figure 96. `Array Browser` Spreadsheet Area

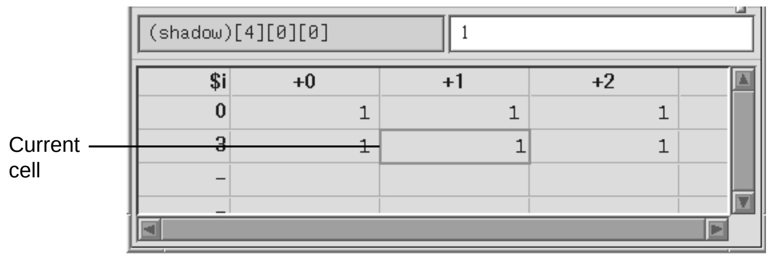
The current element is highlighted by a colored rectangle in the spreadsheet area. Its corresponding expression is shown in the current element identifier field, and the value is shown in the current element value field.

A.7.1.1 `Spreadsheet` Menu

The `Spreadsheet` menu lets you change the appearance of data in the spreadsheet area. It provides these selections:

`Column Width...`	Lets you specify the width of the spreadsheet cells in terms of characters. For instance, a value of 12 indicates that 12 characters, including punctuation and digits are viewable.
`Wrapped Display`	Lets you display a single dimension of an array wrapped around the entire spreadsheet area. The index value for an element is determined by adding the appropriate row index and column index values.

Figure 97, page 196, shows an example of a wrapped array. There is only one index `$i`. The current cell is element 4 in the array (by adding 3 and +1).



(shadow)[4][0][0] 1

$i	+0	+1	+2
0	1	1	1
3	1	1	1
–			
–			

Figure 97. Example of Wrapped Array

A.7.1.2 `Format` Menu

The `Format` menu displays a separate menu that you lets you display the elements in the following formats:

`Default` toggle	Toggles the default format.
`Value` submenu	Contains the following display toggles for formatted values: • `Decimal` • `Unsigned` • `Octal` • `Hex` • `Float` • `Char` • `String`
`Type`	Allows listing by data type.
`Bit Size`	Allows listing by bit size.

The graphical display area presents array data in a three-dimensional graph in one of the following formats:

- Surface (polyhedron)
- Bar chart
- Points

- Multiple lines (array vectors)

A.7.1.3 `Render` Menu

Note: The `Render` tear-off menu is available only if you have installed Open Inventor.

You select the graphical display mode through the `Render` menu. The `Render` menu has the following options:

`Surface`	Exhibits the data as a solid using the data values as vertices in a polyhedron.
`Bar Chart`	Presents the data values as 3-D bar charts.
`Points`	Simply plots the data values in 3-D space.
`Multi Line`	Plots and connects the data values in each row.
`None`	Lets you display with no graphical display, in effect turning off graphical display mode.

A.7.1.4 `Color` Menu

Note: The `Color` tear-off menu is available only if you have installed Open Inventor.

If the `Color` menu is grayed out when the `Array Visualizer` window first opens, select the `Surface` option of the `Render` menu. The `Color` menu provides the following options:

`Monotone Ramp`	Displays the data values in a single tone, with lower numbers being darker and higher values lighter in tone.
`Hue Ramp`	Displays the data values in a spectrum of colors ranging from blue (lowest values) through green, yellow, orange, and red (highest values).

`Exception`	Lets you flag certain conditions by color, usually for the purpose of spotting bad data. When you select `Exception`, the controls shown in Figure 98, page 198 appear in the window.

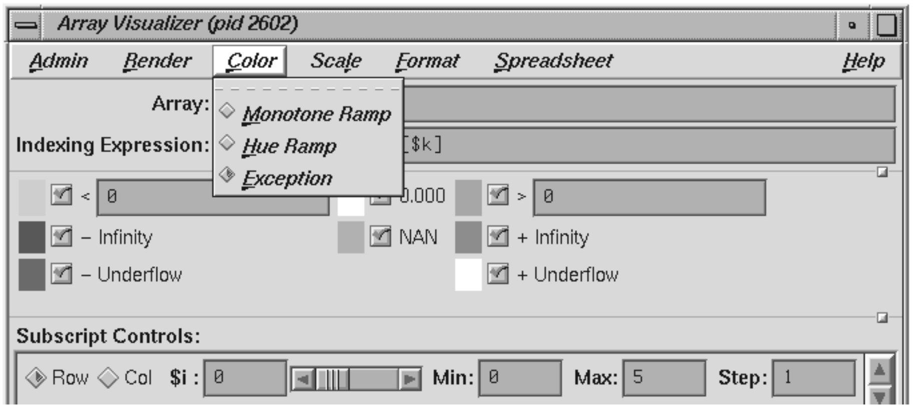


Figure 98. Color Exception Portion of `Array Browser` Window

Thus, you can highlight data values less than or greater than specified values, values of plus or minus infinity, values of plus or minus underflow, zero values, and `NaN` (not a number) values.

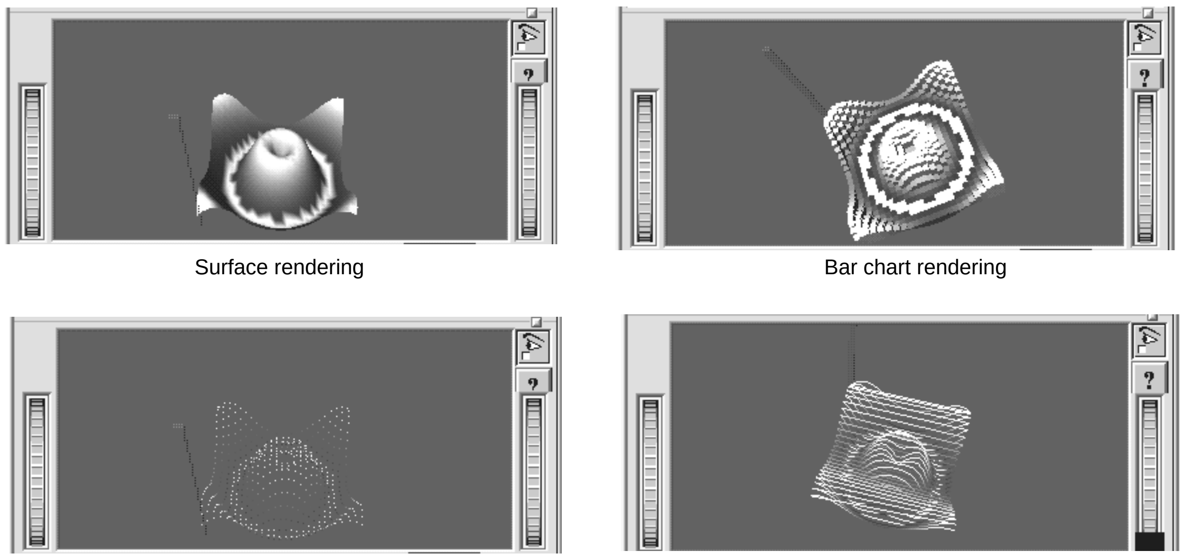


Figure 99. `Array Browser` Graphic Modes

A.7.1.5 `Scale` Menu

Note: The `Scale` tear-off menu is available only if you have installed Open Inventor.

If the `Scale` menu is greyed out when the `Array Visualizer` window first opens, select the `Surface` option of the `Render` menu.

The `Scale` menu provides options for changing the ratio of the *z*-dimension, which represents the value of the element. The number on the left represents the value of the *x* and *y*-dimensions (which are always the same as each other). The number on the right is the *z* dimension.

Manipulating the *z*-dimension affects the ease of spotting differences in values. If your data is scattered over a narrow range of values, you may wish to heighten the graph by selecting `10:1` as your scale; this exaggerates the values in the *z*-dimension. If your data is in a wide range, selecting `1:2` or `1:10` as the scale will minimize the differences, flattening the graph.

A.7.1.6 Examiner Viewer Controls

The graphical display uses controls and menus from Examiner Viewer. Examiner Viewer is based on a camera metaphor and borrows terms from the film industry, such as *zoom* and *dolly*, in naming its controls. The graphical display area of the window is shown in Figure 100, page 201, with its main controls and menus. Note that the buttons on the upper right side of the graphical display area may not be visible if the area is too small; you can expose them by moving either the upper or lower sash to enlarge the display area.

Examiner Viewer provides these controls for viewing the graph. The right side buttons provide the following functions:

`view mode`	Toggles between a view-only mode (closed eye) and manipulation mode (open eye). In view-only mode, the cursor appears as an arrow and the graph cannot be moved. Clicking on a portion of the graph selects the corresponding array element in the spreadsheet. In manipulation mode, the cursor appears as a hand and you can move the graph. Dragging the graph with the left mouse button down moves the graph in any direction as if it were in a trackball; a quick movement spins the graph. Dragging the graph with the left mouse button and the `Ctrl` key rolls (rotates) the graph in the plane of the screen. Dragging the graph with the middle mouse button moves it without changing the viewing angle. If you drag the graph with both the left and middle mouse buttons down, the graph will appear to move into or out of the window (this is the same as the `dolly thumbwheel`, which is described in this section).

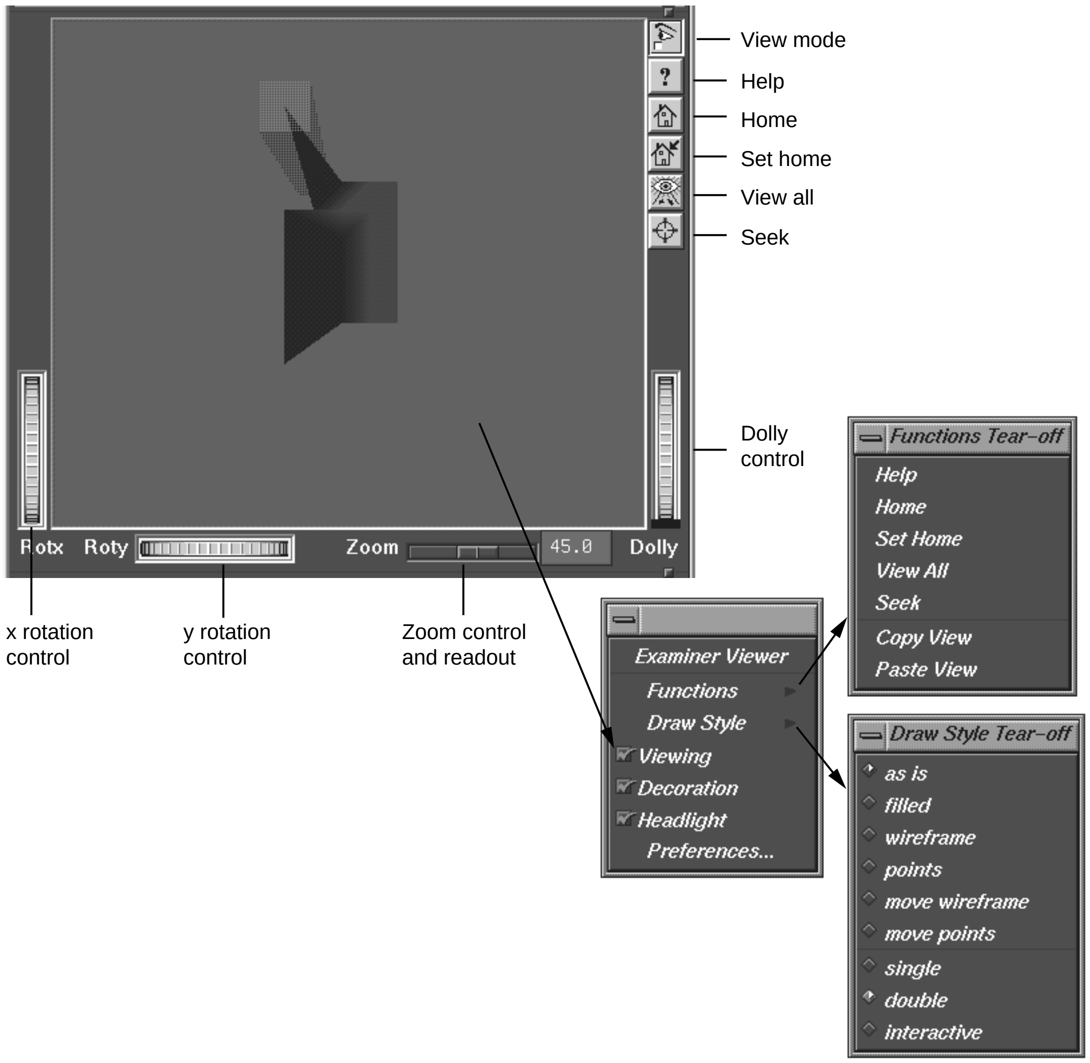


Figure 100. Examiner Viewer with Controls and Menus

`help`	Runs a special help system containing Inventor Viewer information.
`home`	Repositions the graph in its original viewing position.

`set home`	Changes the home (original viewing) position for subsequent use of the `home` button.
`view all`	Repositions the display area so that the entire graph is visible.
`seek`	Provides a special cursor that lets you reposition the graph in the center of the display area or lets you center the view on a point you select with the cursor. See `Seek to point <or object>` in the Preferences dialog box.

The following controls let you move the graphic display:

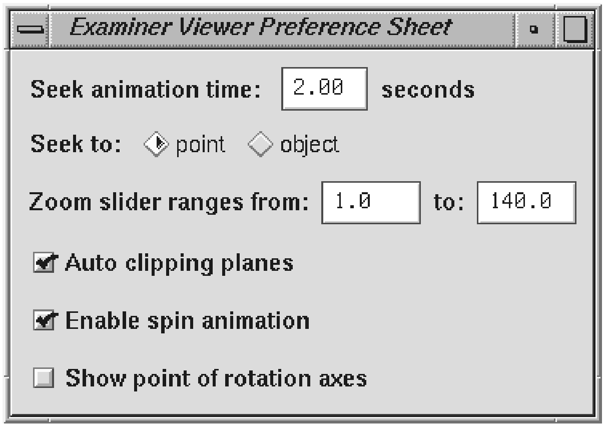
`x rotation thumbwheel`	Rotates the graph around its x-axis.
`y rotation thumbwheel`	Rotates the graph around its y-axis.
`zoom slider and readout`	Changes the size of the graph by scaling it.
`dolly thumbwheel`	Changes the size of the graph and adjusts the angles to maintain perspective. The dolly control simulates moving the viewing camera back and forth with respect to the graph.

A.7.1.7 `Examiner Viewer` Menu

You access the `Examiner Viewer` menu by holding down the right mouse button in the graphical display area. The `Examiner Viewer` menu provides the following options:

`Functions`	Displays a submenu with the selections `Help`, `Home`, `Set Home`, `View All`, and `Seek`, which are the same as the right mouse button controls described in the previous section, and the `Copy View` and `Paste View` selections. These operate like standard copy and paste editing commands, enabling you to transfer graphs.
`Draw Style`	Displays a submenu that controls how the graph is displayed. The options `as is`, `filled`, `wireframe`, and `points` control how the graph is displayed when it is static. These override any `Render` menu selections. The `move wireframe`

	and `move points` options control how the graph is displayed while in motion. The selections `single`, `double`, and `interactive` refer to buffering techniques used in moving the graph. These affect the smoothness of the movement.
`Viewing`	The same as the `view mode` button described in the previous section. When it is off, you can select points from the graph to display in the spreadsheet but cannot move the graph. When on, it lets you manipulate the graph.
`Decoration`	Displays the right side buttons when it is on and hides them when it is off.
`Headlight`	Controls the shadow effect on the graph. When it is on, the light appears to come from the camera.
`Preferences`	Causes the `Examiner Viewer Preferences Sheet` dialog to display.


Examiner Viewer Preference Sheet

Seek animation time: 2.00 seconds

Seek to: [✓ point] object

Zoom slider ranges from: 1.0 to: 140.0

[✓] Auto clipping planes

[✓] Enable spin animation

[ ] Show point of rotation axes

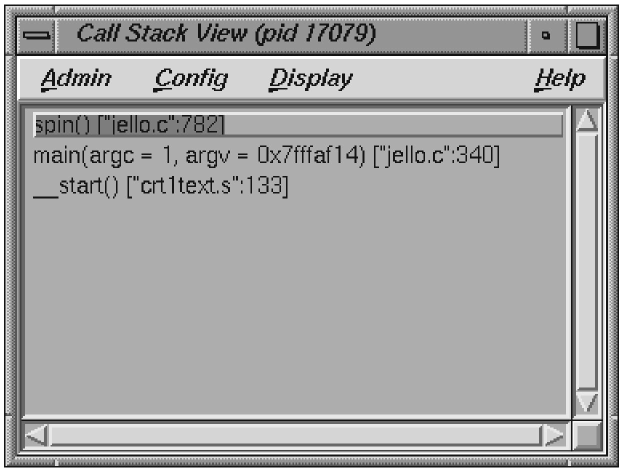
Figure 101. Examiner Viewer Preferences Dialog

`Seek animation time`	Lets you specify the time it takes for the graph to be repositioned after you change the seek point. See also `Seek to point <or object>`.

`Seek to point <or object>`	Lets you change the view of the graph to its center (object) or to a point in the graph that you specify with the seek cursor.
`Zoom slider ranges from`	Lets you change the Zoom range, that is, the distance that the object appears to be away from the front of the window.
`Auto clipping planes`	Centers the graph in your view if enabled. If disabled, it provides controls for removing data from visibility at either end of the z-axis. This is useful if you wish to focus on data above or below a set value.
`Enable spin automation`	Lets you spin the graph. You grab the graph with the mouse, move it quickly in the desired direction, and release the mouse button. The graph spins as if in a trackball.
`Show point of rotation axes`	Displays a set of three axes. You can move the graph around the x and y axes using the thumbwheel controls described in the previous section. When this option is on, you can set the size of the axes in pixels.

A.7.2 `Call Stack View` Window

The `Call Stack View` (Figure 102, page 205) window displays call stack entries when a process has stopped.



```
spin() ["jello.c":782]
main(argc = 1, argv = 0x7fffaf14) ["jello.c":340]
__start() ["crt1text.s":133]
```

Figure 102. `Call Stack View`

The source display has two special annotations:

- The location of the current program state is indicated by a large arrow representing the PC.
- The location of the call to the function selected in the `Call Stack View` window is indicated by a small arrow representing the current context. The source line becomes highlighted.

The `Call Stack View` contains its own menu bar, which contains the `Admin`, `Config`, `Display`, and `Help` menus. The `Admin` menu is the same as that described in Section A.3.1.1, page 150. The `Help` menu is the same as that described in Section A.1.10, page 139. The other menus are described in the following sections.

A.7.2.1 `Config` Menu

The `Config` menu contains the following option:

Preferences...	Launches the Call Stack View Preferences dialog that allows you the option of setting the maximum depth of the Call Stack View.

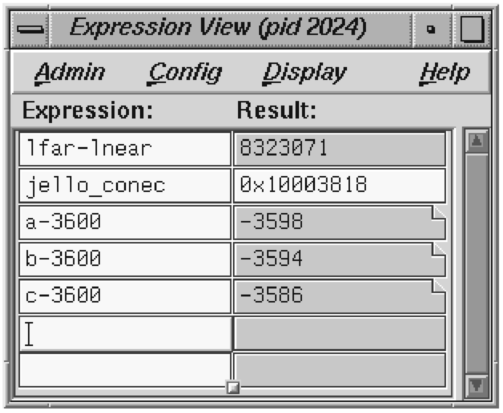
A.7.2.2 Display Menu

The Display menu contains the following toggles:

Arg Values	Allows you to set the argument values.
Arg Names	Allows you to set the argument names.
Arg Types	Allows you to set the argument types.
Location	Allows you to set the function location.
PC	Allows you to set the PC.

A.7.3 Expression View Window

The Expression View window is shown in Figure 103, page 206. Expression View displays a collection of expressions that are evaluated each time the process stops or the context changes.



Expression:	Result:
lfar-lnear	8323071
jello_conec	0x10003818
a-3600	-3598
b-3600	-3594
c-3600	-3586

Figure 103. Expression View

In addition to the items on the menu bar, the window has two pop-up menus: the Language menu and the Format menu. The Admin menu is the same as that described in Section A.3.1.1, page 150. The Help menu is the same as that described in Section A.1.10, page 139. The other menus are described in the following sections.

A.7.3.1 `Config` Menu

The `Config` menu contains the following options:

`Load Expressions...`	Launches the `File` menu that allows you to choose source file from which to load your expressions.
`Save Expressions...`	Launches the `File` menu that allows you to choose a file to which you can save your expressions.

A.7.3.2 `Display` Menu

The `Display` menu contains the following option:

`Clear All`	Clears all fields in the view.

A.7.3.3 `Language` Pop-up Menu

The `Language` pop-up menu contains three button that allow you to select one of three languages for evaluation: C, C++, or Fortran. The `Language` pop-up is invoked by holding down the right mouse button while the cursor is in the `Expression` column.

A.7.3.4 `Format` Pop-up Menu

The `Format` pop-up menu is displayed by holding down the right mouse button in the `Result` column.

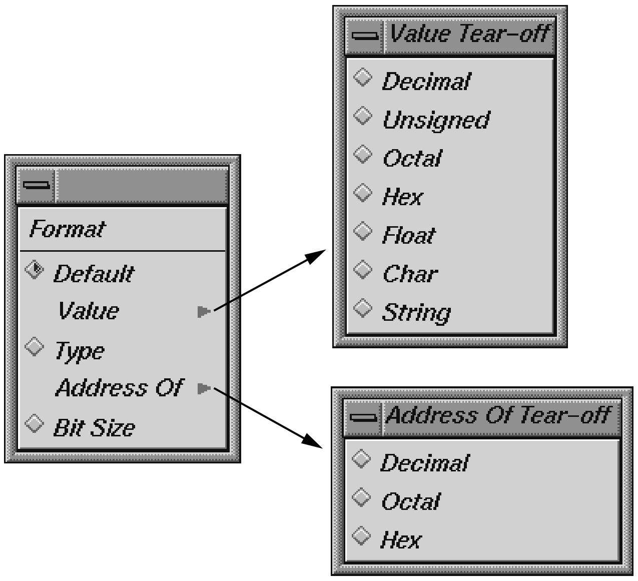


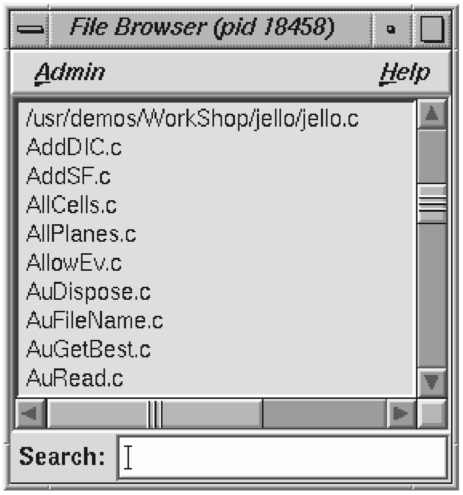
Figure 104. Expression View Format Popup with Submenus

The `Format` popup contains the following options:

`Default`	Sets the format to the default values.
`Value`	Displays a submenu from which you can select a value of `Decimal`, `Unsigned`, `Octal`, `Hex`, `Float`, `Char`, or `String` type.
`Type`	Displays a submenu from which you can select a type of `Decimal`, `Octal`, or `Hex`.
`Bit size`	Sets the format to `Bit Size`.

A.7.4 `File Browser` Window

The `File Browser` window displays a scrollable list of source files used by the current executable. Double-click a file in the list to load it directly into the source display area in Main View or `Source View`. The `Search:` field lets you find files in the list quickly.



/usr/demos/WorkShop/jello/jello.c
AddDIC.c
AddSF.c
AllCells.c
AllPlanes.c
AllowEv.c
AuDispose.c
AuFileName.c
AuGetBest.c
AuRead.c

Figure 105. `File Browser` Window

The `File Browser` contains an `Admin` menu (described in Section A.3.1.1, page 150) and a `Help` menu (described in Section A.1.10, page 139). In addition, the browser has a `Search:` field.

A.7.5 `Structure Browser` Window

The `Structure Browser` window lets you examine data structures and the relationships of the data within them. It displays complex data structures as separate graphical objects, using arrows to indicate relationships. A typical `Structure Browser` example is shown in Figure 106, page 210, with the `Config`, `Display`, `Node`, and `Format` menus displayed.

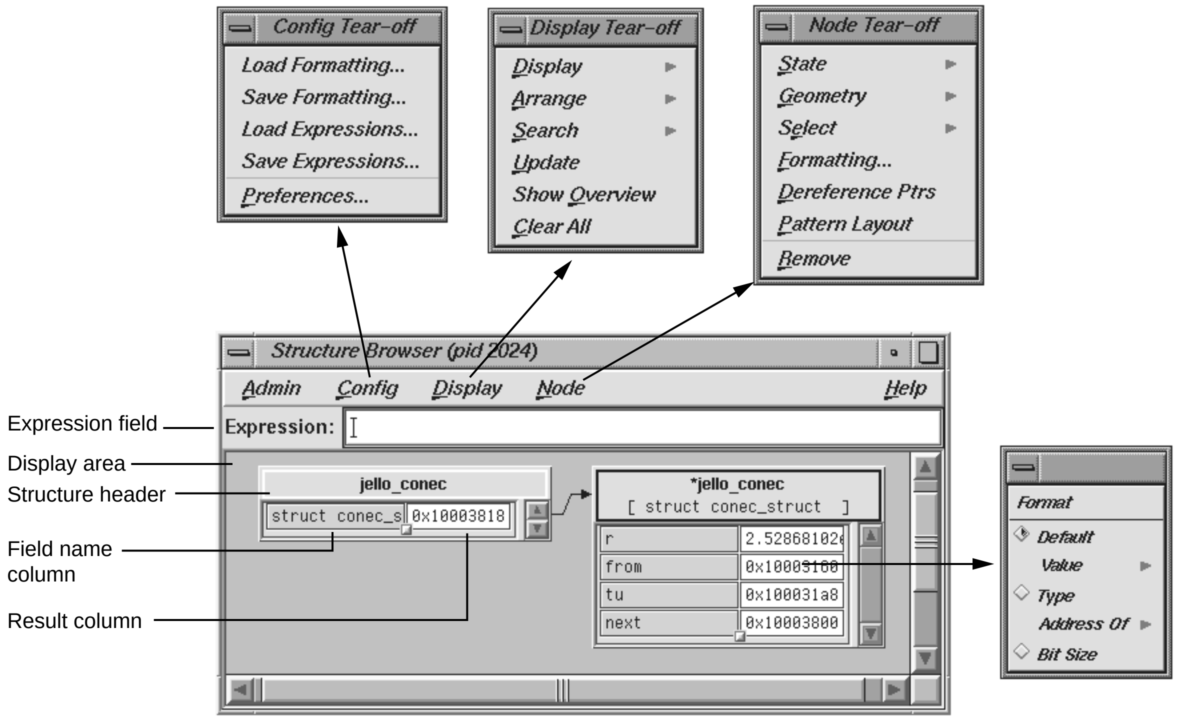


Figure 106. `Structure Browser` with Menus Displayed

The structure name is entered in the `Expression` field. It then appears as an object or set of objects in the display area in the lower portion of the window. Each structure has a header identifying the structure, color coded by data type. Below the header are two columns: the left displays the field name and the right displays the field’s value. If the structures to be displayed exceed the size of the `Structure Browser` window, scroll bars appear.

Menu options let you change the way the data displays. The following menus are available:

`Config`	Used for saving and reusing type-specific formats and expressions. You can also set preferences regarding how objects of a given type are to be displayed.
`Display`	Provides operations for all objects in the display area.

`Node`	Provides operations for selected objects in the display area only.
`Format`	Lets you change or reformat a specific value in the result column. It is a pop-up menu that is accessed by holding down the right mouse button while the cursor is over the result column.

A.7.5.1 Using the `Structure Browser Overview` Window to Navigate

WorkShop provides the `Structure Browser Overview` window (from the `Show Overview` option in the `Display` menu) as another way to navigate around the display.

This window is a reduced-scale view of the requested structures. The structures are represented by solid rectangles color-coded by data type. The relative position of the currently visible area is represented by a transparent rectangle. This rectangle can be dragged (using the left mouse button) to change the display of the `Structure Browser`. Clicking the left mouse button in an area of this window repositions the currently visible area.

A.7.5.2 Entering Expressions

The `Structure Browser` accepts any valid expression. If the result type is simple, a structure displays showing the type and value. If the result type is a pointer, it is automatically dereferenced until a non-pointer type is reached. If the result type is a structure or union, an object is displayed showing the structures' fields and their values. After the expression is entered, the `Expression` field clears. The `Structure Browser` can display unrelated structures at the same time; you simply enter new structures by using the `Expression` field.

The `Expression` field is also used to enter strings used in searches.

A.7.5.3 Working in the `Structure Browser` Display Area

Within the display area, you select objects by clicking in the node headers. Shift-clicks add the selected object to the current selection. You can drag selected objects using the middle mouse button.

Clicking the right button while the cursor is in the right column of an object displays the `Format` menu, which is used to change the display. You can set a default format or request that results be displayed by value, type, address, or size in bits.

Holding down the right button in the header of an object brings up the `Node` pop-up menu, which is the same as the `Node` menu in the menu bar. It is used to change the way selected objects are displayed. When you left-click in the header of an object, it turns on the resizer, which lets you change the size of the object. Left-clicking the handle resizes the object and middle-clicking moves it.

Graphical arrows show the pointer relationships among the displayed structures. If a pointer field is not visible in a structure, its arrow tail is displayed at the top or bottom of the scrolling area for fields. Otherwise, its tail is adjacent to its field.

Double-clicking a value field (right column) for a pointer changes the display so that the data structure it points to is displayed.

Double-clicking a member field (left column) puts the full expression for that member in the `Expression` field.

A.7.5.4 `Structure Browser Display` Menu

The `Display` menu controls the way structures appear in the display area. The `Display` menu provides the following options:

`Display` Determines contents of the display. The `Display` option has the following two options:

- `Expression` — Displays the structure of the expression entered in the `Expression` field.
- `Selection` — Displays the structure based on the text from the current selection.

`Arrange` Rearranges the currently selected nodes. `Arrange` has the following two options: (See Figure 107, page 213.)

- `Tree` — Arranges nodes into a tree-type formation, that is, the hierarchy descends from left to right and child structures are shown as branches to the right of the parent.
- `Linked List` — Arranges nodes into a linked list formation, that is, horizontally.

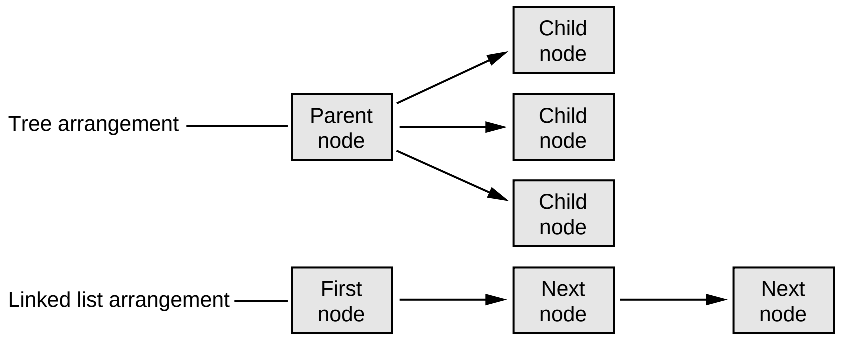


Figure 107. Tree and Linked List Arrangements of Structures

`Search`	Lets you select structures containing the string specified in the `Expression` field. `Search` has the following four options: • `Name` — Selects structures whose names contain the specified string. • `Type` — Selects structures whose types contain the specified string. • `Field Name` — Selects structures that have a field whose name contains the specified string. • `Value` — Selects structures that have a field value containing the specified string.
`Update`	Explicitly updates the displayed structures. This happens automatically in the current `Structure Browser` when the process stops. This can be used in an inactive `Structure Browser` to update it. It can also be used to update the display after changes have been made in other Debugger views.
`Show Overview`	Brings up the `Structure Browser Overview` window.

`Clear All`	Clears all structures from the display area.

A.7.5.5 `Node` Menu

The `Node` menu has the following options that apply to currently selected objects:

`State`	Controls the display of nodes. There are three options: • `Iconic` — Displays the node header only. • `Normal` — Uses the default chart display but hides those fields selected to be invisible. • `Detail` — Uses the default chart display and shows all fields.
`Geometry`	Manages graphical objects in the display area. There are four options: • `Minimize` — Sets the vertical size of an object to the default minimum number of fields. The initial default is four fields but can be changed through either the `Formatting` selection from the `Node` menu or the `Preferences...` selection from the `Config` menu. • `Maximize` — Displays the object as large vertically as necessary to fit all of the fields. • `Raise` — Raises the selected object(s) to the top of the display. • `Lower` — Lowers the selected object(s) to the bottom of the display.
`Select`	Lets you select objects in various ways. There are six options: • `Parents` — Selects all objects that have pointers pointing to a selected object. • `Children` — Selects all objects pointed to by any fields in a selected object.

- `Ancestors` — Selects all objects pointed to a selected object or pointing to an object that has a descendant pointing to a selected object.
- `Descendant` — Selects all objects pointed to by any fields in a selected object or pointed to by any children of a selected object.
- `Family` — Selects all ancestors and descendants of a selected object.
- `All` — Selects all objects.

`Formatting`	Brings up the type formatting dialog for this type.
`Dereference Ptrs`	Dereferences any pointers in selected objects.
`Pattern Layout`	Displays selected structures that are connected by pointers to position related structures in the same way.
`Remove`	Removes selected object from the display.

A.7.5.6 Formatting Fields

Each field in a data structure has certain display characteristics. These can be specified for all objects in the `Structure Browser Preferences` dialog box or for type-specific objects only in the `Structure Browser Type Formatting` dialog box. To display the `Structure Browser Preferences` dialog box, select `Preferences...` from the `Config` menu (see Figure 108, page 216).

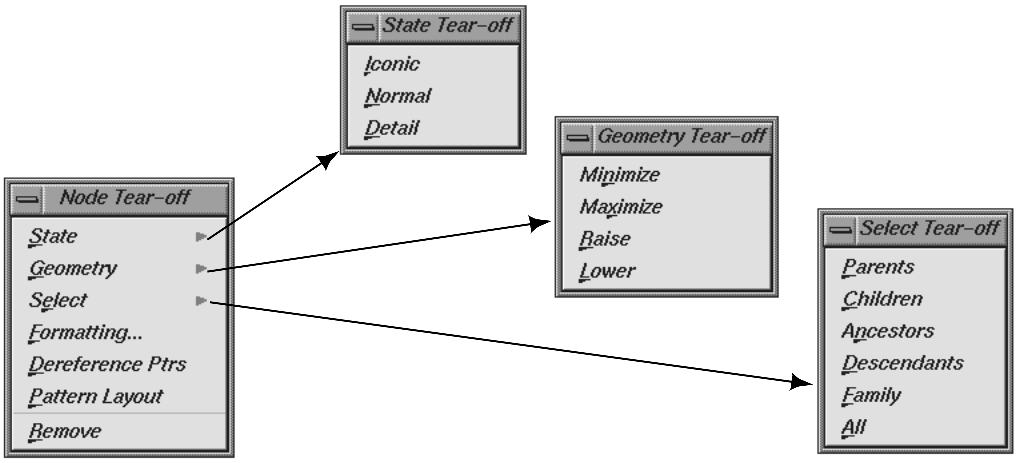
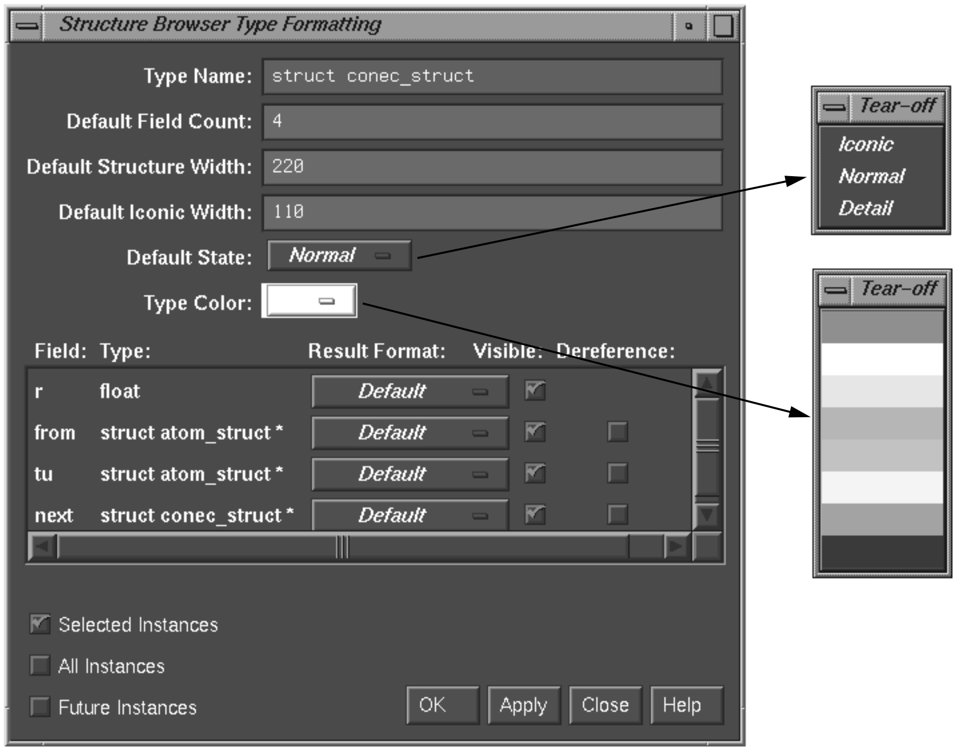


Figure 108. `Structure Browser Preferences` Dialog

The dialog has the following fields:

`Default Structure Field Count`	Sets the number of fields to be displayed initially.
`Default Structure Width`	The width in pixels of the object.
`Default Iconic Width`	The width in pixels of the object when it is in iconic form.
`Automatic Dereference Limit`	Limits the number of structures that are automatically dereferenced.
`Dereference Ptrs By Default`	Toggles automatic dereferencing on and off.

To bring up the `Structure Browser Type Formatting` dialog box, select the set of structures under consideration and select `Node Formatting` from the `Node` menu (see Figure 109, page 217).



Structure Browser Type Formatting

Type Name: struct conec_struct

Default Field Count: 4

Default Structure Width: 220

Default Iconic Width: 110

Default State: Normal

Type Color:

Field:	Type:	Result Format:	Visible:	Dereference:
r	float	Default	✓	
from	struct atom_struct *	Default	✓	☐
tu	struct atom_struct *	Default	✓	☐
next	struct conec_struct *	Default	✓	☐

✓ Selected Instances

☐ All Instances

☐ Future Instances

Figure 109. `Structure Browser Type Formatting` Dialog

The dialog box has the following fields:

`Type Name`	Displays the current data type.
`Default Field Count`	Lists number of fields to be displayed initially for objects of that type.
`Default Structure Width`	Displays the width in pixels of the object.
`Default Iconic Width`	Displays the width in pixels of the object when it is in iconic form.
`Default State`	Brings up a pop-up menu that lets you specify whether structures are first displayed as icons (`Iconic`), with the minimum number of fields

	displayed (`Normal`) or with all fields displayed (`Detail`).
`Type Color`	Provides a submenu for color coding. It lets you select a color for the header and overview rectangles for objects of a given type.

For structure and union types, the list box shows all the fields with their types. For each field, you can change the result format to one of the following types:

- Default
- Decimal
- Unsigned
- Octal
- Hex
- Float
- Char
- String
- Type
- Dec addr
- Oct addr
- Hex addr
- Size in Bits

You can also specify whether a field is visible in normal state, and if it is a pointer field, whether it should be automatically dereferenced.

Once you specify the format for this type, you can apply it to any combination of the following through the toggle buttons in the bottom left portion of the window:

- Selected instances
- All existing instances
- Any future instances of this type

A.7.6 Variable Browser Window

The Variable Browser window lets you view and change the values of local variables and arguments at a specific point in a process. (Global variables can be viewed or changed using Expression View or the Evaluate Expression selection from the Data menu for one-shot evaluations.) In addition to providing values, the Variable Browser is useful for getting a quick list of the local variables in a scope without having to search for their names. A sample Variable Browser window with the Language and Format menus displayed is shown in Figure 110, page 220.

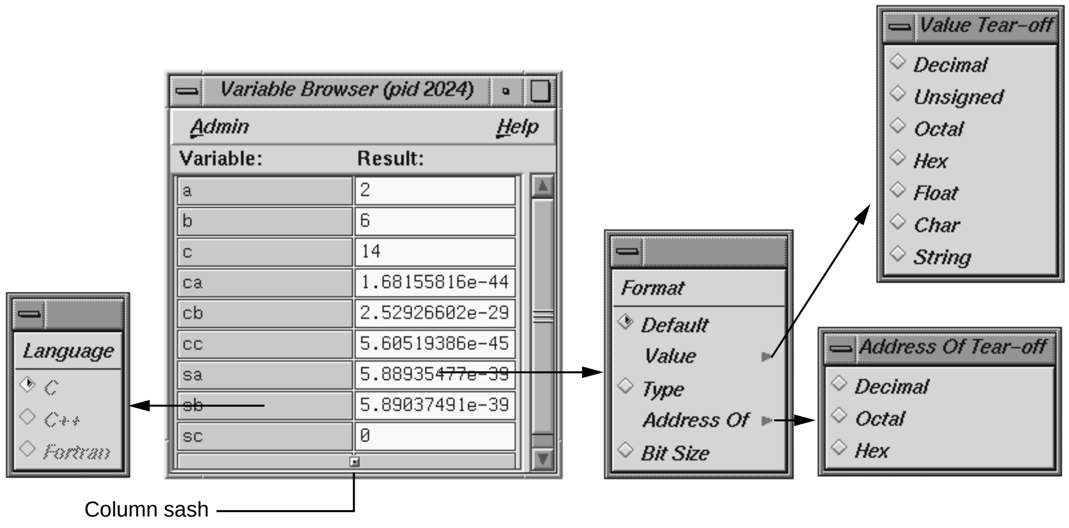
Typically, you inspect variable values at the following points:

- At a breakpoint
- At a frame in a call stack
- As you step through a process

Note: A useful technique is to set a trap at the entry to a function and inspect the values of the variables there. Some variables may be in an uninitialized state at that point. You can then step through the function and make sure that no uninitialized variables are used inadvertently.

A.7.6.1 Entering Variable Values

The Variable Browser lets you change the values of variables in the window. You simply enter the new value in the result column and press Enter. Thus, you can force new values into the process and see their effect.



```
Variable:   Result:
a           2
b           6
c           14
ca          1.68155816e-44
cb          2.52926602e-29
cc          5.60519386e-45
sa          5.88935477e-39
sb          5.89037491e-39
sc          0
```

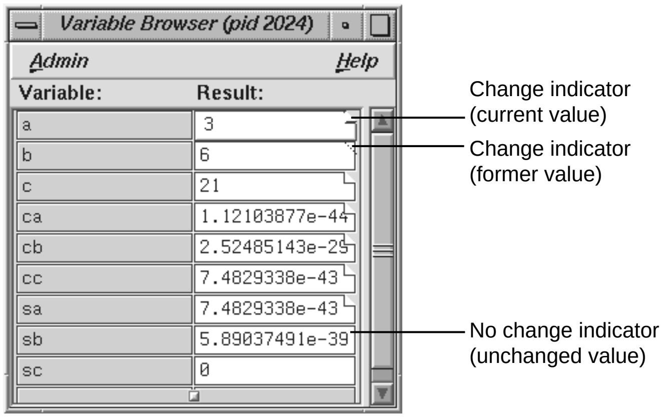
Figure 110. `Variable Browser` with Menus Displayed

A.7.6.2 Changing Variable Column Widths

The `Variable Browser` has a sash between columns that lets you adjust the relative widths of the `Variable` and `Result` columns (see Figure 110, page 220). For example, you may wish to adjust for short variable names and long result values.

A.7.6.3 Viewing Variable Changes

The Debugger views that are involved with variables (that is, the `Variable Browser` and `Expression View`) have indicators that show when the variable has changed since the last breakpoint. If you click the indicator, you can view the previous value. The variable change indicators for a `Variable Browser` window are shown in Figure 111, page 221.


Variable Browser (pid 2024)
Admin
Help
Change indicator (current value)
Change indicator (former value)
No change indicator (unchanged value)

Variable:	Result:
a	3
b	6
c	21
ca	1.12103877e-44
cb	2.52485143e-29
cc	7.4829338e-43
sa	7.4829338e-43
sb	5.89037491e-39
sc	0

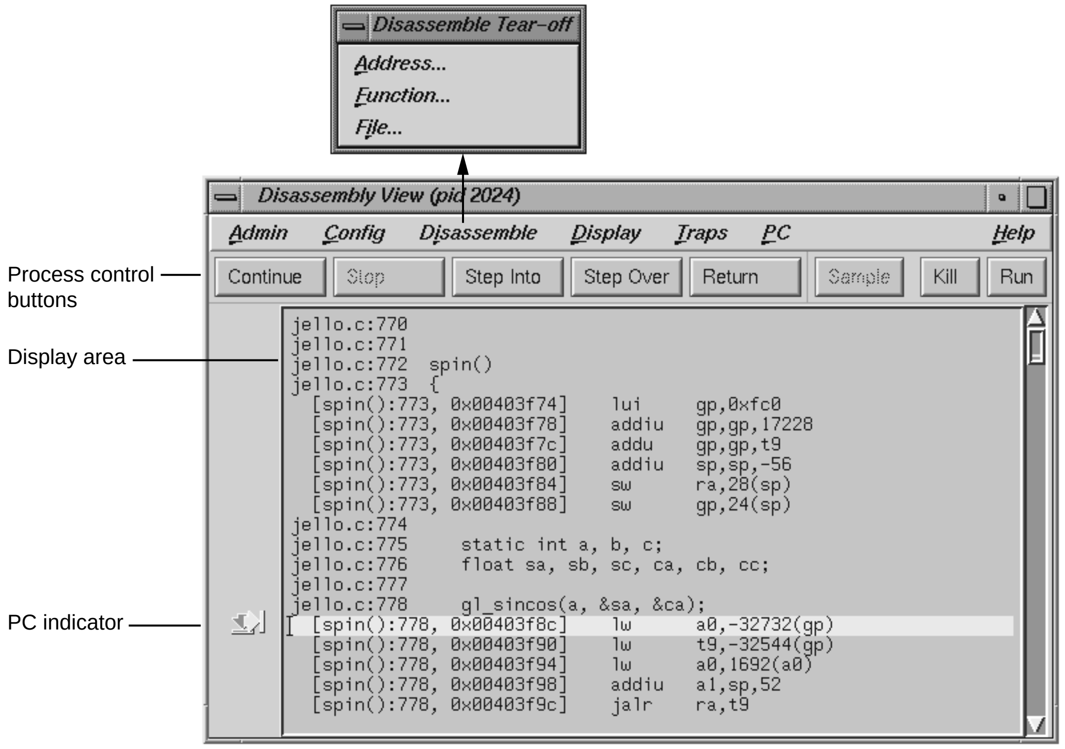
Figure 111. Typical Variable Change Indicators

A.8 Machine-level Debugging Windows

The Debugger offers three views useful in debugging at the machine level: the `Disassembly View`, `Register View`, and `Memory View`.

A.8.1 `Disassembly View`

The `Disassembly View` lets you look at machine-level code rather than source-level code. A typical `Disassembly View` window appears in Figure 112, page 222, with the `Disassemble` menu displayed.



```
jello.c:770
jello.c:771
jello.c:772  spin()
jello.c:773  {
  [spin():773, 0x00403f74]    lui     gp,0xfc0
  [spin():773, 0x00403f78]    addiu   gp,gp,17228
  [spin():773, 0x00403f7c]    addu    gp,gp,t9
  [spin():773, 0x00403f80]    addiu   sp,sp,-56
  [spin():773, 0x00403f84]    sw      ra,28(sp)
  [spin():773, 0x00403f88]    sw      gp,24(sp)
jello.c:774
jello.c:775     static int a, b, c;
jello.c:776     float sa, sb, sc, ca, cb, cc;
jello.c:777
jello.c:778     gl_sincos(a, &sa, &ca);
  [spin():778, 0x00403f8c]    lw      a0,-32732(gp)
  [spin():778, 0x00403f90]    lw      t9,-32544(gp)
  [spin():778, 0x00403f94]    lw      a0,1692(a0)
  [spin():778, 0x00403f98]    addiu   a1,sp,52
  [spin():778, 0x00403f9c]    jalr    ra,t9
```

Figure 112. `Disassembly View` with Menu Displayed

A.8.1.1 Similarities with Main View

At the top of the window are the same process control buttons as those in Debugger Main View. They behave the same way except for `Step Into` and `Step Over`, which do machine-level instruction stepping instead of source-level. Remember that you select the number of steps by holding down the right mouse button over the `Step Into` and `Step Over` buttons.

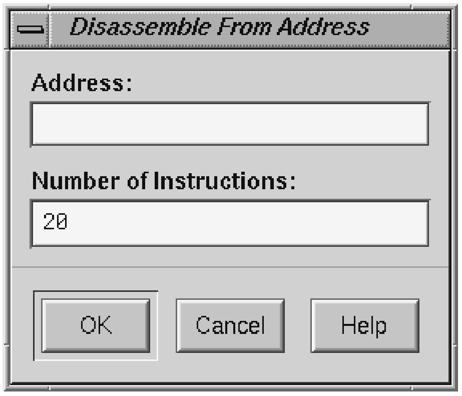
The menus are basically the same as in Main View except for the `Disassemble` menu. The PC menu selections `Continue To` and `Jump To` are based on machine-level instructions rather than source-level steps. The `Config` menu has a `Preferences...` selection that brings up a dialog box oriented to `Disassembly View`.

You can set traps either by using the `Traps` menu or by clicking in the annotation column of the source display area that contains the disassembled code.

A.8.1.2 `Disassemble` Menu

The `Disassemble` menu lets you display disassembled code. It contains the following items:

`Address...`	Allows you to disassemble a specified number of lines, starting from a specified source line address (see Figure 113, page 223).



20

Figure 113. `Disassemble From Address` Dialog

`Function...`	Allows you to disassemble a specified number of lines, starting from the beginning address of a specified function name (see Figure 114, page 224).

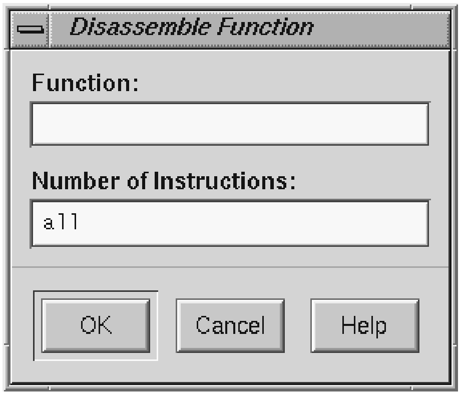
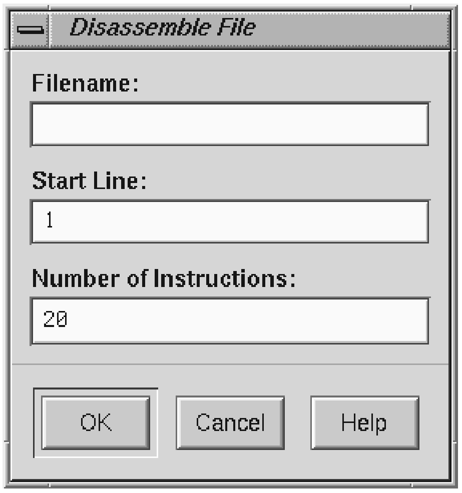


Figure 114. `Disassemble Function` Dialog

`File...`	Allows you to disassemble a specified number of lines, starting from the address corresponding to a specified line number in a specified file (refer to Figure 115, page 225). If you have a current selection in Main View or `Source View`, its file and cursor position are used as the default filename and line number, respectively.



Filename:

Start Line: 1

Number of Instructions: 20

Figure 115. `Disassemble File` Dialog

A.8.1.3 Disassembly View Preferences

Selecting `Preferences...` from the `Config` menu brings up the `Disassembly View Preferences` dialog box (shown in Figure 116, page 226) so that you can change the global preferences.

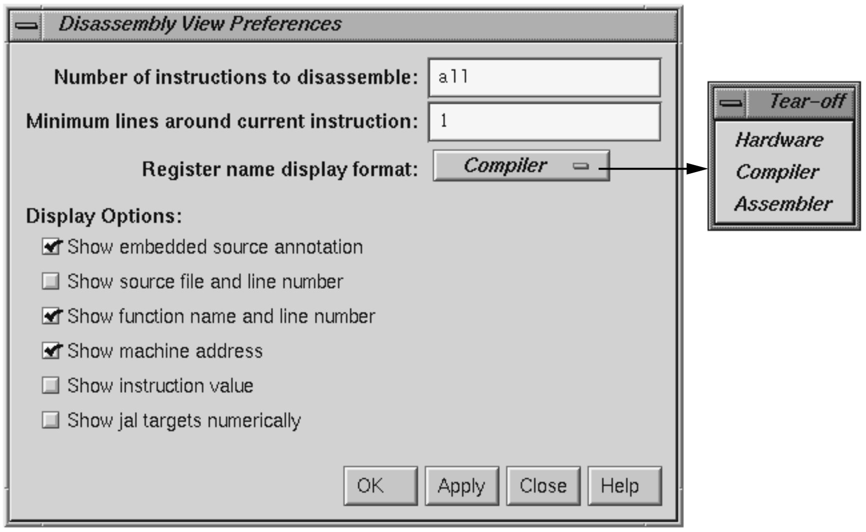

Disassembly View Preferences
Number of instructions to disassemble: all
Minimum lines around current instruction: 1
Register name display format: Compiler
Tear-off
Hardware
Compiler
Assembler
Display Options:
✓ Show embedded source annotation
Show source file and line number
✓ Show function name and line number
✓ Show machine address
Show instruction value
Show jal targets numerically
OK
Apply
Close
Help

Figure 116. `Disassembly View Preferences` Dialog with Pop-up Menu

The dialog box provides you with the following options:

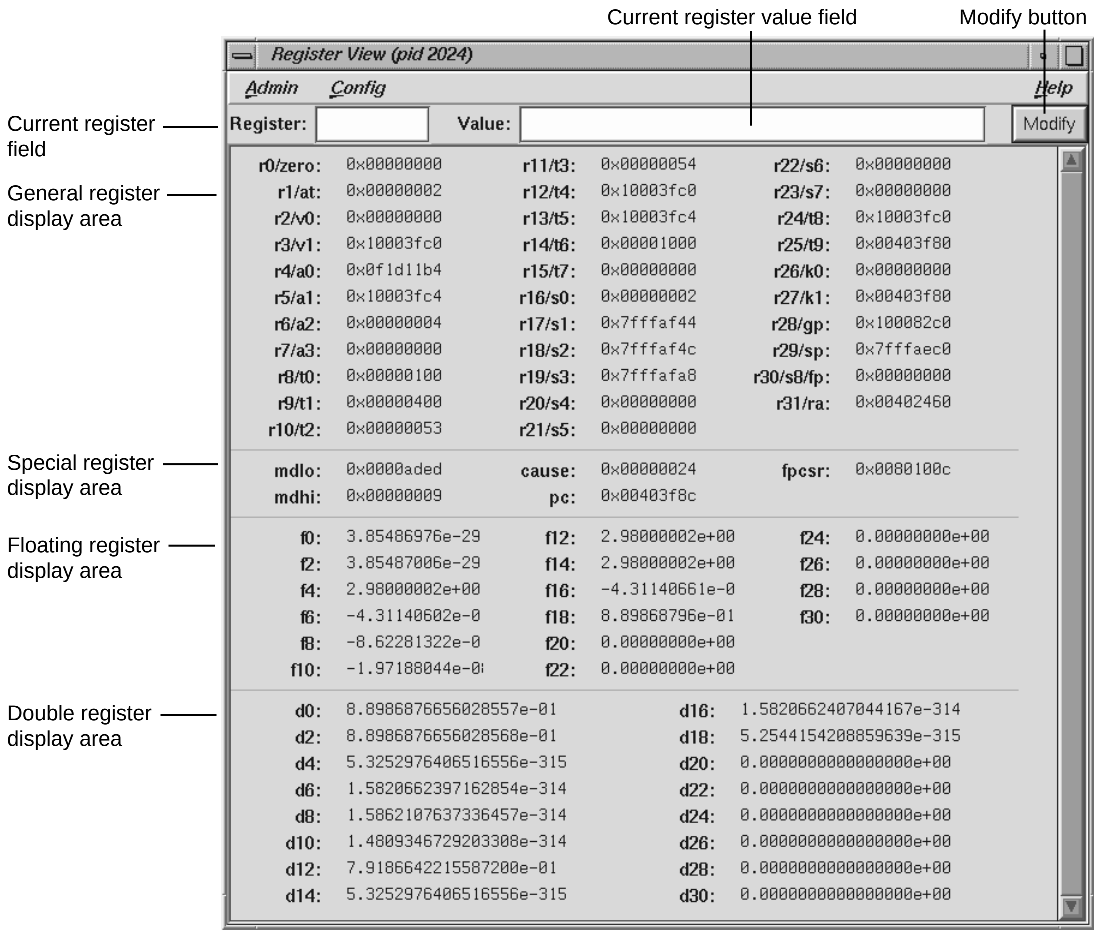
`Number of instructions to disassemble`	Controls the default number of disassembly lines shown when the process stops. This number appears in the dialog boxes selected from the `Disassemble` menu (see Figure 113, page 223, Figure 114, page 224, and Figure 115, page 225). The default is all instructions, indicating that the entire function will be disassembled.
`Minimum lines around current instruction`	Controls the display of the disassembled code, enabling you to view at least the specified number of instructions before and after the current instruction.
`Register name display format`	Controls how register names are displayed. The available modes are `Hardware`, `Compiler`, and `Assembler`.
`Show embedded source annotation`	Turns on interleaved source lines in the appropriate positions.
`Show source file and line number`	Displays the filename and file position along with each machine instruction.

`Show function name and line number`	Displays the function name and file position along with each machine instruction.
`Show machine address`	Displays the memory address of each machine instruction.
`Show instruction value`	Displays the instruction word along with each machine instruction.
`Show jal targets numerically`	Controls whether the target address of a jal instruction is displayed as a hex address or symbolic label.

A.8.2 `Register View` Window

`Register View` window lets you examine and modify register values. You bring it up by selecting `Register View` from the `Views` menu in Main View. Figure 117, page 228, shows a typical `Register View` window that has been resized to show all available registers.

`Register View` displays each register with its current value. A question mark (?) displayed immediately before a register value signifies that the value is suspect; it may not be valid for the current frame. This can occur if a register is not saved across a function call. A colored marker indicates that a register value has changed since the last time the process stopped.


Current register value field
Modify button
Current register field
General register display area
Special register display area
Floating register display area
Double register display area
Register View (pid 2024)
Admin
Config
Help
Register:
Value:
Modify

```
   r0/zero:  0x00000000     r11/t3:  0x00000054      r22/s6:  0x00000000
     r1/at:  0x00000002     r12/t4:  0x10003fc0      r23/s7:  0x00000000
     r2/v0:  0x00000000     r13/t5:  0x10003fc4      r24/t8:  0x10003fc0
     r3/v1:  0x10003fc0     r14/t6:  0x00001000      r25/t9:  0x00403f80
     r4/a0:  0x0f1d11b4     r15/t7:  0x00000000      r26/k0:  0x00000000
     r5/a1:  0x10003fc4     r16/s0:  0x00000002      r27/k1:  0x00403f80
     r6/a2:  0x00000004     r17/s1:  0x7fffaf44      r28/gp:  0x100082c0
     r7/a3:  0x00000000     r18/s2:  0x7fffaf4c      r29/sp:  0x7fffaec0
     r8/t0:  0x00000100     r19/s3:  0x7fffafa8   r30/s8/fp:  0x00000000
     r9/t1:  0x00000400     r20/s4:  0x00000000      r31/ra:  0x00402460
    r10/t2:  0x00000053     r21/s5:  0x00000000

      mdlo:  0x0000aded      cause:  0x00000024       fpcsr:  0x0080100c
      mdhi:  0x00000009         pc:  0x00403f8c

        f0:  3.85486976e-29    f12:  2.98000002e+00     f24:  0.00000000e+00
        f2:  3.85487006e-29    f14:  2.98000002e+00     f26:  0.00000000e+00
        f4:  2.98000002e+00    f16:  -4.31140661e-0     f28:  0.00000000e+00
        f6:  -4.31140602e-0    f18:  8.89868796e-01     f30:  0.00000000e+00
        f8:  -8.62281322e-0    f20:  0.00000000e+00
       f10:  -1.97188044e-0    f22:  0.00000000e+00

        d0:  8.8986876656028557e-01          d16:  1.5820662407044167e-314
        d2:  8.8986876656028568e-01          d18:  5.2544154208859639e-315
        d4:  5.3252976406516556e-315         d20:  0.0000000000000000e+00
        d6:  1.5820662397162854e-314         d22:  0.0000000000000000e+00
        d8:  1.5862107637336457e-314         d24:  0.0000000000000000e+00
       d10:  1.4809346729203308e-314         d26:  0.0000000000000000e+00
       d12:  7.9186642215587200e-01          d28:  0.0000000000000000e+00
       d14:  5.3252976406516556e-315         d30:  0.0000000000000000e+00
```

Figure 117. `Register View`

A.8.2.1 `Register View` Window

The major features of the `Register View` window are the following:

`Current register` field	Identifies the currently selected register. You can switch to a different register by entering its name (either by hardware name or by alias) in this field and pressing `Enter`. You can also switch registers by clicking on the new register in the display area.

`Current register value` field	Shows the contents of the selected register. You can assign a new value to a register by entering either a literal or an expression into the `Value` field. You then click on the `Modify` button to change the value or press `Enter`.
`Register display area`	Shows the registers organized into four groups: general, special, floating, and double. Note that the general registers are identified by both their hardware and software names. Double registers have a one-to-two correspondence with the floating registers.

Note: The special registers `p0`, `p1`, and `p2` are empty in the figure. These are used for instrumentation and display values only when instrumentation has taken place.

A.8.2.2 Changing the `Register View` Display

The `Preferences...` selection in the `Config` menu lets you change the `Register View` display. It brings up the `Register View Preferences` dialog box (see Figure 118, page 230).

The register display toggle buttons let you specify which types of registers are to be displayed by default.

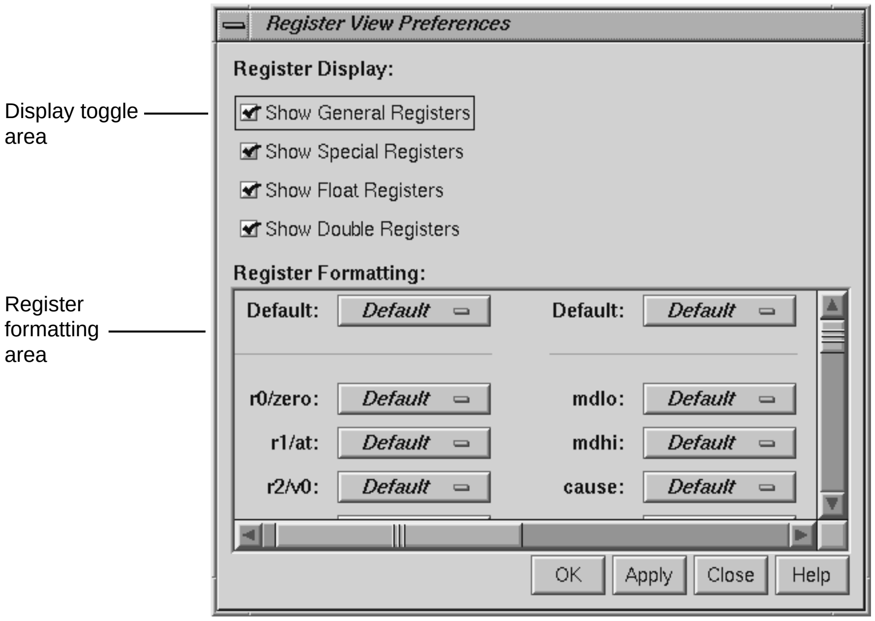


Figure 118. `Register View Preferences` Dialog

The register formatting area lets you select formats for any of the registers.

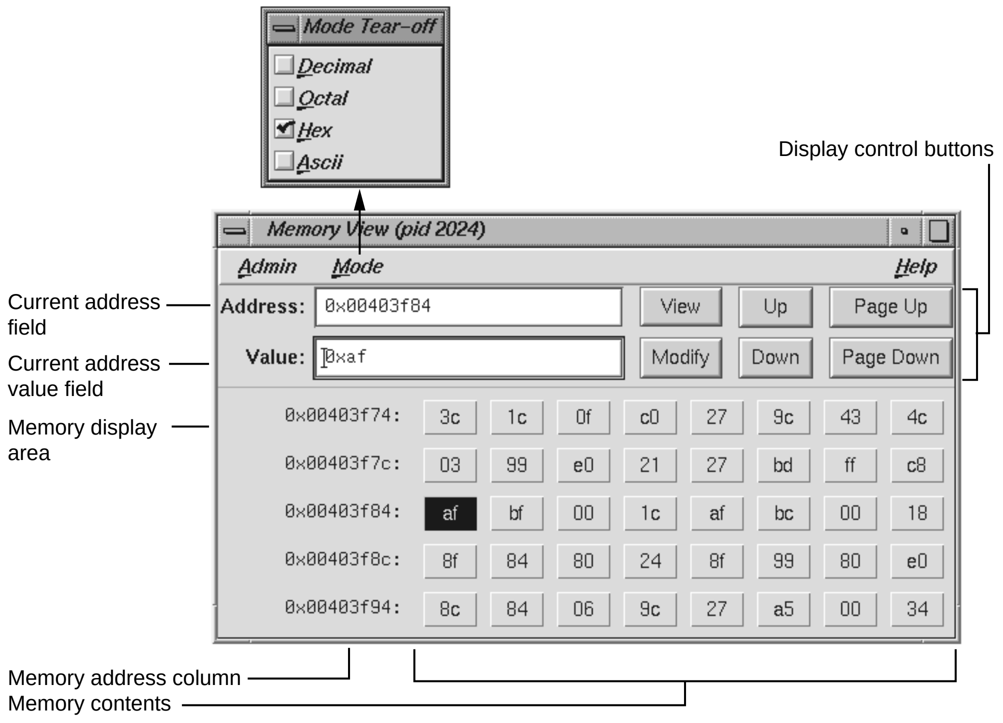
The default fields in the top row let you change defaults for the four major types, which are set as follows:

- General registers — hexadecimal
- Special registers — hexadecimal
- Float registers — floating point
- Double registers — floating point

The rows in the register formatting area let you change the modes for the individual registers.

A.8.3 Memory View

The `Memory View` window lets you examine and modify memory. A typical `Memory View` window appears in Figure 119, page 231.



```
Address: 0x00403f84
Value:   0xaf

0x00403f74:  3c  1c  0f  c0  27  9c  43  4c
0x00403f7c:  03  99  e0  21  27  bd  ff  c8
0x00403f84:  af  bf  00  1c  af  bc  00  18
0x00403f8c:  8f  84  80  24  8f  99  80  e0
0x00403f94:  8c  84  06  9c  27  a5  00  34
```

Figure 119. `Memory View` with Menu Displayed

A.8.3.1 Viewing a Portion of Memory

To view a portion of memory, enter the beginning memory location in the `Address` field. You can enter the literal value or an expression that evaluates to an integer address. These address specifications must be in the language of the current process as indicated by the call stack frame. For example, you can enter **`0x7fff4000+4`** as the memory address when stopped in a C function or enter **`$7fff4000+4`** as the equivalent for a Fortran routine. Press `Enter` while the cursor is in the field or click the `View` button to display the contents of that location and the subsequent locations in the display area. This also displays the contents of the first address in the `Value` field where it can be modified.

The memory display area shows the contents of individual byte addresses. The column at the left of the display shows the first address in the row. The contents at that address are shown immediately to its right, followed by the contents of the next seven byte locations. If you enlarge the `Memory View` window, you can see additional rows of memory.

A.8.3.2 Changing the Contents of a Memory Location

To change the contents of a memory location, you select the address to be changed, either by direct entry or by clicking on the byte value in the display area. You can enter a single value or a sequence of hex byte values separated by spaces (for example, **`00 3a 07 b2`**) in the `Value` field. You can also enter a quoted string to change a consecutive range of values to the ASCII values of that string. Pressing `Enter` while the cursor is in the `Value`field or clicking the `Modify` button substitutes the new value(s) starting at the specified location.

A.8.3.3 Changing the Memory Display Format

The `Mode` menu lets you change the format of the value field or byte locations to either decimal, octal, hex, or ASCII.

A.8.3.4 Moving around the `Memory View` Display Area

The four control buttons at the upper right of the window help you move around the display area. These buttons are:

`Up`	Moves displayed bytes up a single row.
`Down`	Moves displayed bytes down a single row.
`Page Up`	Moves displayed bytes upward by as many rows as are currently displayed.
`Page Down`	Moves displayed bytes downward by as many rows as are currently displayed.

A.9 Fix+Continue Windows

The Fix+Continue utility interacts with several WorkShop windows. The Debugger and `Source View` access the Fix+Continue utility from the menu bar. The results of running redefined code are displayed in the Debugger `Execution View`. Special line numbers (in decimal notation) applied to redefined functions appear in several WorkShop views (refer to Section A.9.4, page 241).

Note: Fix+Continue functionality within the debugger is limited to programs compiled with the `-o32` compiler option.

Fix+Continue comes with the following windows devoted entirely to Fix+Continue functions:

- `Status` window
- `Message` window
- `Build Environment` window

This section describes Fix+Continue menu selections and windows.

The `Fix+Continue` menu is available from the Debugger Main View menu bar. The menu selections operate on the selected function or on the file shown in `Source View`. The `Fix+Continue` menu is also available from `Source View` and from the `Fix+Continue Status` window.

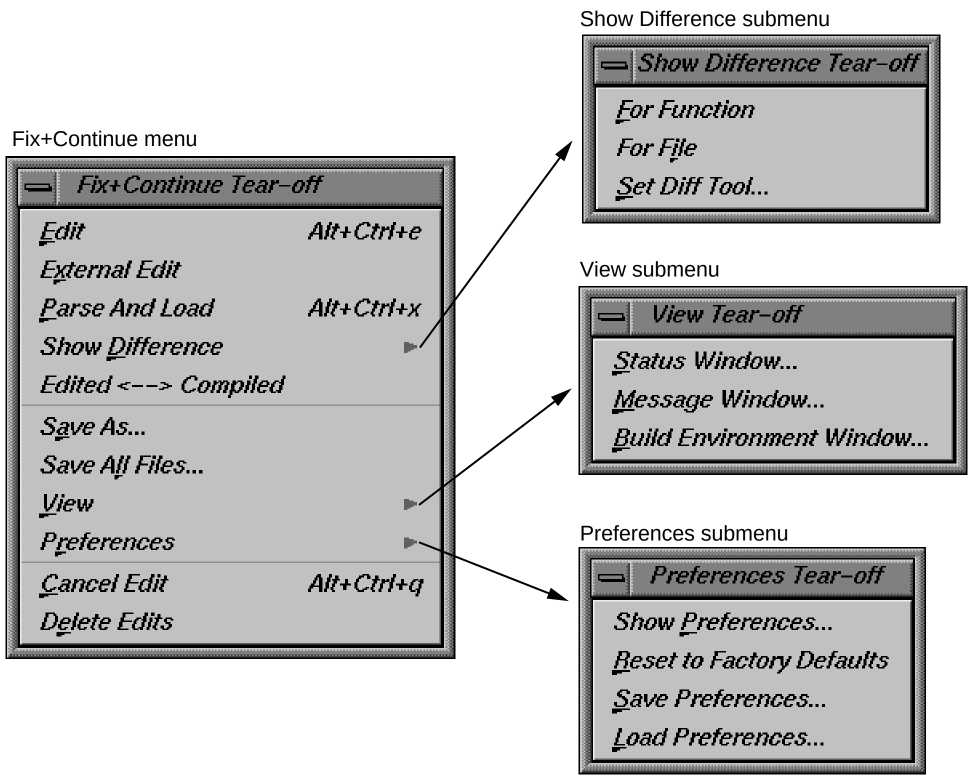
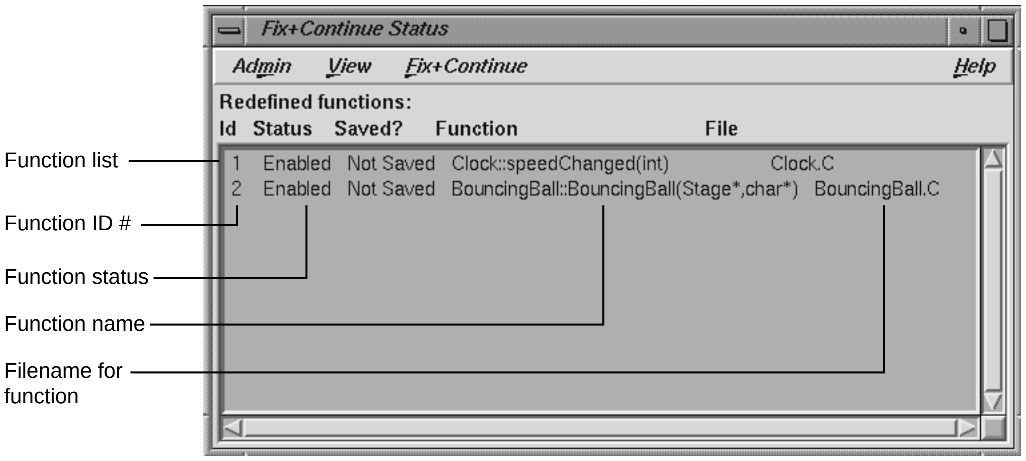


Figure 120. Fix+Continue Menu Selections

A.9.1 Fix+Continue Status Window

The Fix+Continue Status window (see Figure 121, page 234) provides you with a summary of the modifications you have made during your session. It also allows you quick access to your modified functions and somewhat expanded menu options.



Redefined functions:

```
Id  Status   Saved?     Function                                  File
1   Enabled  Not Saved  Clock::speedChanged(int)                  Clock.C
2   Enabled  Not Saved  BouncingBall::BouncingBall(Stage*,char*)  BouncingBall.C
```

Figure 121. Fix+Continue Status Window

The function ID number, status, name, and filename are displayed in the window. Double-clicking a line in the window brings up the corresponding source in the Debugger main window.

The menus and submenus that provide you with extra functionality through the Status window (see Figure 122, page 235) are described below.

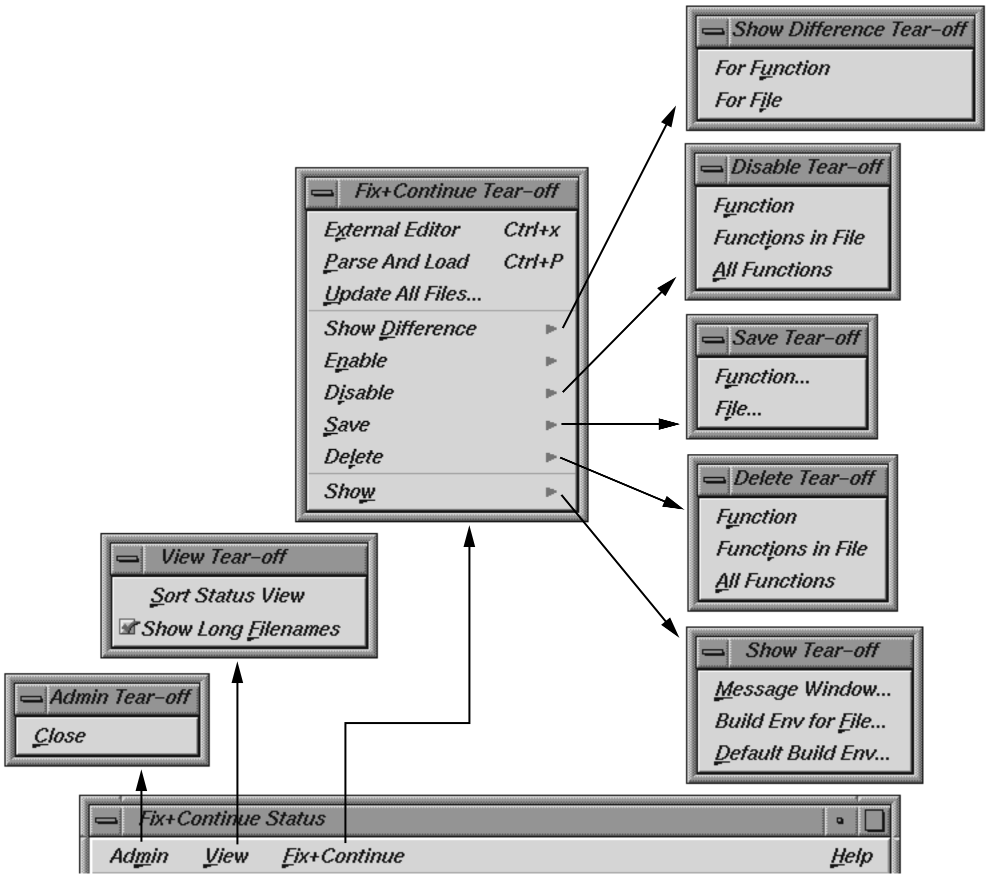


Figure 122. `Fix+Continue Status Window` Menus

A.9.1.1 `Admin` Menu

The `Admin` menu contains an option for closing the window.

`Close`	Closes the status window.

A.9.1.2 `View` Menu

The `View` menu contains options for sorting information in the window and displaying file names.

`Sort Status View`	Sorts the information in the status view according to the field currently selected.

`Show Long Filenames`	Toggles among absolute (long) pathnames, relative pathnames, or base names.

A.9.1.3 `Fix+Continue` Menu

The `Fix+Continue` menu available from the `Fix+Continue Status` window is somewhat different from that available through the Debugger Main View. It contains a number of options and submenus that are described below. These options and submenus are active on the function that you select in the `Source View`. You can select a function by clicking on it. The following options and submenus are available:

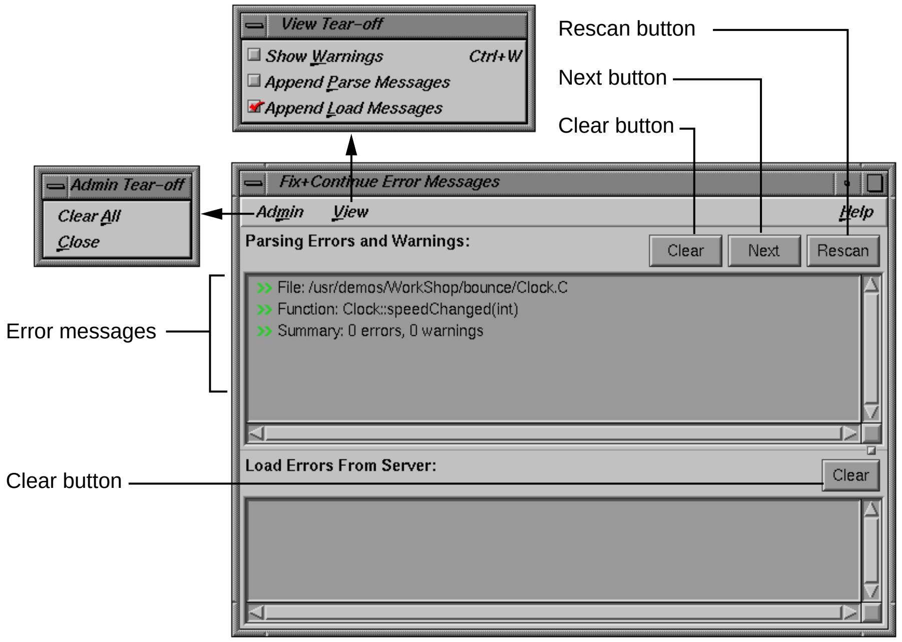
`External Editor`	Allows you to edit with an external editor such as `vi`, rather than the Debugger's default editor.
`Parse And Load`	Parses your modified function and loads it for execution. You can execute the modified function by clicking on the `Run` or `Continue` buttons in the Debugger main view.
`Update All Files...`	Launches the `Save File+Fixes As...` dialog that allows you to update the current session while saving all the modified functions to the appropriate files.
`Show Difference` submenu	Allows you to show the difference between the original source and your modified code. You can show the difference in the code in one of the two following ways: • `For Function` — Shows the differences for the current function only. • `For File` — Shows the differences for the entire file that contains the current function.
`Enable` submenu	Allows you to enable the changes in your modified code in one of the following ways: • `Function` — Enables the changes in the current function. • `Functions in File` — Enables the changes to the current function in its own file. • `All Functions` — Enables the changes to all functions in the modified code.

Disable submenu	Has the same menu choices as the `Enable` submenu, but disables rather than enables.
`Save` submenu	Allows you to save your code changes to a file. You can save the changes in one of the following ways: • `Function...` — Launches the File dialog, allowing you to save only the current function to a file. • `File...` — Launches the `Save File+Fixes As...` pop-up window allowing you to save the entire file that contains the current function.
`Delete` submenu	Has the same menu choices as the `Save` submenu, but deletes rather than saves.
`Show` submenu	Allows you to launch any of the following windows: • `Message Window` — Launches a `Fix+Continue Error Messages` window for the selected function. See Section A.9.2, page 238, for more details. • `Build Env for File` — Launches a `Fix+Continue Build Environment` window for the file shown in the `Source View`. See Section A.9.3, page 239, for more details on the `Fix+Continue Build Environment` window. • `Default Build Env` — Launches the `Fix+Continue Build Environment` window to show the options that are to be used in cases where they could not be obtained from the target. See Section A.9.3,

page 239, for details on the `Fix+Continue Build Environment` window.

A.9.2 `Fix+Continue Error Messages` Window

The `Fix+Continue Error Messages` window (see Figure 123, page 238) contains a list of errors and other system messages that pertain to your source modifications, parses, and attempts to run your modified source.



```
>> File: /usr/demos/WorkShop/bounce/Clock.C
>> Function: Clock::speedChanged(int)
>> Summary: 0 errors, 0 warnings
```

Figure 123. `Fix+Continue Error Messages` Window

You can highlight the source line where the error occurred by double-clicking the appropriate line in the window. The `Fix+Continue Error Messages` window contains the following buttons:

`Clear`	Clears all the parsing errors and warnings.
`Next`	Puts a tick mark on the next unticked error warning entry in the parse messages. It displays the corresponding file and line in the Source view, highlighting it according to the type of

	error or warning. `Next` does not function after all the entries in the messages are ticked.
`Rescan`	Erases all the ticks, so that you can rescan all the error warnings from the beginning.

The added functionality available through the window's `Admin` and `View` menus is described below.

A.9.2.1 `Admin` Menu

The `Admin` menu allows you to perform either of the following two operations:

`Clear All`	Clears all messages in the window.
`Close`	Closes the window.

A.9.2.2 View Menu

The `View` menu allows you to set any of the following toggles:

`Show Warnings`	Causes compile warnings to be displayed in the parse errors list.
`Append Parse Messages`	Causes parse messages to be appended to the parse errors list.
`Append Load Messages`	Causes load messages to be appended to the load errors list.

A.9.3 `Fix+Continue Build Environment` Window

This section describes the `Fix+Continue Build Environment` window (see Figure 124, page 240). The `Fix+Continue Build Environment` window provides you with the build information for your source code in your current environment. It displays the command that was used to build your executable and the name of the file that contains the function that you currently have selected.

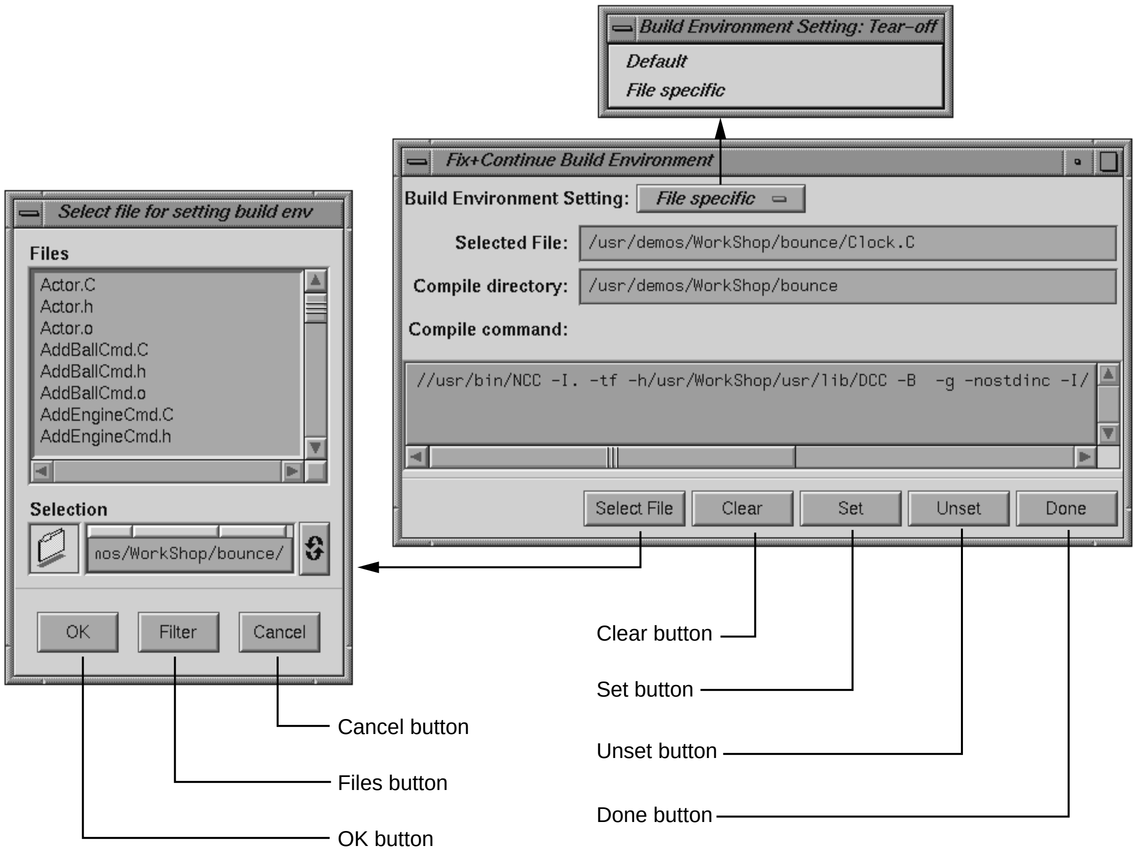

Build Environment Setting: Tear-off
Default
File specific
Fix+Continue Build Environment
Build Environment Setting: File specific
Selected File: /usr/demos/WorkShop/bounce/Clock.C
Compile directory: /usr/demos/WorkShop/bounce
Compile command:
//usr/bin/NCC -I. -tf -h/usr/WorkShop/usr/lib/DCC -B -g -nostdinc -I/
Select File
Clear
Set
Unset
Done
Select file for setting build env
Files
Actor.C
Actor.h
Actor.o
AddBallCmd.C
AddBallCmd.h
AddBallCmd.o
AddEngineCmd.C
AddEngineCmd.h
Selection
nos/WorkShop/bounce/
OK
Filter
Cancel
Cancel button
Files button
OK button
Clear button
Set button
Unset button
Done button

Figure 124. `Fix+Continue Build Environment` Window

The compiler and associated flags that were used to compile the file are normally gathered from the target. You can use this window to make any changes to these flags.

The window allows you to select your build environment setting through the `Build Environment Setting` toggle that contains the following two options:

`Default`	Sets the build environment to default that is displayed in the window.
`File Specific`	Sets the build environment to that of the file that contains the currently selected function. You can

change the file by clicking the `Select File` button, which launches the `File` dialog.

The `Fix+Continue Build Environment` window also contains the following buttons:

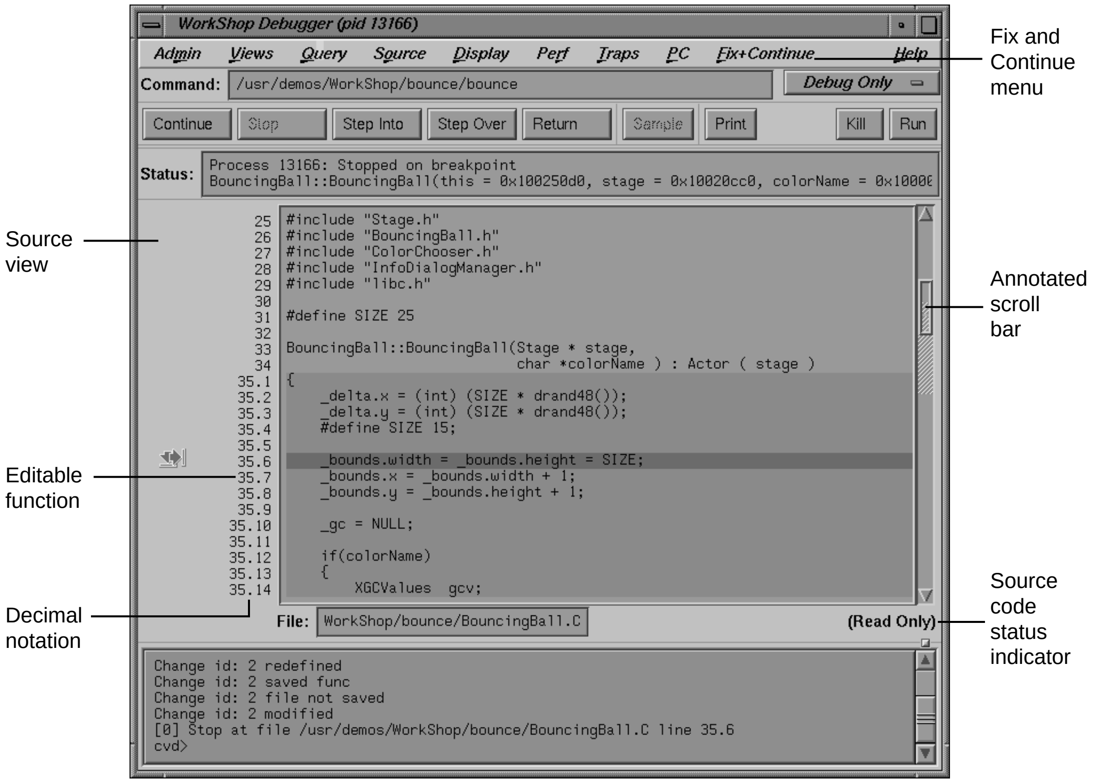
`Select File`	Launches the `File` dialog and allows you to select a file from which to set the build environment.
`Clear`	Clears the window.
`Set`	Sets the build environment to what is displayed in the window.
`Unset`	Unsets the build environment.
`Done`	Dismisses the window.

A.9.4 Changes to Debugger Views

When you use Fix+Continue, Debugger views change to show redefined functions or stopped lines containing redefined functions.

A.9.4.1 Main View

All Fix+Continue functions are available through the `Fix+Continue` menu on the Debugger Main View. See Figure 125, page 242, for details.



```
Command: /usr/demos/WorkShop/bounce/bounce

Status: Process 13166: Stopped on breakpoint
        BouncingBall::BouncingBall(this = 0x100250d0, stage = 0x10020cc0, colorName = 0x1000
```

```
  25  #include "Stage.h"
  26  #include "BouncingBall.h"
  27  #include "ColorChooser.h"
  28  #include "InfoDialogManager.h"
  29  #include "libc.h"
  30
  31  #define SIZE 25
  32
  33  BouncingBall::BouncingBall(Stage * stage,
  34                             char *colorName ) : Actor ( stage )
35.1  {
35.2      _delta.x = (int) (SIZE * drand48());
35.3      _delta.y = (int) (SIZE * drand48());
35.4      #define SIZE 15;
35.5
35.6      _bounds.width = _bounds.height = SIZE;
35.7      _bounds.x = _bounds.width + 1;
35.8      _bounds.y = _bounds.height + 1;
35.9
35.10     _gc = NULL;
35.11
35.12     if(colorName)
35.13     {
35.14         XGCValues  gcv;
```

```
File: WorkShop/bounce/BouncingBall.C                    (Read Only)
```

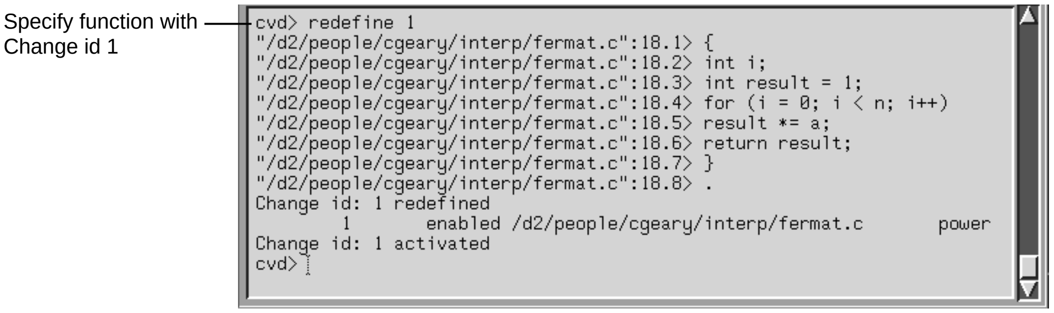
```
Change id: 2 redefined
Change id: 2 saved func
Change id: 2 file not saved
Change id: 2 modified
[0] Stop at file /usr/demos/WorkShop/bounce/BouncingBall.C line 35.6
cvd>
```

Figure 125. Debugger Main View

You can select commands from the `Fix+Continue` menu or enter them at the Debugger command line. The source code status is `Read Only`. Color coding shows the differences between editable code, enabled redefinitions, disabled definitions, and breakpoints. Line numbers in redefined functions have decimal notation that is used for every reference to the line number. The integer portion of the decimal is the same as the first line of the function. This ensures that compiled source code line numbers remain unchanged.

A.9.4.2 Command Line Interface

The Debugger command line interface accepts Fix+Continue commands and reports status involving redefined functions or files. Figure 126, page 243, shows a function successfully redefined using the command line. Change id 1 was previously redefined and assigned the number 1.

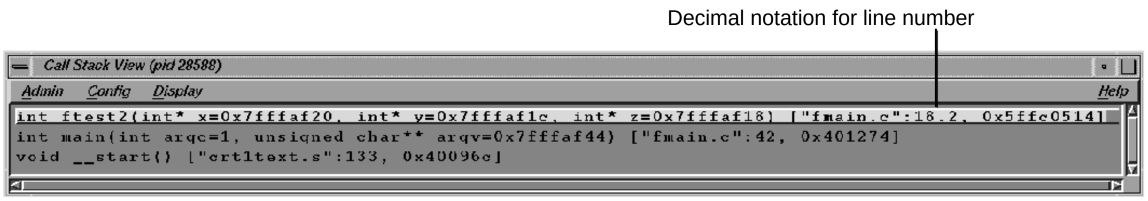


```
cvd> redefine 1
"/d2/people/cgeary/interp/fermat.c":18.1> {
"/d2/people/cgeary/interp/fermat.c":18.2> int i;
"/d2/people/cgeary/interp/fermat.c":18.3> int result = 1;
"/d2/people/cgeary/interp/fermat.c":18.4> for (i = 0; i < n; i++)
"/d2/people/cgeary/interp/fermat.c":18.5> result *= a;
"/d2/people/cgeary/interp/fermat.c":18.6> return result;
"/d2/people/cgeary/interp/fermat.c":18.7> }
"/d2/people/cgeary/interp/fermat.c":18.8> .
Change id: 1 redefined
        1       enabled /d2/people/cgeary/interp/fermat.c       power
Change id: 1 activated
cvd>
```

Figure 126. Command Line Interface with Redefined Function

A.9.4.3 `Call Stack View`

The `Call Stack View` recognizes redefined functions. It uses the decimal notation for line numbers, as shown in Figure 127, page 243.

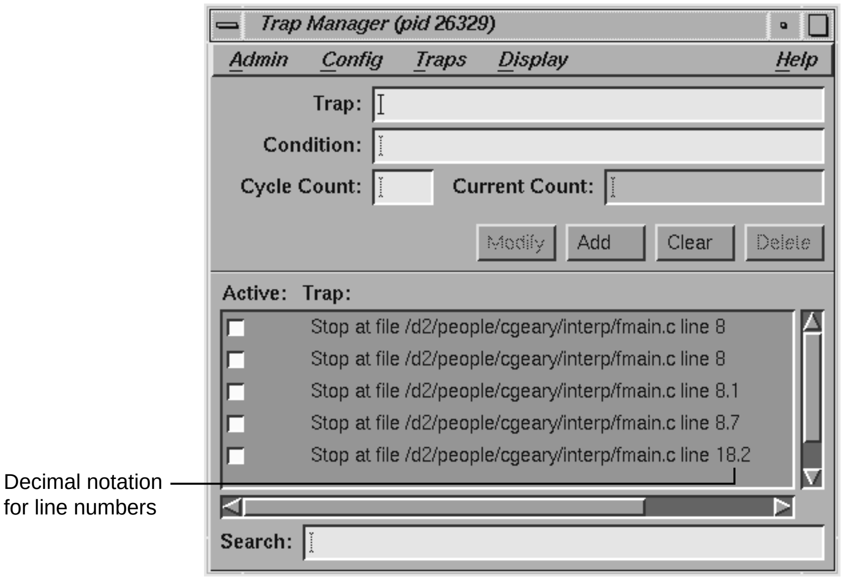


```
int ftest2(int* x=0x7fffaf20, int* y=0x7fffaf1c, int* z=0x7fffaf18) ["fmain.c":18.2, 0x5ffc0514]
int main(int argc=1, unsigned char** argv=0x7fffaf44) ["fmain.c":42, 0x401274]
void __start() ["crt1text.s":133, 0x40096c]
```

Figure 127. `Call Stack View`

A.9.4.4 Trap Manager

The Trap Manager recognizes redefined functions. It uses the decimal notation for line numbers, as shown in Figure 128, page 244.



Stop at file /d2/people/cgeary/interp/fmain.c line 8
Stop at file /d2/people/cgeary/interp/fmain.c line 8
Stop at file /d2/people/cgeary/interp/fmain.c line 8.1
Stop at file /d2/people/cgeary/interp/fmain.c line 8.7
Stop at file /d2/people/cgeary/interp/fmain.c line 18.2

Figure 128. `Trap Manager` Window with Redefined Function

A.10 Debugger Command Line

To use the Debugger commands, which are entered at the command line at the bottom of Main View (see Figure 59, page 118), you should be familiar with `dbx` commands. For more information, refer to the *dbx Reference Manual*. The syntax for the debugging commands is as follows:

`add_source {"`*filename*":*line_number* }

Prompts you to add source code lines (for example, `add_source "fmain.c":15.2`). *line_number* must be within the body of a function. Entering a period (.) specifies the end of your input. The source lines you provide are added after the specified line. This command returns an ID existing or new, depending on whether the function affected has already been changed or not. The resulting new definition of the function is executed on its entry next time. See also `delete_source` and `replace_source`. Applies to –o32 code only.

`alias` [*shortform command*]

Lists all aliases without arguments. With arguments, it assigns *command* to *shortform*.

`assign` *expression1=expression2*

Assigns *expression2* to *expression1*.

`attach` *pid*

Attaches to specified process ID (*pid*).

`call function_name [` *argument*`, ...]`

Executes the specified function with any arguments supplied.

`catch [` *signal_name* `| all]`

With no arguments, lists signals to be trapped. If a signal is specified, it's added to the list. If `all` is specified, it traps all signals.

`clear [all |` *source_line*`]`

Clears breakpoints. The `all` option clears all breakpoints. The *source_line* option clears the breakpoint at the specified source line.

`clearbuffer`

Clears the currently displayed lines.

`clearcalls`

Cancels pending function calls.

`cont in` *function_name*

Continues execution from the current line to the entry to the specified function.

`cont to` *line_number*

Continues execution from the current line until the specified line.

`continue` [all]

Continues executing a program, or all programs, after a breakpoint. You can use both `c` and `cont` as aliases for `continue`.

`continue` [*signal*]

Sends specified signal and continues executing a program after a breakpoint.

`corefile` [*filename*]

With no arguments, reports whether data referencing commands reference a core file. If so, displays the current core file. With *filename* provided, specifies core file to be debugged.

`delete all`

Deletes all traps.

delete [,*displaynumber*,. . .]

Deletes the specified expression from the display.

`delete_changes` {*func_spec* `| -all | {-file` *filename*}}

Deletes changes corresponding to the selected functions (for example, `delete_changes getNumbers -file fmain.c`). Once IDs are deleted, you will not be able to use the IDs again because the IDs associated with the selected functions are released. The default is `-all`. See also `save_changes`. Applies to –o32 code only.

`delete_source` {"*filename*":*line_number*[,line_number]}

Deletes the given line(s) if *line_number* or *,line_number* (range) is within the body of a function. An example is:

```
delete_source "fmain.c":8.6,8.7
```

This command returns an ID existing or new, depending on whether the function affected has already been changed or not.

The resulting new definition of the function is executed on its entry next time. Applies to –o32 code only.

`delete` *trap_number* `[,`*trap_number*`, ...]`

Deletes the specified breakpoint from the status list.

`detach`

Detaches from the current process.

`disable all`

Deactivates all traps.

`disable_changes {`*func_spec* `| -all | {-file` *filename*`}}`

Disables specified changes for selected functions (for example, `disable_changes getNumbers -file fmain.c`. Nothing happens if the selected function is already disabled. The compiled definition of the function is executed on its next entry. You can invoke this command when the process is stopped or on a running process when a function entry breakpoint is set.

`disable` *trap_number* `[,`*trap_number*`, ...]`

Deactivates a trap set by `stop` command.

`display [`*expression*`, ...]`

With *expression*, adds *expression* to the list of expressions displayed whenever the process stops. With no arguments, lists all expressions.

`down` [*expression*]

Moves down the specified number of frames in the call stack.

`dump`

Prints local variable values.

`enable all`

Reactivates all inactive traps.

`enable_changes` {*func_spec* `| -all | {-file` *filename*}}

Enables specified changes for selected functions (for example, `enable_changes getNumbers -file fmain.c`. Nothing happens if the selected function is already enabled. The latest accepted definition of the function is redefined on its next entry. You can invoke this command when the process is stopped or on a running process when a function entry breakpoint is set. Applies to –o32 code only.

`enable` *trap_number* [,*trap_number*, ...]

Reactivates a disabled breakpoint.

expression / [*count*] [*format*] or *expression*,[*count*] / [*format*]

Prints the contents of the memory address specified by *expression*, according to the specified format. *count* represents the number of formatted items. The following format options are available:

`d`	Prints a short word in decimal.
`D`	Prints a long word in decimal.
`o`	Prints a short word in octal.
`O`	Prints a long word in octal.
`x`	Prints a short word in hexadecimal.
`X`	Prints a long word in hexadecimal.
`b`	Prints a byte in octal.
`c`	Prints a byte as a character.
`s`	Prints a string of characters that ends in a null byte.
`f`	Prints a single-precision real number.

`g`	Prints a double-precision real number.

`file` [*filename*]

Displays the name of the current or specified file (*filename*). If a file is specified, it becomes the current file.

`func` [*func_name*]

Moves to the source code corresponding to the specified frame in the call stack or to the function in the executable if not on the stack.

`givenfile` [*filename*]

With no arguments, displays name of current object file. With *filename*, specifies object file to be debugged.

`goto` *linenumber*

Skips over lines going directly to the specified line number. Unlike `dbx`(1), `cvd`(1) does not begin execution at the specified line.

`ignore [`*signal_name* `| all]`

With no arguments, lists those signals not to be trapped. If a signal is specified, this command removes it from the list of signals to be trapped. If `all` is specified, ignores all signals.

`kill` [*pid*][all]

Kills the specified process currently controlled by the Debugger or kills all processes.

`list [[`*expression1* `[,`*expression2*`]] | [`*function_name*`]]`

Lists source lines in *expression1* for *expression2* number of lines. The default is 10 lines. Optionally, you can specify the name of a function from which you want lines listed.

`list_changes [`*func_spec* `| -all | {-file` *filename*}]

Lists one or more lines using the following syntax:

```
change_id isEnabled filename function_spec
```

For example:

```
4 enabled foo.c foo
8 disabled A.c++ A::bingo
```

The default is `list_changes -all`.

`next` [*int*]

Steps over the specified number of source instructions. This command does not step into procedures. The default is one instruction.

`nexti` [*int*]

Steps over the specified number of machine instructions. This command does not step into procedures. The default is one line.

`print` *expression*[,*expression*, ...]

Prints the value of the specified expression(s). If the expression is a character pointer or array, both the string and address print.

`printd` *expression* [,*expression*, ...]

Prints the value of the specified expression(s) in decimal format. You can use `pd` as an alias.

`printo` *expression* [,*expression*, ...]

Prints the value of the specified expression(s) in octal format. You can use `po` as an alias.

`printregs`

Prints the contents of the registers.

`printx` *expression*[,*expression*, ...]

Prints the value of the specified expression(s) in hexadecimal format. You can use `px as an alias`.

`pwd`

Displays the current directory.

`quit`

Exits the debugging session.

`redefine` *func_spec*

`[-edit |{ -read` *filename*`[`*line_number*`,`*line_number*`]}]`
Specifies a new body for a function. The new definition is checked, and errors (if any) are printed. The new function body is redefined on the next function entry. Breakpoints (if set) on the old definition are put on the new definition based on their relative line number position from the beginning of the function definition. (Note that some breakpoints may not make it to the new definition.) You can invoke this command when the process is stopped or on a running process when a function entry breakpoint is set. There are three ways to provide a new definition:

- `-edit` pops up an editor of your choice containing the current definition of the function. The specification of the new definition is complete when you exit the editor. You may not leave the editor open.
- `-read` takes the contents of the file specified (within the line numbers if given) as the new function definition.
- No option allows you to type in replacement code from the next line. A period in the first column on a fresh line terminates the definition. For example:

  ```
  redefine getNums
  "/usr/fmain.c'':8.1> {
  "/usr/fmain.c'':8.2> printf(''In getNums.\n'');
  "/usr/fmain.c'':8.3> }
  "/usr/fmain.c'':8.4> .
  ```

  You can use a combination of characters (yet to be determined) to open an editor of your choice containing the lines typed. The specification of the new definition is complete when you exit the editor. Applies to –o32 code only.

`replace_source {"`*filename*`":`*line_number*`[,`*line_number*]}

Prompts you to type in replacement source if *line_number* or *,line_number* (range) is within the body of a function. The source lines you provide replace the specified line(s). An example is: `replace_source "fmain.c":12`. This command returns an existing or new id depending on whether the function affected has already been changed or not. The resulting new definition of the function is executed on its entry next time. See also `add_source` and `delete_source`. Applies to –32 code only.

`rerun`

Runs the program again using the same arguments.

`return`

Continues executing the current procedure and returns to the next sequential line in the calling function.

`run` [all]

Runs the program (s).

`runtime_check` *func_spec* `[-options` *key* `[`*key*`,...]]`

Enables all run-time checking options by default. If `-options` is specified, then run-time checking is restricted to the *keys*. The result of theruntime checks are printed the next time the specified function (*func_spec*) is entered. You can invoke this command on a stopped or a running process.

`save_changes {`*func_spec*`| {-file` *filename*`}}` [-[w | a]] *filename_to_save*

Saves (enabled or disabled) function redefinitions or an entire file to another file (*filename_to_save*). The following example shows how to save a function definition:

```
save_changes getNumbers getNumbersFunc
```

If you specify the `-file` option, then before saving to *filename_to_save*, all function changes are applied to the compiled source of the file (with the condition that the file has had only its functions redefined, and has not been edited since the last build). An example of saving an entire file is the following:

```
save_changes -file fmain.c fmain.c
```

The `-w` option replaces the *filename_to_save*. The `-a` option appends to the *file_to_save*. An example of adding a function to a file is the following:

See also `delete_changes`. Applies to –o32 code only.

`setbuildenv ["`*filename*`"] compiler-flag-list`

Overrides default build environment flags (compiler options). Without *filename*, the flags are passed along with `-c -g` flags to the compiler for any function in any file except those set separately with `setbuildenv`. An example is the following:

```
setbuildenv -DnameA -Idir
```

If *filename* is given, this command sets separate flags specifically for that file. For example, consider the following:

```
setbuildenv "fermat.c" -DnameB -Ianotherdir
```

Applies to –o32 code only. See also `unsetbuildenv`.

`sh [`*shell_command*`]`

Call a shell if no arguments; otherwise, executes the specified shell command.

`showbuildenv ["`*filename*`"]`

Lists all the build environment flags set. `showbuildenv` with a *filename* lists any build environment specifications that have been set separately with `setbuildenv "`*filename*`"`. Applies to –o32 code only.

`show_changes [`*func_spec* `| -all | {-file` *filename*`}]`

Prints the code of all enabled redefinitions of the specified function(s). The default is `show_changes -all`. See also `enable_changes` and `disable_changes`. Applies to –o32 code only.

`show_diff {`*func_spec* `| {-file` *filename*`}}`

Launches a `xdiff` comparing the compiled source and its latest redefinition for the specified function. If `-file` *filename* is specified, `xdiff` shows the difference between the compiled file and the file with all redefinitions applied to the compiled source of the file (with the condition that the file has had only its functions redefined, and has not been edited since the last build). Applies to –o32 code only.

`source` *filename*

Executes commands in the specified file.

`status`

Displays a list of currently set breakpoints and traces.

`step [`*int*`]`

Steps the specified number of source instructions. This command steps into procedures. The default is one instruction.

`stepi [`*int*`]`

Steps the specified number of machine instructions. This command steps into procedures. The default is one line.

`stop at [`*filename*`:]` *line_number*`[if` *expression*`]`

Traps at the specified line in the specified file. If the `if` option is used, the trap fires only if *expression* is true.

`stop in [`*filename*`:]` *function_name* `[if` *expression*`]`

Traps at the entry to the specified function. If the `if` option is used, then the trap fires only if *expression* is true. If the *filename* is given, the function is assumed to be in that file's scope.

`syscall catch | ignore [call | return] \ [`*sys_call_name* `| all]`

The `catch` option adds a system call to the list of system calls to be trapped. The `ignore` option removes a system call from the system call trap list. The `call` option specifies the entry to the system call and `return` signifies the return from the call.

`trace [`*variable*`] at [["`*filename* [*line_number* ":] \ | *function_name*] \ [if *expression*]]

Traces the specified variable. You can specify a file and/or test condition. You can also specify a line number or a function where the trace is to take place.

`unalias` *aliasname*

Cancels the alias specified as *aliasname*.

`undisplay [`*displaynumber*`, ...]`

Stops display of expression with specified *displaynumber* when the process stops. Removes the expression from the display list.

`unsetbuildenv ["`*filename*`"]`

Disregards the default build environment flags if specified earlier. For all functions in files that don't have an overriding build environment, `unsetbuildenv` passes only the `-c` and `-g` flags.

If *filename* is given, this command disregards the build environment flags specified for the file earlier. Further redefinition of the functions in the file use the default build environment flags, if set. Applies to –o32 code only. See also `setbuildenv`.

`up` [*expression*]

Moves up the specified number of frames in the call stack. `up` moves in the direction of the caller.

`use` [*path*]

Uses the specified path to search for source files.

`whatis` *identifier*

Displays all the qualifications of the specified variable.

`when at` *filename*`:]` *line_number* {*command*[`;` *command* `...`]}

Stops the process and performs other Debugger commands when the process reaches a specified line number.

`when in` [*filename*`:]` *function_name* {*command* [`;` *command* `...`]}

Stops the process and performs other Debugger commands at entry to function. If the *filename* is given, the function is assumed to be in that file's scope.

`which` [*identifier*]

Displays the qualification of the specified variable.

`where` [*thread*]

Performs a stack trace showing the activation levels of a program or, optionally, of the specified thread.

Using the Build Manager [B]

WorkShop lets you compile software without leaving the WorkShop environment. Thus, you can look for problems using the WorkShop analysis tools (Static Analyzer, Debugger, and Performance Analyzer), make changes to the source, suspend your testing, and run a compile. WorkShop provides two tools to help you compile:

- *Build View*—for compiling, viewing compile error lists, and accessing the code containing the errors in `Source View` (the WorkShop editor) or an editor of your choice. `Build View` helps you find files containing compile errors so that you can quickly fix them, recompile, and resume testing.
- *Build Analyzer*—for viewing build dependencies and recompilation requirements and accessing source files.

`Build View` uses the UNIX `make`(1) facility as its default build software. Although `cvmake` can be set up to run any program instead of make (for example, `gnumake`), `cvbuild` will only parse and display standard makefiles (in particular, it does not understand `gnu` make constructs).

B.1 `Build View` Window

You can access the `Build View` window from the WorkShop analysis tools, from the command line (by typing **`cvmake`**), or from the `Build Analyzer` (see next section).

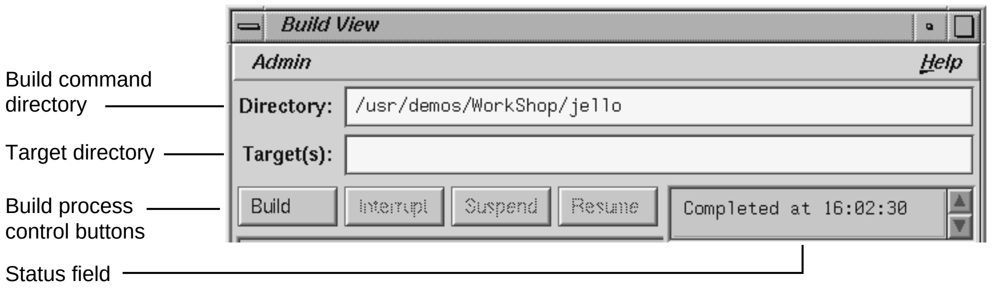
To access `Build View` from WorkShop, select `Recompile` from the `Source` menu in the Main View window in the Debugger or from the `File` menu in `Source View` (for more information on Main View and `Source View`, refer to Chapter 1, page 1). Selecting `Recompile` detaches the current executable from the WorkShop analysis tools and displays `Build View`. You can edit the `Directory` and `Target(s):` fields as needed and click `Build` to compile. If the source compiles successfully, the new executable is reattached when you reenter the WorkShop analysis tools.

The `Build View` window has three major areas:

- Build Process Control Area, Section B.2, page 260
- Transcript Area, Section B.3, page 261
- Error List Area, Section B.4, page 261

B.2 Build Process Control Area

The build process control area lets you run or stop the build and view the status. See Figure 129, page 260.



/usr/demos/WorkShop/jello

Completed at 16:02:30

Figure 129. Build Process Control Area in `Build View` Window

The directory in which the build will run displays in the `Directory:` field at the top of the area. The current directory displays by default. You can specify the build using `make`, `smake`, `pmake`, `clearmake`, or any other builder and any flags or options that the builder understands (see Section B.5.1, page 262, and Section B.5.2, page 263). The target to be built is specified in the `Target(s):` field.

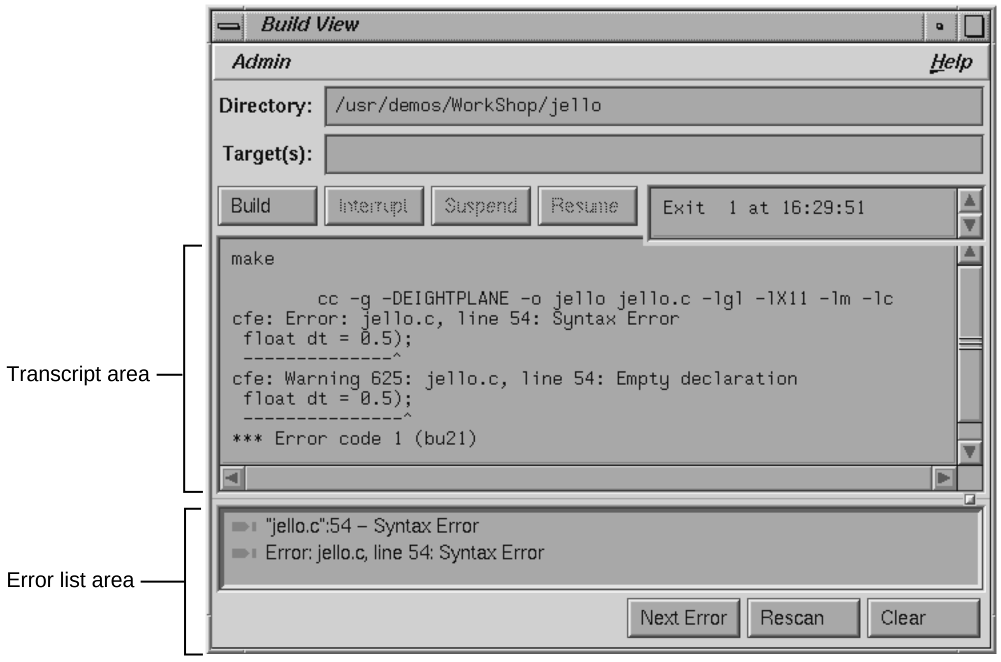
The build process control buttons let you control the build process. The following buttons are available:

`Build`	Runs (or reruns) a build. If you have modified any files you will be prompted to save the new versions prior to the compile.
`Interrupt`	Stops a build.
`Suspend`	Stops a build temporarily.
`Resume`	Restarts a suspended build.

The status field is to the right of the build process control buttons. It indicates the progress of the build.

B.3 Transcript Area

The transcript area displays the verbatim output from the build. The vertical scroll bar lets you go through the list; the horizontal scroll bar lets you see long messages obscured from view. A sash between the compile transcript area and the error list area lets you adjust the lengths of the lists displayed. See Figure 130, page 261.



```
make

        cc -g -DEIGHTPLANE -o jello jello.c -lgl -lX11 -lm -lc
cfe: Error: jello.c, line 54: Syntax Error
 float dt = 0.5);
 --------------^
cfe: Warning 625: jello.c, line 54: Empty declaration
 float dt = 0.5);
 ---------------^
*** Error code 1 (bu21)
```

```
"jello.c":54 – Syntax Error
Error: jello.c, line 54: Syntax Error
```

Figure 130. `Build View` Window with Typical Data

B.4 Error List Area

The error list area consists of the error list display and three control buttons. The following buttons are available:

`Next Error`	Brings up the default editor scrolled to the next error location. This button is below the error list display.
`Rescan`	Refreshes the error list display.

`Clear`	Clears the error list display area.

The error list area displays compile errors (see Figure 130, page 261). The errors are annotated according to their severity level (fatal has a solid icon and the warning icon is hollow). Double-clicking the text portion of an error brings up the default editor scrolled to the error location and displays a check mark to help you keep track of where you are in the error list. Check marks also display when you click the `Next Error` button.

B.5 Build View Admin Menu

The `Admin` menu in `Build View` has two selections in addition to the standard WorkShop entries:

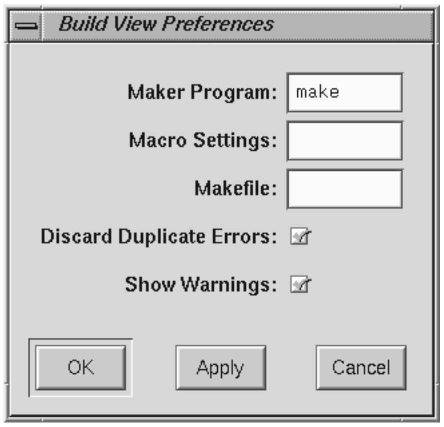
- Build View `Preferences...`, Section B.5.1, page 262
- `Build Options...`, Section B.5.2, page 263

For information on `Launch Tool`, `Project`, and `Exit` menu selections, refer to Section A.1.1, page 123.

B.5.1 Build View Preferences

The `Preferences...` selection brings up the dialog box shown in Figure 131, page 263. The options are:

`Maker Program` field	Lets you enter the program you use to build your executable.
`Macro Settings` field	Lets you enter build macros, such as `CFLAGS=-g`.
`Makefile` field	Lets you enter the name of a makefile if you do not wish to use the default.
`Discard Duplicate Errors` button	Eliminates subsequent duplicates of errors in the error list area.
`Show Warnings` button	Toggles the option to display warnings in the list.

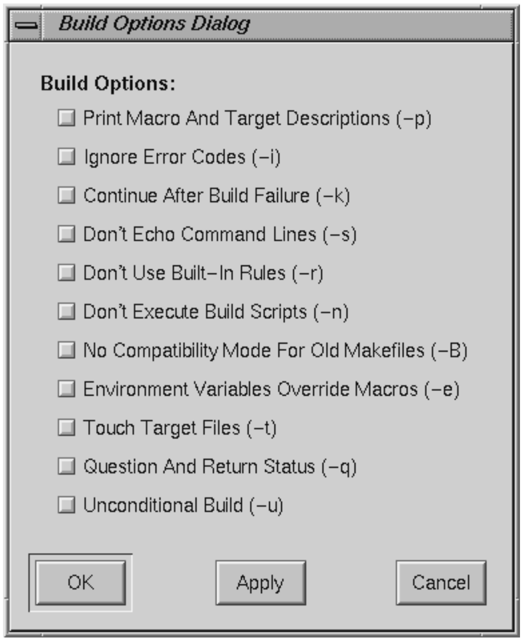


Maker Program: make
Macro Settings:
Makefile:
Discard Duplicate Errors: ✓
Show Warnings: ✓

Figure 131. `Build View Preferences` Dialog

B.5.2 Build Options

The `Build Options Dialog` lets you add the options shown in Figure 132, page 264, to your `make` command.


Build Options Dialog

Build Options:

- ☐ Print Macro And Target Descriptions (–p)
- ☐ Ignore Error Codes (–i)
- ☐ Continue After Build Failure (–k)
- ☐ Don't Echo Command Lines (–s)
- ☐ Don't Use Built–In Rules (–r)
- ☐ Don't Execute Build Scripts (–n)
- ☐ No Compatibility Mode For Old Makefiles (–B)
- ☐ Environment Variables Override Macros (–e)
- ☐ Touch Target Files (–t)
- ☐ Question And Return Status (–q)
- ☐ Unconditional Build (–u)

Figure 132. `Build Options Dialog`

B.5.3 Using `Build View`

The steps in running a compile using `Build View` are as follows:

1. Bring up the `Build View` window.
2. Edit the `Target(s):` and `Directory:` fields as required.
3. Specify your preference regarding duplicate errors and warnings using the `Admin` menu (optional).
4. Click `Build` to start the build. All compile information displays in the transcript area. Errors are grouped in a list below.
5. Click `Interrupt` to terminate or `Suspend` for a temporary stop, if you want to stop the build. The `Resume` button restarts a suspended build.
6. Double-click an error to bring up your preferred editor with the appropriate source code. A check mark indicates that an error has been accessed.

Note: The default editor is determined by the `editorCommand` resource in the `app-defaults` file. The value of this resource defaults to `wsh -c vi +%d`, which means run `vi` in a `wsh` window and scroll to the current line. If the editor lets you specify a starting line, enter **`%d`** in the resource to indicate the new line number.

7. Click `Build` to restart the build.

B.6 `Build Analyzer` Window

The `Build Analyzer` window displays a graph indicating the source files and derived files in the build, and their dependency relationships and current status. Source files refers to input files, such as code modules, documentation, data files, and resources. Derived files refers to output files, such as compiled code. You request builds in `Build Analyzer` by either:

- Double-clicking a derived module
- Making a selection from the `Build` menu

You access `Build Analyzer` from WorkShop by selecting `Launch Tool` from the `Admin` menu in Main View. Outside of WorkShop, you can access `Build Analyzer` by typing **`cvbuild`** at the command line. A typical `Build Analyzer` window appears in Figure 133, page 266, with the menus displayed.

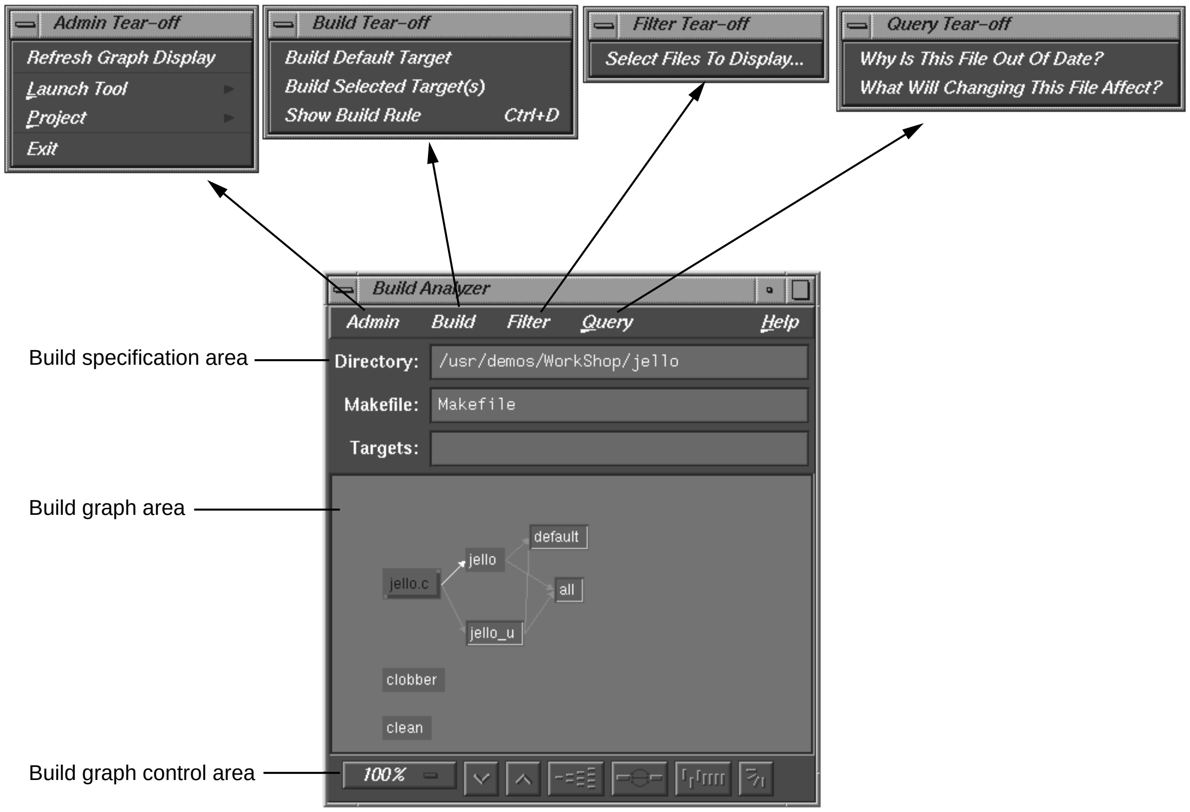


Figure 133. `Build Analyzer` Window

B.7 Build Specification Area

The three fields in the build specification area identify the working directory, makefile script, and target file(s) for compilation. You can edit the `Directory:`, `Makefile:`, and `Target(s):` fields directly. The `Target(s):` field also lets you specify a search string for locating a file in the build graph.

B.8 Build Graph Area

The build graph area displays the specified source and derived files and their dependency relationships. Files are depicted as rectangles; dependency relationships are shown as arrows, with the supplying file at the base of the arrow and the dependent file at the head. The colors used to depict the files depends on your color scheme. `Build Analyzer` differentiates the two types of files by depicting one with light characters on a dark background and the other with dark text on a light background. If you double-click a source file icon, an editor is brought up for that file. Double-clicking a derived file starts a build and displays Build View.

In addition to dependency relationships, Build Analyzer indicates the status of the files and relationships as follows:

- Source file availability status: `normal` or `checked out`
  - Normal means that the source file is read-only and needs to be made writable to be edited. Normal files appear as light rectangles with black text.
  - `Checked out` means that you have a writable version of this file available and can thus edit it. A checked out file appears in a different color (from normal files) with a shadow.
- Derived file compile status: `current` or `obsolete`
  - When applied to a derived file, the term current means that none of the files on which the derived file depends have been edited since the derived file was created. Current derived files appear as dark rectangles with white text.
  - `Obsolete` means that one or more of the source files have been modified since the derived file was created. Obsolete files appear in the same color as current derived files but with a colored outline.
- Dependency relationship: `current` or `obsolete`
  - `Current` means that the derived file is up to date with the source files. Note that a relationship can be current even if both files are obsolete. This happens when a file on which both files are dependent has been modified. Current arcs are black.
  - `Obsolete` means that the source file has changed and the derived file has not been updated accordingly. Obsolete arcs appear as colored arrows.

Some typical build graph icons are shown in Figure 134, page 268.

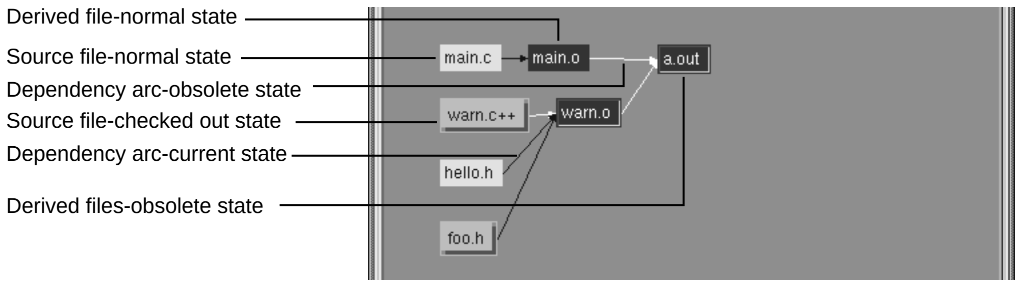

Derived file-normal state
Source file-normal state
Dependency arc-obsolete state
Source file-checked out state
Dependency arc-current state
Derived files-obsolete state
main.c
main.o
a.out
warn.c++
warn.o
hello.h
foo.h

Figure 134. Build Graph Icons

The `main.c` and `hello.h` source files are in their normal state. The source files `warn.c++` and `foo.h` are checked out and thus appear highlighted and with dropped shadows. The derived file `main.o` is current, since it has not changed since the last compile. The black dependency arcs indicate that the source and derived files at either end are current with each other. When an arc is highlighted, it indicates that the source has changed since the last compile. The derived files `warn.o` and `a.out` are obsolete because `warn.c++` has changed.

B.9 Build Graph Control Area

The build graph control area contains a row of graph control buttons similar to the ones in the WorkShop Static Analyzer and the `Call Graph View` in the Performance Analyzer. The `Overview` button is particularly useful in the `Build Analyzer` because it helps you quickly find obsolete files where a lot of dependencies are involved.

The build graph control area is shown in Figure 135, page 268.

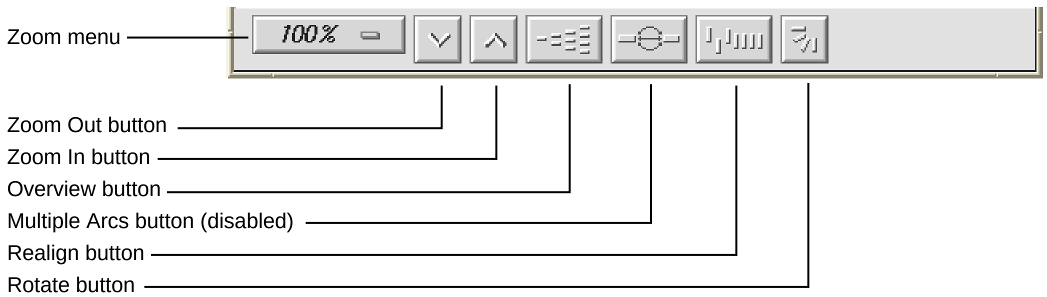
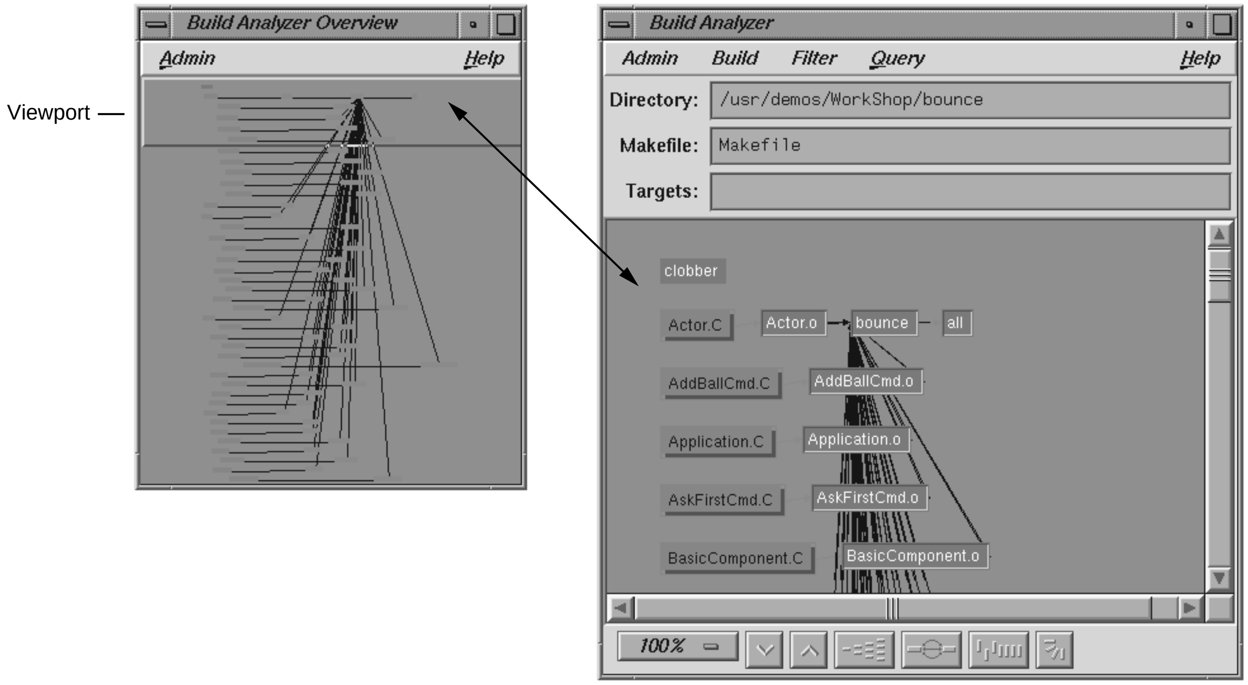


Figure 135. Build Graph Control Area

B.9.1 `Build Analyzer Overview` Window

Since build graphs can get quite complicated, an overview mode (similar to those in Static Analyzer and Profiling View) is supplied that lets you view the entire graph at a reduced scale. To display the overview window, you click the `overview` icon (see Figure 135, page 268).

Figure 136, page 269, shows a typical `Build Analyzer Overview` window with the resulting graph. The window has a movable viewport that lets you select the portion of the build graph displayed in `Build Analyzer`. Source files that have changed and derived files needing recompilation are highlighted for easy detection. In this particular color scheme, the `Build Analyzer Overview` window displays normal source files in turquoise, checked out source files in pink, current derived files in dark blue, and obsolete derived files in yellow. Arcs appear only in black in this window.



Directory: /usr/demos/WorkShop/bounce
Makefile: Makefile
Targets:

Figure 136. `Build Analyzer Overview` Window with Build Analyzer Graph

B.9.2 Build Analyzer Menus

The Build Analyzer window contains the following menus:

- Admin
- Build
- Filter
- Query

B.9.2.1 Admin Menu

The Admin menu provides one selection Refresh Graph Display in addition to the standard WorkShop selections.

Refresh Graph Display	Refreshes the window.
Launch Tool	Lets you execute the WorkShop tools. For more information, see the Section A.1.1, page 123.
Project	Lets you control the WorkShop tools operating on the same executable as a group. For more information, see Section A.1.1, page 123.

B.9.2.2 Build Menu

The selections in the Build menu let you perform builds as follows:

Build Default Target	Performs a make with no arguments.
Build Selected Target(s)	Performs the build(s) as entered in the Target(s): field.
Show Build Rule	Displays a dialog box showing the makefile line for the selected node.

B.9.2.3 Filter Menu

The Filter menu has only one selection:

Select files to show in graph	Opens the File Filter dialog box that lets you enter a regular expression to filter files displayed in the build graph.

The upper list area lets you specify files to be excluded from the build graph. The lower list is for specifying files to appear in the graph.

B.9.2.4 `Query` Menu

The `Query` menu lets you request information about the build graph. The following selections are available:

`Why Is This File Out Of Date?`	Identifies the source files requiring this file to be recompiled. This query only applies to derived files.
`What Will Changing This File Affect?`	Shows all derived files dependent on this source file.

Index

A

access to freed memory, 79
Access to uninitialized memory, 79
accessing files, 17
Active selection in admin menu, 30
active toggle, 150, 157
Add button in trap manager, 29
adding a breakpoint, 109
Address... selection in disassemble menu, 223
Admin menu, 235
  general description, 124
  Library search path..., 124
admin menu, 157
  active toggle, 150, 157
  clone, 150, 157
  close, 151, 157
  save as text, 150, 157
alias debugger command, 245
all trap debugger command option, 44
arguments, command line, 51
Arrange selection in structure browser display menu, 212
Array browser, 4, 36
  general description, 192
  subscript controls, 37
Array browser selection in views menu, 36, 126
Array field in array browser, 192
array subscripts, 193
array variables, 4, 192
assign debugger command, 245
assigning values to variables, 63
attach debugger command, 245
Automatic dereference limit field in structure browser preferences box, 216

B

boundary overruns, 79
boundary underrun, 79
breakpoint, 3
breakpoint results, viewing, 115
breakpoint type option button, 160
breakpoint, adding, 109
breakpoints examiner, 158
  callback, 160
  event-handler, 162
  input-handler, 167
  resource-change, 164
  state-change, 168
  timeout-procedure, 165
  X-event, 170
breakpoints tab, 113
breakpoints, setting, 73
breakpoints, setting for a class, 110
Build analyzer, 265
Build environment window, 239
Build manager, 259
build path, 10
Build view, 259

C

C expressions, 63
C function calls, 64
C++ expressions, 64
call debugger command, 245
Call stack, 4
Call stack selection in views menu, 30, 126
Call stack view, 30, 57, 204
call stack view, 243
callback breakpoints examiner, 160
callback context, viewing, 111

callback examiner, 111, 179
callstack view, 115
catch debugger command, 245
change id, 10
changes, re-enabling, 76
classes, examining widget, 109
Clear all selection in structure browser display menu, 214
Clear button in trap manager, 29
clear debugger command, 245
Clear trap selection in traps menu, 42
clearbuffer debugger command, 245
clearcalls debugger command, 245
Click for help selection in help menu, 139
clone current window, 150, 157
close current window, 151, 157
code, changing, 71
code, changing from command line, 72
code, comparing, 76
code, deleting changed, 72
code, switching between compiled and redefined, 76
Col button in array browser, 36
Column width... selection in array browser display menu, 195
Command field in main view, 51
command line interface, 242
comparing function definitions, 77
Condition field in trap manager, 46
Config menu in structure browser, 210
Config menu in trap manager, 43
cont in debugger command, 245
cont to debugger command, 245
continue signal debugger command, 246
Continue button in main view, 28, 29, 52
continue debugger command, 246
Continue to selection in disassembly view pc menu, 223
Continue to selection in pc menu, 55, 134
corefile debugger command, 246
cvd
  Execution view, 104
  Main view, 103
Cycle count field in trap manager, 30, 47

D

data structures, 4
dbx commands, 244
Debugger
  call stack view, 243
  changes to views, 241
  command line interface, 242
  exiting, 2
  main view, 241
  process execution control, 51
  starting, 1, 23
  trap manager, 243
debugger
  Execution view, 104
  Main view, 103
Debugger command line, , 2, 244
Debugger data, 57
Debugger views, 57, 204, 218
Debugger with fix and continue support
  Fix and continue
    debugger support with, 10
Debugger, exiting, 79
Default field count field in structure browser type formatting, 217
Default iconic width field in structure browser preferences box, 216
Default iconic width field in structure browser type formatting box, 217
Default state field in structure browser type formatting box, 217
Default structure field count field in structure browser preferences box, 216
Default structure width field in structure browser preferences box, 216
Default structure width in structure browser type formatting box, 217
delete all debugger command, 246
delete trap debugger command, 247

Dereference ptrs by default field in structure browser preferences box, 216
Dereference ptrs selection in structure browser node menu, 215
detach, 247
Detach selection in admin menu, 125
Detail selection in structure browser submenu, 214
difference tools, 78
disable all debugger command, 247
disable debugger command, 247
disabling traps, 29
Disassemble file dialog box, , 225
Disassemble function dialog box, , 224
Disassemble menu in disassembly view, 223
disassembled code, 4
Disassembly view, 4
  preferences, , 225
Disassembly view selection in views menu, 126
display area in structure browser, 211
display debugger command, 247
Display menu
  Main view, 130, 270
Display menu in structure browser, 210
Display menu in traps manager, 43
Display selection in structure browser display menu, 212
double frees, 79
down debugger command, 248
dump debugger command, 248

E

enable all debugger command, 248
enable trap debugger command, 248
environment variables
  setting, 55, 140
erroneous frees, 79
Error messages window, 238
event examiner, 181
event-handler breakpoints examiner, 162
examin menu
  widget tree, 157
examine menu, 157
  selection, 157
  widget, 157
  widget class, 157
  X event, 157
examiner
  breakpoint, 109
  breakpoints, 158
  callback, 111, 179
  callback breakpoints, 160
  event, 181
  event-handler breakpoints, 162
  graphics context (GC), 182
  input-handler breakpoints, 167
  pixmap, 183
  resource-change breakpoints, 164
  state-change breakpoints, 168
  timeout-procedure breakpoints, 165
  trace, 175
  tree, 107, 178
  widget, 106, 177
  widget class, 184
  window, 112, 180
  X-event breakpoints, 170
examiner menu
  X graphics context, 158
  X pixmap, 158
examiner tabs, 158
examiners
  overview, 12
  selections, 13
examining debugger data, 4, 57
Examining view data, 30
examining widget classes, 109
examining widgets, 108
Exception view selection in views menu, 126
execution control buttons, 52
Execution view, 55, 104, 140
Execution view selection in views menu, 126
Exit selection in admin menu, 126
exiting debugger, 79

exiting the debugger, 2
Expression column in expression view, 60, 207
expression count debugger command, 249
Expression field in structure browser, 34, 211
Expression selection in structure browser
    display submenu, 212
Expression view, 4, 33, 60, 206
Expression view selection in views menu, 33, 126
expressions
  C, 63
  C++, 64
  Fortran, 64

F

File browser, 17
File browser selection in views menu, 126
file debugger command, 250
File menu, source view, 146
"File..." selection in disassemble menu, 224
files
  opening, 18
files, comparing source code, 78
files, finding, 11
finding files, 11
Fix and continue
  basic cycle, 7
  build path, 10
  change id, 10
  environment, 10
  functionality, 7
  redefining functions with, 6
  restrictions, 9
  starting, 6
  WorkShop integration, 8
Fix+continue
  breakpoints, 73
  Build environment window, 239
  changing code, 71
  changing code from command line, 72
  deleting changed code, 72
  editing a function, 70
  Error message window, 238
  GUI, 232
  menu operations, 134
  sample session, 67
  Session, 236
  Show difference, 135
  Status window, 76, 234
  traps, 73
  View, 136
Fix+Continue menu, 236
Fork editor selection in source menu, 4, 128
Format menu in expression view, 33, 60, 206, 208
Format menu in structure browser, 211
Format menu in variable browser, 219
formatting fields in structure browser, , 215
Fortran expressions, 64
Fortran function calls, 67
Fortran variables, 65
frames, 57, 204
func debugger command, 250
function definitions, comparing, 77
function, editing, 70
function, redefining
  Fix+continue
    redefining functions, 69
Function... selection in disassemble menu, 223
functions, identifying, 10

G

Geometry selection in structure browser node
    menu, 214
givenfile debugger command, 250
GLdebug, 124
GLdebug selection in admin menu, 124
Go to line... selection in source menu, 129
goto debugger command, 250
Goto dialog box, 129
graphics context (GC) examiner, 182

H

heap corruption
  detection, 79
heap corruption problems
  defined, 79
Help menu, 139
Hide icons selection in display menu, 131
Hide line numbers selection in display menu, 130

I

Iconic selection in structure browser submenu, 214
Iconify selection in admin menu, 125
identifying functions, 10
ignore debugger command, 250
index identifiers in array browser, 194
index maximum specification in array browser, 194
index minimum specification in array browser, 194
index sliders in array browser, 194
index values in array browser, 194
Index... selection in help menu, 139
Indexing expression field in array browser, 193
input-handler breakpoints examiner, 167
Insert source... selection in source menu, 128
integration of workShop tools, 5
interface, command line, 242

J

jello program, 23
Jump to selection in disassembly view pc menu, 223
Jump to selection in pc menu, 55, 134

K

Keys & shortcuts selection in help menu, 139
Kill button in main view, 52
kill debugger command, 250

L

Language menu in expression view, 33, 60, 206, 207
Language menu in variable browser, 219
Launch selection in admin menu, 126
launching the x/Motif analyzer, 11
launching x/Motif analyzer, 105
$LD_LIBRARY_PATH, setting, 12
Library search path dialog box, 124
Linked list selection in structure browser display menu, 213
list debugger command, 250
Load expressions... selection in expression view config menu, 62
Load settings... selection in admin menu, 125
Load traps... selection in config menu in trap manager, 48

M

Main view, 103
  Command field, 51
  Continue button, 52
  control panel, 51
  Display menu, 130, 270
  general description, 2
  Kill button, 52
  PC menu, 55
  Run button, 52
  Sample button, 55
  Status field, 52
  Step into, 52
  Step over button, 54

Stop button, 52
main view, debugger, 241
Make editable selection in source menu, 128
Make read only selection in source menu, 128
managing source files, 17
Maximize selection in structure browser node submenu, 214
memory locations, 4
Memory view, 4, 230
Memory view mode menu, 232
Memory view selection in views menu, 126
menu operations, 134
Message window
Admin menu, 239
buttons, 238
View menu, 239
Messages window, 238
Minimize selection in structure browser node submenu, 214
Minimum lines around current instruction field in disassembly view preferences box, 226
multiprocess traps, 43
multiprocess view, 89
Multiprocess view... selection in admin menu, 124

N

"N..." selection in step into menu, 53
"N..." selection in step over menu, 53
next debugger command, 251
nexti debugger command, 251
Node menu in structure browser, 211
Node pop-up menu in structure browser, 212
Normal selection in structure node submenu, 214
Number of instructions to disassemble field in disassembly view preferences box, 226

O

Open... selection in source menu, 128
opening files, 18
Overview selection in help menu, 139

P

path remapping, 19
Pattern layout in structure browser node menu, 215
PC, 134
PC menu, 55, 134
Continue to, 55
Jump to, 55
PC menu in disassembly view, 223
PC menu in main view, 55
performance data
Sample button, 55
pgrp trap debugger command option, 44
pixmap examiner, 183
pollpoint, 4
pollpoint trap debugger command option, 45
Preference menu, 136
preparing the fileset, 103
print expression debugger command, 251
printd expression debugger command, 251
printo expression debugger command, 251
printregs debugger command, 251
printx expression debugger command, 251
process execution control, 51
Main view control panel, 51
PC menu, 55
Process meter selection in views menu, 126
Product information selection in help menu, 140
program counter, 55, 134
program output, tracking, 11
pwd debugger command, 251

Q

quit debugger command, 251

R

Raise selection in admin menu, 125
Raise selection in structure browser node submenu, 214
Read-Only
  debugger status, 10
Recompile selection in the source menu, 128
redefining functions, 6
Register name display format field in disassembly view preferences box, 226
Register view, 4, 227
Register view formatting, 230
Register view preferences dialog box, 230
Register view selection in views menu, 127
Register view window, 228
registers, 4
"Remap paths..." selection in session menu, 19
Remove selection in structure browser node menu, 215
removing traps with mouse, 42
rerun debugger command, 253
resource-change breakpoints examiner, 164
restrictions and limitations, 13
Result column in expression view, 60, 208
Return button in main view
  Main view
    Return button, 54
return debugger command, 253
row/column toggles in array browser, 194
Run button in main view, 52
run debugger command, 253

S

Sample At Function Entry, 133
Sample at function entry selection in traps submenu, 41
Sample at function exit selection in traps submenu, 41, 134
Sample button in main view, 55
sample session, 103
  Interpreter, 67
  preparing fileset, 103
  setting up, 103
sample session setup, 67
sample trap, 40
sample trap command, 43
sample traps, 3
save as text, 150, 157
Save as text... selection in source menu, 128
Save as... selection in source menu, 128
Save expressions... selection in expression view config menu, 62
Save selection in source menu, 128
Save settings... selection in admin menu, 125
Save traps... selection in config menu in trap manager, 48
saving to source file, 73
Search field in trap manager, 48
"Search" selection in source menu, 25
"Search" selection in structure browser display menu, 213
Search... selection in source menu, 128
Select selection in structure browser node menu, 214
selection, 157
selection in structure browser display menu, 212
Session submenu, 236
Set trap selection in traps menu, 41, 133
setting traps, 27
setting traps with the mouse, 42
sh debugger command, 254
Show difference submenu, 135
Show embedded source annotation field in disassembly view preferences box, 226
Show icons selection in display menu, 131
Show instruction value field in disassembly view preferences box, 227
Show jal target numerically field in disassembly view preferences box, 227
Show line numbers selection in display menu, 130

Show machine address field in disassembly view preferences box, 227
Show overview selection in structure browser display menu, 213
Show source file and line number field in disassembly view preferences box, 226
Signal panel, 48
Signal panel selection in views menu, 127
signal trap debugger command option, 45
signals
  traps, 3
source annotation column
  traps, 41
source code display area, 2
source code status indicator, 10, 68
source debugger command, 255
source file, saving to, 73
source files
  managing, 17
Source view
  File menu, 146
Source view selection in views menu, 127
special libraries, 12
spreadsheet area in array browser, 194
stack frame, 31
stack frames, 58, 205
starting fix and continue, 6
starting the debugger, 1, 23
starting the x/Motif analyzer, 11
starting, process execution, 52
State selection in structure browser node menu, 214
state-change breakpoints examiner, 168
status debugger command, 255
Status field in main view, 51, 52
status line
  Main view, 2
Status window, 76, 234
  Admin menu, 235
  Fix+Continue menu, 236
  Preference menu, 136
  View menu, 235
status, viewing, 76
step debugger command, 255
step indicators in array browser, 194
Step into button in disassembly view, 222
Step into button in main view, 52
Step over button in disassembly view, 222
Step over button in main view, 53
stepi debugger command, 255
stop at debugger command, 255
Stop at function entry selection in traps submenu, 41, 133
Stop at function exit selection in traps submenu, 41, 133
Stop button in main view, 52
stop in debugger command, 255
stop trap, 40
stop trap command, 43
stop traps, 3, 27
stopping, process execution, 52
Structure browser, 4
  general description, 209
Structure browser preferences dialog box, 216
Structure browser selection in views menu, 34, 127
subscript controls in array browser, 37
subscripts
  array, 193
Switch executable... selection in admin menu, 125
Switch process dialog box, 125
Switch process... selection in admin menu, 125
syscall debugger command, 255
Syscall panel, 48
Syscall panel selection in views menu, 127
syscall trap debugger command option, 45
system calls
  traps, 4

T

tab overflow area, 113
tabs, 114
tabs, examiner, 158

Task view selection in views menu, 127
timeout procedure breakpoints examiner, 165
trace debugger command, 256
trace examiner, 175
tracking program output, 11
trap actions, 39
trap condition, 46
trap examples, 46
Trap manager, 28
trap manager, 243
Trap manager menus, 43
Trap manager selection in views menu, 127
trap terminology, , 39
traps, 3, 27
  disabling, 29
  general description, 39
  one-time, 55
  removing with mouse, 43
  setting conditions, 46
  setting cycle count, 47
  setting with mouse, 42
  Signal panel, 48
  Syscall panel, 48
  triggering, 39
Traps menu in main view, 41
Traps menu in trap manager, 43, 47
traps multiprocess, 43
traps, setting, 73
tree examiner, 107, 178
Tree selection in structure browser display submenu, 212
triggering traps, 39
Type color field in structure browser type formatting box, 218
Type formatting dialog, , 217
Type name field in structure browser type formatting box, 217

U

unalias debugger command, 256
undisplay debugger command, 256
Update selection in structure browser display menu, 213
use debugger command, 257
using
  Interpreter, 67
using the x/Motif analyzer, 12

V

Variable browser, 4, 31
  general description, 219
Variable browser selection in views menu, 31, 127
variables
  assignment, 63
view changes in debugger, 241
View menu, 235
View submenu, 136
view, call stack, 243
viewing status, 76
views
  Debugger, 219
Views menu in main view, 126

W

watch command, 29
watch trap debugger command option, 45
watchpoint, 29
watchpoints, 4
whatis debugger command, 257
when at debugger command, 257
when in debugger command, 257
where debugger command, 257
which debugger command, 257
widget class examiner, 184
widget class menu item, 157
widget classes, examining, 109
widget examiner, 177
widget hierarchy, 107
widget item, 157

widget structure, navigating, 105
widget tree menu item, 157
widgets, examining, 108
window attributes, viewing, 112
window examiner, 112, 180
window menu item
  examine menu
    window, 157
WorkShop integration, 8
Wrapped display selection in array browser display menu, 195

X

X event menu item, 157
X graphics context menu item, 158
X pixmap menu item, 158
X-event breakpoints examiner, 170
X/Motif analyzer
  additional features, 112
  default view, 106
  launching, 11, 105
  navigating widget structure, 105
  restrictions and limitations, 13
  sample session, 103
  starting, 11
  using, 12
X/Motif analyzer selection in views menu, 127
xdiff, 77

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2579-004.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389