

Developer Magic™ Debugger User's Guide

Document Number 007-2579-003

CONTRIBUTORS

Written by Douglas B. O'Morain, John Stearns, and Carol Geary

Illustrated by Douglas B. O'Morain, John Stearns, and Carol Geary

Production by Laura Cooper

Engineering contributions by David Henke, Stuart Liroff, Song Liang, Ashok Mouli,

Michey Mehta, Anil Pal, Kim Rachmeler, Jack Repenning, Krishna Sethuraman,

Ravi Shankar, John Templeton, Shankar Unni, and Mike Yang

St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower
image courtesy of Xaview Berenguer, Animatica.

©1996-97, Silicon Graphics, Inc. — All Rights Reserved

The contents of this document may not be copied or duplicated in any form, in whole
or in part, without the prior written permission of Silicon Graphics, Inc.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure of the technical data contained in this document by
the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the
Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/
or in similar or successor clauses in the FAR, or in the DOD or NASA FAR
Supplement. Unpublished rights reserved under the Copyright Laws of the United
States. Contractor / manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd.,
Mountain View, CA 94043-1389.

Silicon Graphics and IRIS are registered trademarks and IRIX, IRIS IM, IRIS ViewKit,
Indigo Magic, and Indigo Magic Desktop, are trademarks of Silicon Graphics, Inc.
Open Software Foundation, Motif, OSF, OSF/Motif are trademarks of the Open
Software Foundation, Inc. PostScript is a registered trademark of Adobe Systems, Inc.

Contents

List of Tables	xxv
Introduction	xxvii
What This Guide Contains	xxvii
What You Should Know Before Reading This Guide	xxviii
Related Information	xxix
Conventions	xxix
1. Getting Started with the WorkShop Debugger	1
Typical Debugger Usage	1
Starting and Exiting the Debugger	1
Using the Debugger From a Remote Host	2
Using Main View	2
Setting Traps	3
Inspecting Debugger Data	4
Changing Source Code	5
Updating and Saving Views	5
Integration With Other WorkShop Tools	6
Accessing the Performance Analyzer From Main View	6
Accessing the Static Analyzer From Main View	6
Accessing Editors From Main View	7
Accessing Configuration Management Tools	7
Recompiling From Main View or Source View	7

- Debugging with Fix+Continue 7
 - Redefining Functions Using Fix and Continue 7
 - Fix and Continue Functionality 8
 - Fix and Continue/WorkShop Integration 9
 - How Redefined Code Is Distinguished From Compiled Code 10
 - Restrictions on Fix and Continue 10
 - The Fix and Continue Environment 11
 - Debugger With Fix and Continue Support 12
 - GUI Debugger Command Line 12
 - Change ID, Build Path, and Other Concepts 12
- Debugging with the X/Motif Analyzer 13
 - Special Libraries 14
 - Using the X/Motif Analyzer 14
 - Examiners Overview 14
 - Examiners and Selections 15
 - Inspecting Data 15
 - Inspecting the Control Flow 15
 - Tracing the Execution 16
 - Restrictions and Limitations 16
- Customizing the Debugger 16
 - Using a Startup File 17
 - Implementing User-Defined Buttons 17
- 2. Managing Source Files 21**
 - Accessing Files Used by an Executable 21
 - Opening a New File 22
 - Path Remapping 23
- 3. A Short Debugger Tutorial 27**
 - Starting the Debugger 27
 - Performing a Search 29
 - Setting Traps 31
 - Examining Data 34

- 4. **Setting Traps** 43
 - Trap Terminology 44
 - Trap Triggers 44
 - Trap Actions 44
 - Setting Traps in Main View and Source View 45
 - Setting Traps With the Traps Menu in Main View 45
 - Setting Traps With the Mouse 47
 - Setting Traps in Trap Manager 48
 - Setting Single-process and Multiprocess Traps in the Trap: Field 48
 - Setting a Trap Condition 51
 - Setting a Trap Cycle Count 52
 - Setting a Trap With the Traps Menu and Source Display 52
 - Moving around the Trap Display Area 53
 - Enabling and Disabling Traps 53
 - Saving and Reusing Trap Sets 53
 - Setting Traps With Signal Panel and Syscall Panel 53
- 5. **Controlling Process Execution** 55
 - Main View Control Panel 55
 - Status and Entry Fields in the Main View Control Panel 56
 - Execution Control Buttons 56
 - Controlling Process Execution With PC Menu 59
 - Execution View 59
- 6. **Examining Debugger Data** 61
 - Tracing Through Call Stack View 61
 - Evaluating Expressions 64
 - Expression View 65
 - Assigning Values to Variables 67

- Evaluating Expressions in C 68
 - C Function Calls 68
- Evaluating Expressions in C++ 69
 - Limitations 69
- Evaluating Expressions in Fortran 70
 - Fortran Variables 70
 - Fortran Function Calls 71
- 7. Debugging with Fix+Continue: A Tutorial 73**
 - Setting Up the Sample Session 73
 - Redefining a Function 75
 - Editing a Function 75
 - Changing Code 77
 - Deleting Changed Code 79
 - Changing Code From the Debugger Command Line 79
 - Saving Changes 80
 - Setting Breakpoints in Redefined Code 81
 - Viewing Status 84
 - Comparing Original and Redefined Code 84
 - Switching Between Compiled and Redefined Code 85
 - Comparing Function Definitions 85
 - Comparing Source Code Files 86
 - Ending the Session 87
- 8. Detecting Heap Corruption 89**
 - Typical Heap Corruption Problems 89
 - Detecting Heap Corruption Errors 90
 - Compiling With the Malloc Library 90
 - Setting the Environment Variables 91
 - Trapping Heap Errors using the Malloc Library 92
 - Heap Corruption Detection Tutorial 93

- 9. **Multiple Process Debugging** 99
 - Debugging With Multiprocess View 99
 - Displaying the Multiprocess View 100
 - Viewing Process Status 101
 - Multiprocess Control Buttons 102
 - Multiprocess Traps 102
 - Adding and Removing Processes 103
 - Multiprocess Preferences 104
 - Controlling Execution and Setting Traps in a Multiprocess Program 105
 - Using the Multiprocess View to Control Execution 107
 - Using the Trap Manager to Control Trap Inheritance 109
 - Debugging a Multiprocess Fortran Program 111
 - General Fortran Debugging Hints 111
 - Multiprocess Debugging Session 112
- 10. **Using the X/Motif Analyzer: A Tutorial** 117
 - Setting Up the Sample Session 117
 - Preparing the Fileset 117
 - Launching the X/Motif Analyzer 119
 - Navigating the Widget Structure 120
 - Examining Widgets 122
 - Setting Callback Breakpoints 124
 - Using Additional Features of the Analyzer 126
 - Ending the Session 130

- A. Debugger Reference 131**
 - Main View 132
 - Admin Menu 138
 - Views Menu 142
 - Query Menu 144
 - Source Menu 145
 - Display Menu 147
 - Perf Menu 148
 - Traps Menu 153
 - PC Menu 154
 - Fix+Continue Menu 154
 - Show Difference Submenu 155
 - View Submenu 156
 - Preferences Submenu 156
 - Keyboard Accelerators 159
 - Help Menu 159
 - Basic Windows 160
 - Execution View 160
 - Source View 160
 - Menu Bar 161
 - Process Meter 163
 - Charts Menu 164
 - Scale Menu 164
 - Ada-specific Windows 165
 - Task View 165
 - Admin Menu 167
 - Config Menu 167
 - Layout Menu 168
 - Display Menu 168
 - Exception View 169

- X/Motif Analyzer Windows 171
 - Global Objects 172
 - Admin Menu 172
 - Examine Menu 173
 - Examiner Tabs 174
 - Return Button 174
 - Breakpoints Examiner 174
 - Callback Breakpoints Examiner 176
 - Event-Handler Breakpoints Examiner 178
 - Resource-Change Breakpoints Examiner 180
 - Timeout-Procedure Breakpoints Examiner 182
 - Input-Handler Breakpoints Examiner 184
 - State-Change Breakpoints Examiner 185
 - X-Event Breakpoints Examiner 188
 - X-Request Breakpoints Examiner 189
 - Trace Examiner 191
 - Widget Examiner 193
 - Tree Examiner 194
 - Callback Examiner 196
 - Window Examiner 196
 - Event Examiner 197
 - Graphics Context Examiner 198
 - Pixmap Examiner 199
 - Widget Class Examiner 200
- Project Session Management Windows 201
 - Project View 203
 - Project View Admin Menu 204
 - Project View Text Fields 204
 - Project Display Area 204
 - Project Popup Menu 204

- Trap Management Windows 204
 - Trap Manager 205
 - Config Menu 206
 - Traps Menu 206
 - Display Menu 207
 - Signal Panel 207
 - Syscall Panel 208
- Data Examination Windows 209
 - Array Browser 209
 - Spreadsheet Menu 213
 - Format Menu 214
 - Render Menu 214
 - Color Menu 215
 - Scale Menu 216
 - Examiner Viewer Controls 216
 - Examiner Viewer Menu 219
 - Call Stack View 221
 - Config Menu 222
 - Display Menu 223
 - Expression View 223
 - Config Menu 224
 - Display Menu 224
 - Language Popup 224
 - Format Popup 224
 - File Browser 225
 - Structure Browser 226
 - Using the Overview Window to Navigate 228
 - Entering Expressions 228
 - Working in the Structure Browser Display Area 229
 - Structure Browser Display Menu 230
 - Node Menu 232
 - Formatting Fields 233

Variable Browser	237
Entering Variable Values	237
Changing Variable Column Widths	238
Viewing Variable Changes	238
Machine-level Debugging Windows	239
Disassembly View	239
Similarities With Main View	240
Disassemble Menu	241
Disassembly View Preferences	242
Register View	244
Register View Window	245
Changing the Register View Display	246
Memory View	248
Viewing a Portion of Memory	248
Changing the Contents of a Memory Location	249
Changing the Memory Display Format	249
Moving around the Memory View Display Area	249
Multiple Process Debugging Windows	250
Multiprocess View	250
Viewing Process Status	251
Multiprocess Control Buttons	251
Multiprocess Traps	252
Adding and Removing Processes	252
Multiprocess Preferences	253
Fix+Continue Windows	254
Fix+Continue Status Window	255
Admin Menu	257
View Menu	258
Fix+Continue Menu	258
Fix+Continue Message Window	260
Admin Menu	262
View Menu	262

- Fix+Continue Build Environment Window 262
- Changes to Debugger Views 264
 - Main View 264
 - Command-Line Interface 266
 - Call Stack 266
 - Trap Manager 266
- Debugger Command Line 267
- B. Using the Build Manager 279**
 - Build View 279
 - Build Process Control Area 280
 - Transcript Area 281
 - Error List Area 282
 - Build View Admin Menu 283
 - Build View Preferences 283
 - Build Options 284
 - Using Build View 285
 - Build Analyzer 286
 - Build Specification Area 287
 - Build Graph Area 288
 - Build Graph Control Area 290
 - Overview Window 290
 - Build Analyzer Menus 291
 - Build Analyzer Admin Menu 291
 - Build Menu 292
 - Filter Menu 292
 - Query Menu 293
- Index 295**

List of Figures

Figure 1-1	Major Areas of the Main View Window	3
Figure 1-2	Admin Menu in Debugger Views	5
Figure 1-3	Fix and Continue Cycle	9
Figure 1-4	Line Numbers in Decimal Notation	10
Figure 1-5	Launching the X/Motif Analyzer	13
Figure 1-6	User-Defined Button Example	18
Figure 2-1	File Browser Window	21
Figure 2-2	Open Source File Dialog Box	22
Figure 2-3	Path Remapping Dialog Box	24
Figure 3-1	The Main View Window with <i>jello</i> Source Code	28
Figure 3-2	The <i>jello</i> Window	29
Figure 3-3	The Search Dialog Box	30
Figure 3-4	Search Target Indicators	31
Figure 3-5	Stop Trap Indicator	32
Figure 3-6	The Trap Manager Window	33
Figure 3-7	Call Stack View at spin Stop Trap	35
Figure 3-8	Variable Browser at spin	36
Figure 3-9	Variable Browser after Changes	37
Figure 3-10	Expression View With Language and Format Menus Displayed	38
Figure 3-11	Structure Browser Window With <i>jello_conec</i> Structure	39
Figure 3-12	Structure Browser Window With <code>next</code> Pointer Dereferenced	39
Figure 3-13	Array Browser Window for <i>shadow</i> Matrix	40
Figure 3-14	Subscript Control Area in Array Browser	41
Figure 4-1	Traps Menu in Main View	46
Figure 4-2	Typical Trap Icons	47
Figure 4-3	Config, Traps, and Display Menus in the Trap Manager	48

Figure 4-4	Trap Examples 51
Figure 4-5	Traps Menu in Trap Manager 52
Figure 4-6	Signal Panel and Syscall Panel 54
Figure 5-1	Main View Control Panel 55
Figure 5-2	Step Into Popup Menu and Dialog Box 57
Figure 5-3	Step Over Popup Menu and Dialog Box 58
Figure 5-4	PC Menu in Main View 59
Figure 6-1	Call Stack View Window With Config and Display Menus and Preferences Dialog Box 62
Figure 6-2	Tracing Through Call Stack View 64
Figure 6-3	Expression View With Major Menus Displayed 66
Figure 6-4	Change Indicators in Expression View 67
Figure 7-1	Execution View Icon 73
Figure 7-2	Debugger Main View With Fix and Continue Menu 74
Figure 7-3	Program Results in Execution View 75
Figure 7-4	Selecting a Function for Redefinition 76
Figure 7-5	Redefined Function 77
Figure 7-6	Checking Syntax Opens Fix and Continue Status Window 78
Figure 7-7	Report of Successful Redefinition 78
Figure 7-8	Bounce Window 79
Figure 7-9	Saving a Function File 80
Figure 7-10	Stopping After Breakpoints in Redefined Code 82
Figure 7-11	Call Stack BreakPoint Results 83
Figure 7-12	Trap Manager BreakPoint Results 83
Figure 7-13	Using the View Status Window 84
Figure 7-14	Comparing Compiled vs. Redefined Function Code: <i>xdiff</i> 86
Figure 8-1	Setting Traps to Detect Heap Corruption 95
Figure 8-2	Heap Corruption Warning Displayed in Execution View 95
Figure 8-3	Call Stack at Boundary Overrun Warning 96
Figure 8-4	Main View at Bus Error 97
Figure 8-5	Watch Point Error Displayed in Main View 98

Figure 9-1	Multiprocess View With Config and Process Menus Displayed 101
Figure 9-2	Process Menu in Multiprocess View 103
Figure 9-3	Add Process Dialog Box 103
Figure 9-4	Multiprocess View Preferences Dialog Box 104
Figure 9-5	Launching a Debug Session Dialog Box 108
Figure 9-6	Using the Multiprocess View to Examine Process State 109
Figure 9-7	Modifying a Trap to Affect a Process Group 110
Figure 9-8	Setting the Group Trap Default 110
Figure 9-9	Launching a New Debugging Session From Multiprocess View 115
Figure 9-10	Comparing Variable Values From Two Processes 116
Figure 10-1	Execution View Icon 117
Figure 10-2	Debugger Main View 118
Figure 10-3	Program Results in Execution View 119
Figure 10-4	First View of the X/Motif Analyzer (Widget Examiner) 120
Figure 10-5	Widget Hierarchy Displayed in the Tree Examiner 121
Figure 10-6	Adding a Breakpoint for a Widget 123
Figure 10-7	Setting Breakpoints for a Widget Class 124
Figure 10-8	Viewing the Callback Context With the Callback Examiner 125
Figure 10-9	Viewing Window Attributes With the Window Examiner 126
Figure 10-10	Selecting the Breakpoints Tab From the Overflow Area 128
Figure 10-11	Viewing Breakpoint Results in the Callstack View 129
Figure A-1	Major Areas of the Main View Window 132
Figure A-2	Show/Hide Annotations Button in Main View 137
Figure A-3	Admin Menu in Main View 138
Figure A-4	The Library Search Path Dialog Box 139
Figure A-5	The Switch Process Dialog Box 140
Figure A-6	The Switch Executable Dialog Box 140
Figure A-7	“Launch Tool” Submenu 141
Figure A-8	“Project” Submenu 142

Figure A-9	Views Menu in Main View	142
Figure A-10	Query Menu With Submenus	144
Figure A-11	Source Menu in Main View	145
Figure A-12	The Search Dialog Box	146
Figure A-13	Go to Dialog Box	146
Figure A-14	Versioning Submenu	147
Figure A-15	Display Menu in Main View	147
Figure A-16	Preferences Dialog Box	148
Figure A-17	Perf Menu and Subwindows	149
Figure A-18	Launching Performance Analyzer From Perf Menu	150
Figure A-19	Custom Task Dialog	151
Figure A-20	Traps Menu	153
Figure A-21	Set Trap Submenu	153
Figure A-22	Clear Trap Submenu	154
Figure A-23	PC Menu in Main View	154
Figure A-24	Fix+Continue Menu	154
Figure A-25	“Save File+Fixes As...” Popup Window	155
Figure A-26	Show Difference Submenu	155
Figure A-27	View Submenu	156
Figure A-28	Preferences Submenu	156
Figure A-29	Fix+Continue Preferences Dialog	157
Figure A-30	Help Menu	159
Figure A-31	Execution View	160
Figure A-32	Source View	161
Figure A-33	Source View File Menu	161
Figure A-34	Go To Line Dialog	162
Figure A-35	Process Meter	163
Figure A-36	Process Meter Charts Menu	164
Figure A-37	Process Meter Scale Menu	164
Figure A-38	Task View	165
Figure A-39	Task View Process Detail View	166
Figure A-40	Task View Callstack Detail View	167
Figure A-41	Task View Admin Menu	167

Figure A-42	Task View Config Menu	168
Figure A-43	Task View Layout Menu	168
Figure A-44	Task View Display Menu	168
Figure A-45	Exception View	169
Figure A-46	“When” Exception Option Menu	170
Figure A-47	Launching the X/Motif Analyzer	171
Figure A-48	Admin Menu	172
Figure A-49	“Save Text” Dialog	173
Figure A-50	Examine Menu	173
Figure A-51	Examiner Tabs	174
Figure A-52	Removing Tabs	174
Figure A-53	Breakpoints Examiner	175
Figure A-54	Breakpoint Type Option Button	176
Figure A-55	Callback Breakpoints Examiner	177
Figure A-56	Event-Handler Breakpoints Examiner	179
Figure A-57	Event Type Option Button	180
Figure A-58	Resource-Change Breakpoints Examiner	181
Figure A-59	Timeout-Procedure Breakpoints Examiner	183
Figure A-60	Input-Handler Breakpoints Examiner	184
Figure A-61	State-Change Breakpoints Examiner	186
Figure A-62	State Type Option Button	187
Figure A-63	X-Event Breakpoints Examiner	188
Figure A-64	Event Type Option Button	189
Figure A-65	X-Request Breakpoints Examiner	190
Figure A-66	“Request Type Selection” Dialog	191
Figure A-67	Trace Examiner	192
Figure A-68	Widget Examiner	193
Figure A-69	Tree Examiner	195
Figure A-70	Widget View Type Option Button	195
Figure A-71	Callback Examiner	196
Figure A-72	Window Examiner	197
Figure A-73	Event Examiner	198
Figure A-74	Graphics Context Examiner	199

Figure A-75	Pixmap Examiner 200
Figure A-76	Widget Class Examiner 201
Figure A-77	Iconify and Raise Facilities 202
Figure A-78	Project View Window with Menus 203
Figure A-79	Trap Manager 205
Figure A-80	Trap Manager Config Menu 206
Figure A-81	Trap Manager Traps Menu 206
Figure A-82	Trap Manager Display Menu 207
Figure A-83	Signal Panel 207
Figure A-84	Syscall Panel 208
Figure A-85	Array Browser With Display Menu Options 210
Figure A-86	Subscript Control Area in Array Browser 211
Figure A-87	Array Browser Spreadsheet Area 213
Figure A-88	Spreadsheet Menu 213
Figure A-89	Example of Wrapped Array 214
Figure A-90	Format Menu With Value Submenu 214
Figure A-91	Render Menu 214
Figure A-92	Color Menu 215
Figure A-93	Color Exception Portion of Array Browser Window 215
Figure A-94	Array Browser Graphic Modes 216
Figure A-95	Scale Menu 216
Figure A-96	Examiner Viewer With Controls and Menus 218
Figure A-97	Examiner Viewer Preferences Dialog Box 220
Figure A-98	Call Stack View 221
Figure A-99	Call Stack View Config Menu 222
Figure A-100	Call Stack View Display Menu 223
Figure A-101	Expression View 223
Figure A-102	Expression View Config Menu 224
Figure A-103	Expression View Display Menu 224
Figure A-104	Expression View Language Popup 224
Figure A-105	Expression View Format Popup with Submenus 225
Figure A-106	File Browser 226

Figure A-107	Structure Browser With the Config, Display, Node, and Format Menus 227
Figure A-108	Structure Browser Overview Window 228
Figure A-109	Structure Browser Format Menu 229
Figure A-110	Structure Browser Display Menu 230
Figure A-111	Tree and Linked List Arrangements of Structures 231
Figure A-112	Structure Browser Node Menu 232
Figure A-113	Structure Browser Preferences Dialog Box 234
Figure A-114	Type Formatting Dialog Box 235
Figure A-115	Variable Browser With Language and Format Menus 238
Figure A-116	Typical Variable Change Indicators 239
Figure A-117	Disassembly View With Disassemble Menu Displayed 240
Figure A-118	Disassemble Menu 241
Figure A-119	Disassemble From Address Dialog Box 241
Figure A-120	Disassemble Function Dialog Box 242
Figure A-121	Disassemble File Dialog Box 242
Figure A-122	Disassembly View Preferences Dialog Box with Format Popup Menu 243
Figure A-123	Register View 245
Figure A-124	Register View Preferences Dialog Box 247
Figure A-125	Memory View With Mode Menu Displayed 248
Figure A-126	Multiprocess View with Config and Process Menus Displayed 250
Figure A-127	Process Menu in Multiprocess View 252
Figure A-128	Add Process Dialog Box 252
Figure A-129	Multiprocess View Preferences Dialog Box 253
Figure A-130	Fix+Continue Menu Selections 255
Figure A-131	Fix+Continue Status Window 256
Figure A-132	Fix+Continue Status Window Menus 257
Figure A-133	Status Window Admin Menu 257
Figure A-134	Status Window View Menu 258
Figure A-135	Status Window Fix+Continue Menu 258
Figure A-136	Show Difference Submenu 258

Figure A-137	Enable Submenu	259
Figure A-138	Save Submenu	259
Figure A-139	File Dialog	259
Figure A-140	Show Submenu	260
Figure A-141	Fix+Continue Message Window	261
Figure A-142	Fix+Continue Build Environment Window	263
Figure A-143	Debugger Main View	265
Figure A-144	Command-Line Interface With Redefined Function	266
Figure A-145	Call Stack	266
Figure A-146	Trap Manager With Redefined Function	267
Figure A-147	Editing a Function in the vi Editor	273
Figure B-1	Build View Window With Admin Menu Displayed	280
Figure B-2	Build Process Control Area in Build View Window	281
Figure B-3	Build View Window With Typical Data	282
Figure B-4	Admin Menu in Build View Window	283
Figure B-5	Build View Preferences Dialog Box	284
Figure B-6	Build Options Dialog Box	285
Figure B-7	Build Analyzer Window	287
Figure B-8	Build Graph Icons	289
Figure B-9	Build Graph Control Area	290
Figure B-10	Overview Window With Resulting Build Analyzer Graph	291
Figure B-11	Admin Menu in Build Analyzer	291
Figure B-12	Build Menu	292
Figure B-13	Filter Dialog Box	293
Figure B-14	Query Menu	293

List of Tables

Table 1-1	Fix and Continue Compile Time Cycle	8
Table 6-1	Valid C Operations	68
Table 6-2	Valid Fortran Operations	70
Table A-1	Fix and Continue Keyboard Accelerators	159

Introduction

This guide describes the Debugger. The Debugger is part of ProDev WorkShop, a suite of graphical, interactive, software engineering tools designed especially for programmers who develop and maintain C, C++, Fortran, and Ada libraries and applications.

What This Guide Contains

This guide contains the following chapters:

- Chapter 1, “Getting Started with the WorkShop Debugger” presents an overview of the ProDev Debugger, including how the Debugger is typically applied, a road map of the commands available from the Debugger Main View window, and a summary of the Debugger command line interface.
- Chapter 2, “Managing Source Files” presents the details for accessing source code files from the Debugger.
- Chapter 3, “A Short Debugger Tutorial” provides a tutorial to introduce you to the major features of the ProDev Debugger.
- Chapter 4, “Setting Traps” describes the facilities for setting stop traps and sample traps from the Debugger Main View, Source View, the Trap Manager, the Signal Panel, the Syscall Panel, and the Debugger command line interface.
- Chapter 5, “Controlling Process Execution” describes how to control process execution in a debugging session through the Debugger Main View control panel, the PC menu, Execution View, and the Debugger command line interface.
- Chapter 6, “Examining Debugger Data” presents reference information for the high-level Debugger views, which let you examine the Call Stack expressions, variables, arrays, and data structures. The chapter

also describes how you can access debugger data through the Debugger command line interface.

- Chapter 7, “Debugging with Fix+Continue: A Tutorial,” teaches you to perform the basic tasks that the Fix and Continue utility allows, such as making changes to functions and running the program with compiling or linking. Each task description is accompanied by a corresponding tutorial session.
- Chapter 8, “Detecting Heap Corruption” presents techniques for solving heap corruption problems and includes a tutorial.
- Chapter 9, “Multiple Process Debugging” describes how to use the Debugger Multiprocess View to debug programs with multiple processes.
- Chapter 10, “Using the X/Motif Analyzer: A Tutorial,” provides a tutorial to introduce you to the major features of the X/Motif analyzer.
- Appendix A, “Debugger Reference” contains a complete description of the Debugger’s graphical user and command-line interfaces.
- Appendix B, “Using the Build Manager,” describes the Build Manager, the tool for performing builds from ProDev. Two windows comprise the Build Manager:
 - Build View for watching compiles and correcting errors
 - Build Analyzer for viewing build dependency relations between files

What You Should Know Before Reading This Guide

This guide assumes that you’re familiar with C, C++, and object-oriented programming.

Related Information

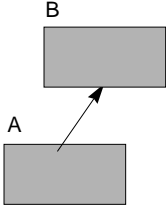
The Debugger is layered on the core ProDev WorkShop toolset (available from Silicon Graphics, Inc.). For further information about related tools, refer to the following documents:

- *Developer Magic Overview*, which provides an overview of the Developer Magic/ProDev toolset.
- *Developer Magic: Static Analyzer and Browser User's Guide*, which contains detailed information on how to use the static analyzer.
- *Developer Magic: Performance Analyzer and Tester User's Guide*, which contains detailed information on how to use the performance analyzer.
- *C++ Programmer's Guide*, which describes the Silicon Graphics C++ programming environment.
- *Ada Programmer's Guide*, which describes the Silicon Graphics C++ programming environment.
- *IRIS ViewKit User's Guide*, which describes how to create programs using IRIS ViewKit, a C++ toolkit that provides commonly needed facilities for applications based on the IRIS user interface toolkit.
- *MIPSpro Compiling, Debugging and Performance Tuning*, which discusses how to compile, debug, and tune the performance of programs written in the Silicon Graphics development environment (C, Fortran, and C++).

Conventions

Below are the typographical and graphic conventions used in this guide:

- **Bold**—Functions, option flags, and classes.
- *Italics*—Filenames, button names, field names, variables, emphasis, and IRIX commands.
- Regular—Menu and window names, data types, keywords, and text.

- “Quoted”—Menu choices.
- Fixed-width—Code examples and command syntax.
- **Bold fixed-width**—User input. Nonprinting <keys> are bracketed.
-  Graphic convention—Pull-down or popup menus.

Getting Started with the WorkShop Debugger

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”
- “Customizing the Debugger”

Typical Debugger Usage

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

- “Starting and Exiting the Debugger”
- “Using the Debugger From a Remote Host”
- “Using Main View”
- “Setting Traps”
- “Inspecting Debugger Data”
- “Changing Source Code”
- “Updating and Saving Views”
- “Integration With Other WorkShop Tools”

Starting and Exiting the Debugger

To start the Debugger, you 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 argument *executable* 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 later.

You can also invoke the Debugger and specify a core file (with its executable) to try to determine why a program crashed.

To exit the Debugger, select “Exit” from the Admin menu in the Main View window. You have two other options: you can type `quit` at the Debugger command line or press `<Ctrl-C>` where you first entered `cmd` into a terminal. You can also double-click the window system menu, or select its “Quit” entry.

Using the Debugger From a Remote Host

The `-host` option lets you specify a remote host on which the target executable will be run; the Debugger runs locally. This option is useful if

- you don’t want the Debugger windows to interfere with the application you are debugging
- you are supporting an application remotely
- you don’t want to use the Debugger on the target machine for some other reason

Using Main View

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

- 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 special debugging commands (see “Debugger Command Line” on page 271 for the syntax)

The major areas of the Main View window are shown in Figure 1-1.

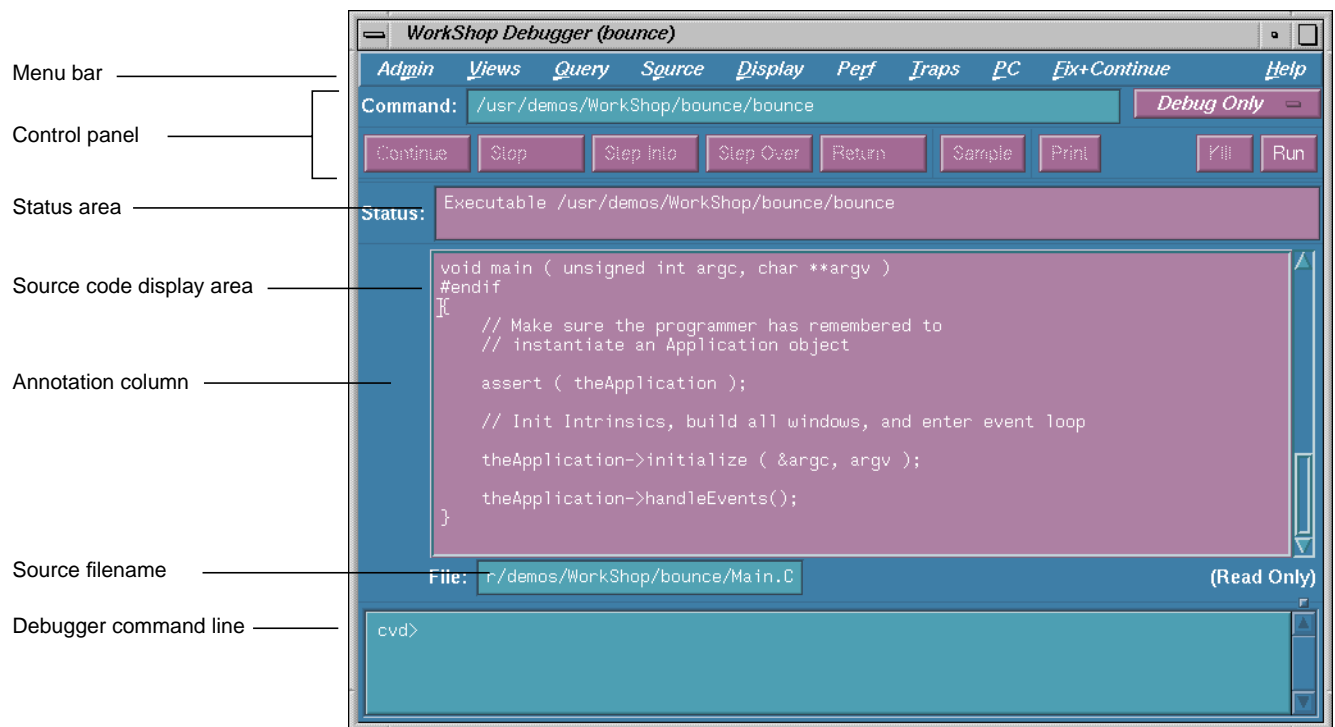


Figure 1-1 Major Areas of the Main View Window

Setting Traps

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

The Debugger lets you set traps:

- at a line in a file (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 (watchpoint)
- at set time intervals (pollpoint)

For more information on traps, refer to Chapter 4, “Setting Traps.”

Inspecting Debugger Data

When you stop the process, you then have a number of options for examining the data. You can inspect

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

For more information on the standard Debugger views, refer to Chapter 6, “Examining Debugger Data.”

Changing Source Code

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

1. Switch to a text editor (“Fork Editor” selection in Source menu) or edit in Main View (“Make Editable” in Source menu).

If you are using a configuration management system, then 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 “Save...”
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, “Using the Build Manager.”

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

Updating and Saving Views

Updating and saving view data is done through the Admin menu in the particular view (see Figure 1-2).

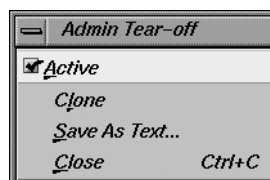


Figure 1-2 Admin Menu in Debugger Views

Typically, you display the WorkShop views of interest when you are stopped at a trap and leave them on the screen as you run the process.

- “Active” Updates the view each time the process stops or if the context is changed in the Call Stack View window. The active state is the default.
- “Clone” Makes a duplicate, inactive copy of the view, allowing you to save current information for future reference.
- “Save As Text...” Lets you save the data currently displayed in a view to a text file. Note that not all views have the “Save as Text...” option.
- “Close” Closes the window of the view.

Integration With Other WorkShop Tools

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

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. A performance task must be specified before the process is run; this enables the correct data collection. If you are in the middle of a run, you must terminate it, select a task, and restart the target to collect the data. Selecting “Performance Analyzer” from the “Launch Tool” submenu displays the main Performance Analyzer for analyzing the experiment results.

Accessing the Static Analyzer From Main View

You can access the Static Analyzer from the “Launch Tool” submenu, if desired. The Static Analyzer displays source information from Main View, making it easy to set traps in source code located by the Static Analyzer. Note that the Query menu enables you to conduct some of the same queries as in the Static Analyzer.

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.

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” in the Source menu, which provides selections for checking files in and out.

Recompiling From Main View or Source View

To access the Build View window (which lets you start the compile), in the Main View, pull down the Source menu and select “Recompile.” To examine the build dependencies, in the Main View, pull down the Admin menu and select “Build Analyzer.” For more information on the Build Manager tools, see Appendix B, “Using the Build Manager.”

Debugging with Fix+Continue

Fix and Continue is integrated with the Debugger. You issue Fix and Continue commands graphically from the Fix+Continue pulldown menu of the Debugger main window. You may also issue Fix and Continue commands from the Debugger command line (`cvd>`).

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. Fix and Continue enables you to speed up your development cycle significantly.

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

Table 1-1 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

Fix and Continue Functionality

Fix and Continue lets you:

- Redefine existing function definitions
- Disable, re-enable, 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 1-3.

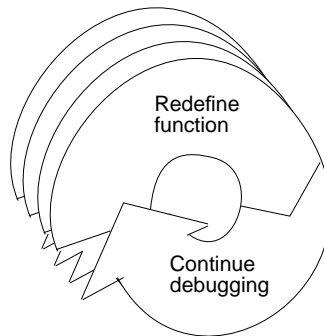


Figure 1-3 Fix and Continue Cycle

A typical session would be the following:

1. Using the 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 popup 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, re-enabling, 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 satisfied with the behavior of your application, you save the file, replacing the compiled source.

Fix and Continue/WorkShop Integration

Using Fix and Continue affects these WorkShop tools:

- The WorkShop Debugger Main View, the Source View, and the Fix and Continue Status window make a clear distinction between compiled and redefined code, and allow editing only in redefined code.

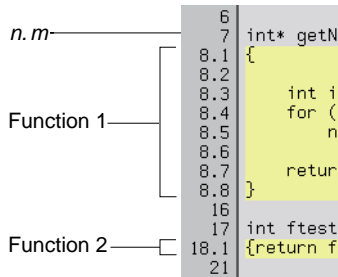


Figure 1-4 Line Numbers in Decimal Notation

- The different WorkShop views that are knowledgeable about redefined code:
 - Call Stack
 - Trap Manager
 - Debugger command-line

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 ($n.1, n.2, \dots, n.m$). 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 1.

Figure 1-4 shows two redefined functions. Function 1 replaces lines 8-15. Function 2 replaces lines 18-20. Although its three lines are now one, the following line number is still 21.

The Call Stack and Trap Manager 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. You can easily operate on several files at once.

Restrictions on Fix and Continue

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

- You may not add, delete, or reorder the 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**.
- Both the old and new functions must be compiled with -g.

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 it is called.

- If you redefine functions which 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 re-run.

For example, consider the following call stack:

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

1. If you redefine **foo()**, you can continue execution provided the layout of the stack frames are same.
2. 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()**].
3. If you redefine **main()** after you have run, it will be executed only when you re-run.

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 completely 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 “Fix+Continue Windows” on page 258.

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—see Figure A-1) remains Read-Only. Fix and Continue edits are saved in an intermediate state and must be explicitly written with the “Save File+Fixes As...” option to be saved.

When you edit a function, it is color-highlighted. Then, if you switch to the compiled version, the color changes to show that there is a redefinition. If you try to edit the compiled version, the Debugger beeps, indicating it is Read-Only.

When you have completed your edits and wish to see the results, click “Parse and Load.” When the Parse and Load has executed successfully, the color changes again. If the color doesn’t change, there may be errors, and you should check your “Message Window...” view.

GUI Debugger Command Line

Just as you can enter any *dbx* command at the Debugger command line, you can enter Fix and Continue commands there.

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 is redefined, enabled, disabled, deleted, or detached.
- Fix and Continue needs to know where to find include files and other parameters specified by compiler build flags (compiler options). 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 pulldown 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.

A successful run results in output in the Execution View. This functionality is the same as it is in the Debugger without Fix and Continue.

Debugging with the X/Motif Analyzer

The X/Motif analyzer is integrated with the Debugger. You issue X/Motif analyzer commands graphically from the X/Motif analyzer subwindow of the main Debugger window (see Figure 1-5). To access the subwindow, you must pull down the Views menu and select “X/Motif Analyzer”

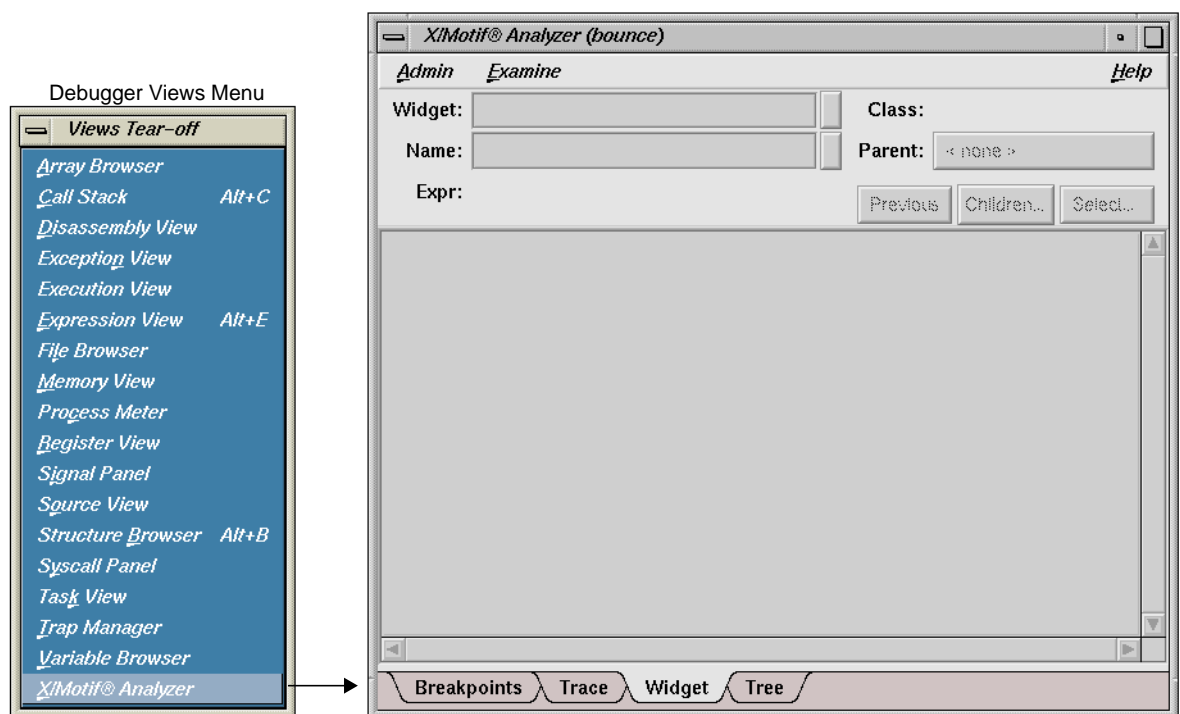


Figure 1-5 Launching the X/Motif Analyzer

Special Libraries

When you first bring up the X/Motif Analyzer, it may ask you if you want to change `$LD_LIBRARY_PATH` 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 OK to the dialog and use these libraries. Doing so enables all of the features of the X/Motif Analyzer. These libraries are identical in functionality to the libraries shipped with IRIX 5.3. The analyzer uses the Silicon Graphics enhanced version of these libraries. There are no instrumented MIPS/ABI versions of the libraries.

Using the X/Motif Analyzer

The X/Motif Analyzer provides specific debugging support for X/Motif applications. There are various examiners for different X/Motif objects (for example, widgets and X graphics contexts) that are normally difficult or impossible to inspect using ordinary debugger functionality. Also, you can set widget-level breakpoints and collect X event history information (in the same manner as *xscope*).

Examiners Overview

When you first bring up the X/Motif Analyzer, you see the Widget examiner. It may be blank or displaying a Widget that it found on the callstack if you have a stopped process. Most of the examiners in the X/Motif Analyzer try to detect interesting objects from the callstack and offer to display them for you, automatically.

At the bottom of the X/Motif Analyzer is a tab panel showing the current set of examiners. Besides the Widget examiner, the Breakpoints, Tree, and Trace examiners are available by default. These four tabs are always present.

To bring up new examiners, use the Examine menu and select one from below the separator. Some examiners (for example, the Callback examiner) cannot be manually selected—they appear only when the callstack context

is appropriate. In the case of the Callback examiner, it appears only when the process is stopped somewhere in a widget callback.

To remove an examiner from the tab panel, put the pointer over the tab, click the right button of your mouse, and select 'Remove Examiner' from the popup menu. The tab disappears.

Examiners and Selections

If you select text in one examiner and then choose another 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 evaluated to 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 the selected text. If the type of the text is unknown, the error `Couldn't examine selection in more detail` is generated. Otherwise, the appropriate examiner is chosen and the text is evaluated.

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

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 provides inspection capability for you to see into the data structures behind the ID values.

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 (at a lower level) for specific X events or requests.

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.

Restrictions and Limitations

Due to implementation details, there are several nuisances that currently pose some restrictions to the X/Motif Analyzer:

- The Breakpoints area is active only after you've stopped the process once, and if you've 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*-type requests (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 *cod* 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.

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 with 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 that you use frequently but apply only at specific times in a debugging session

Using a Startup File

The startup file feature lets you preload your favorite buttons and aliases in a file that runs when *cvd* is invoked. It's also useful if you have traps that you set the same way each time. The suggested name for the startup file is *.cvdrc*; you can supply a different name as long as you specify its path in the environment variable *CVDINIT*. The steps in the algorithm that *cvd* follows when looking for a startup file are:

1. Check the environment variable *CVDINIT*.
2. Check for the file *.cvdrc* in the current directory.
3. Check for the file *.cvdrc* in the user's home directory.

Implementing User-Defined Buttons

If there are Debugger commands or combinations of Debugger commands that you use frequently, you may find it convenient to define a button for the graphical interface. 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 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 1-6 is a typical example of Main View with user-defined buttons. The definitions for the user-defined buttons display in the Debugger command line area.

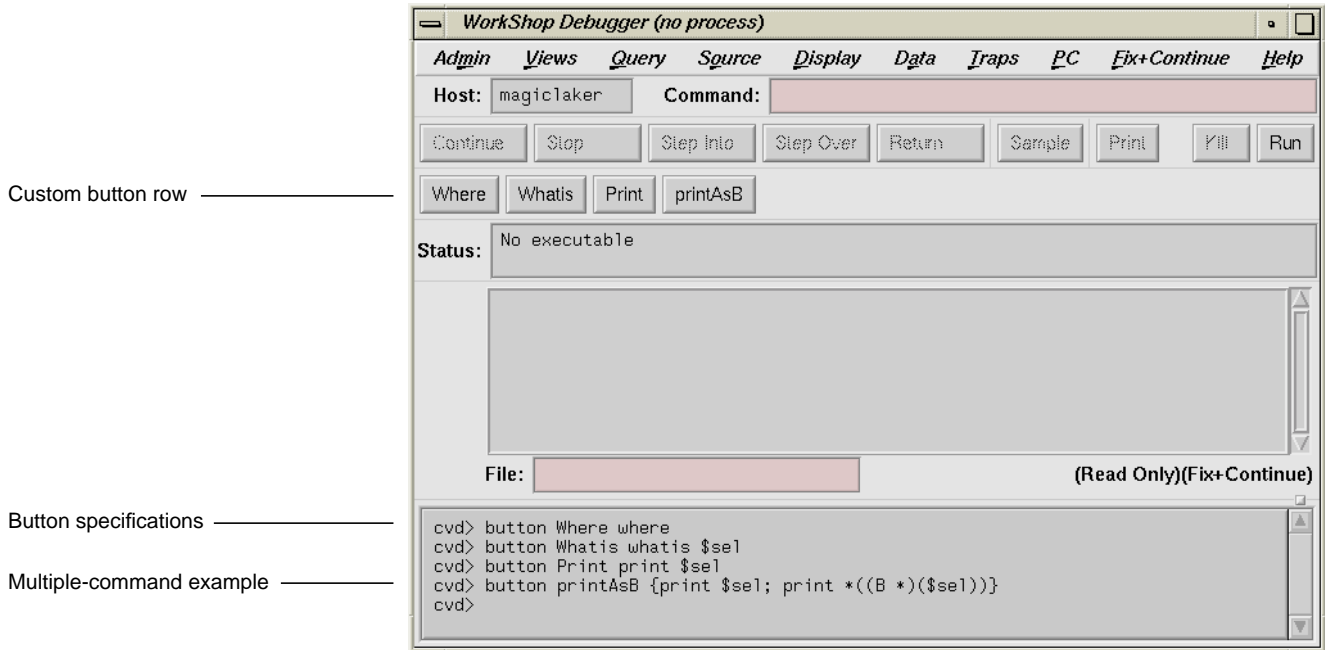


Figure 1-6 User-Defined Button Example

The syntax for creating a button is

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

The syntax for creating a multiple-command button is

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

where

button is the Debugger command for defining buttons. It can be applied at the Debugger command line or in a startup file.

label is the name appearing on the button. 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 is one of the the Debugger commands, which are entered at the command line at the bottom of Main View. See “Debugger Command Line” on page 271.

\$sel is a substitute for the current cursor selection and should be appropriate as an argument to the selected command.

command1, *command2* (and any additional commands) are Debugger commands to be applied in order. They 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 syntax

button

displays a list of all currently defined buttons.

The syntax

unbutton *label*

deletes the button corresponding to the label. You might use this if you needed room to implement different buttons. The effect of **unbutton** is temporary so that subsequently running the startup file reactivates the button.

The syntax

button *label*

displays the button’s definition if it exists. If the button does not exist, an error message displays along with the standard usage syntax for **button**.

Managing Source Files

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

- “Accessing Files Used by an Executable”
- “Opening a New File”
- “Path Remapping”

Accessing Files Used by an Executable

The File Browser, available from the Views menu in the Main View, provides a scrollable list of the source files used by your executable, including files in linked libraries. See Figure 2-1.

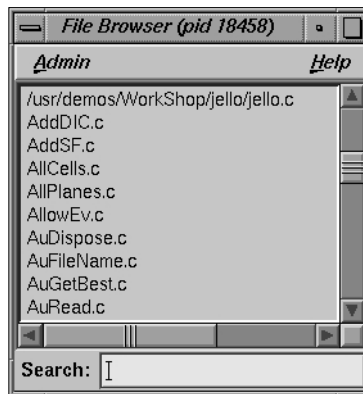


Figure 2-1 File Browser Window

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 the filename.

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 in Figure 2-2 displays, listing the files in the file list display area and the currently selected directory in the selection field.

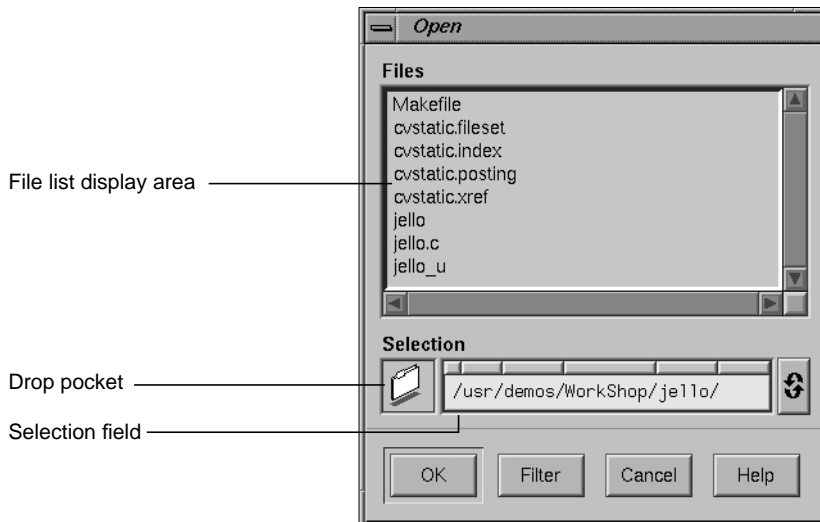


Figure 2-2 Open Source File Dialog Box

To load a file listed in the file list, double-click the file’s name. You can also type the full pathname of the file in the *Selection* field and click the *OK* button to load the file. Another alternative is to 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. The files in the new directory are listed in the file list.

If you specify a filename without a full path, the Debugger will use the current path remapping information to attempt to locate the file.

Another method for opening a file in Main View is to enter its full name in the *File* field, below the source code display area, and press <Enter>.

Path Remapping

The path remapping option allows you to modify the set of mappings used to redirect filenames located in your executable to their actual locations in your file system. Since WorkShop uses full (absolute) pathnames, path remapping is generally 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 using the *use* and *dir* 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 2-3).

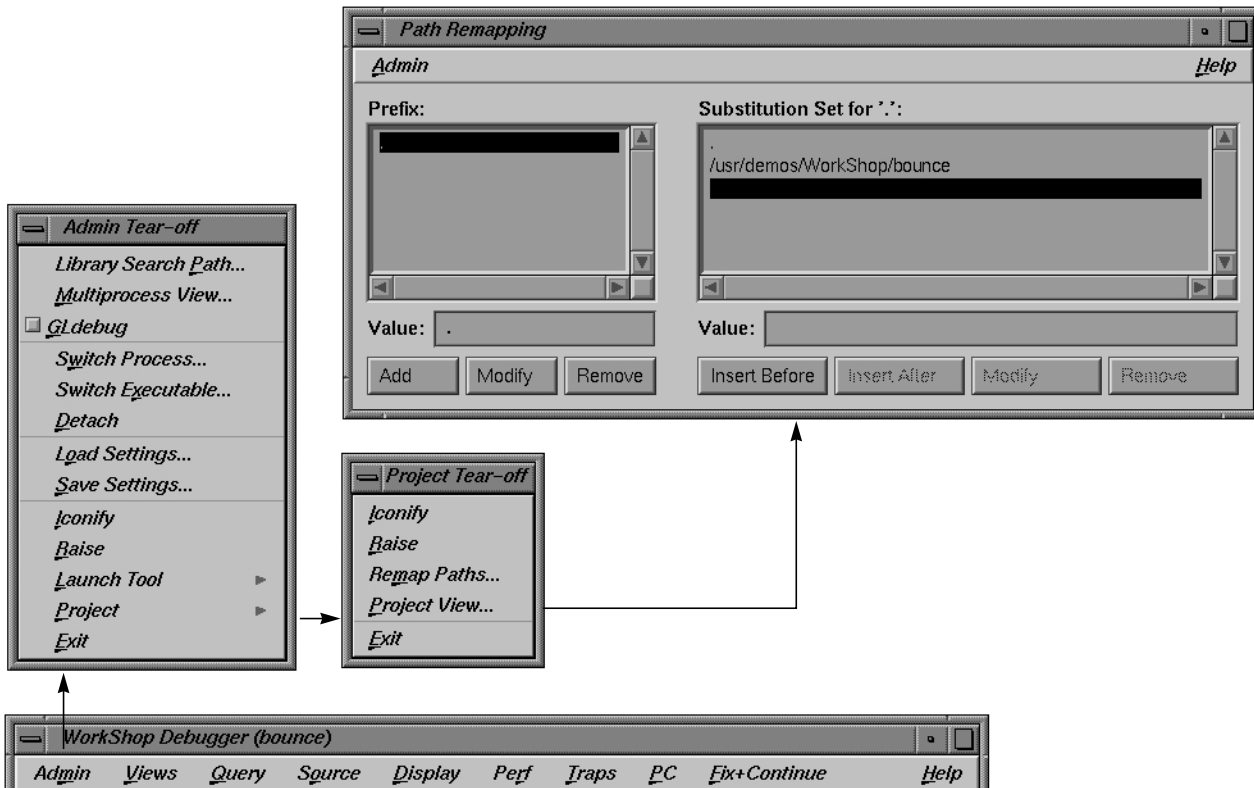


Figure 2-3 Path Remapping Dialog Box

For each prefix listed in the *Prefix* list, there is an ordered set of substitutions that are used to find a real file. By default, the 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. Here are some operations you can perform:

- 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

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”
- “Performing a Search”
- “Setting Traps”
- “Examining Data”

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.

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. Go to the directory `/usr/demos/WorkShop/jello`.
2. If the *jello* executable does not yet exist, type `make jello`
3. To invoke the Debugger, type `cvd jello`

The Main View window appears as shown in Figure 3-1. The display scrolls automatically to the `main` function.

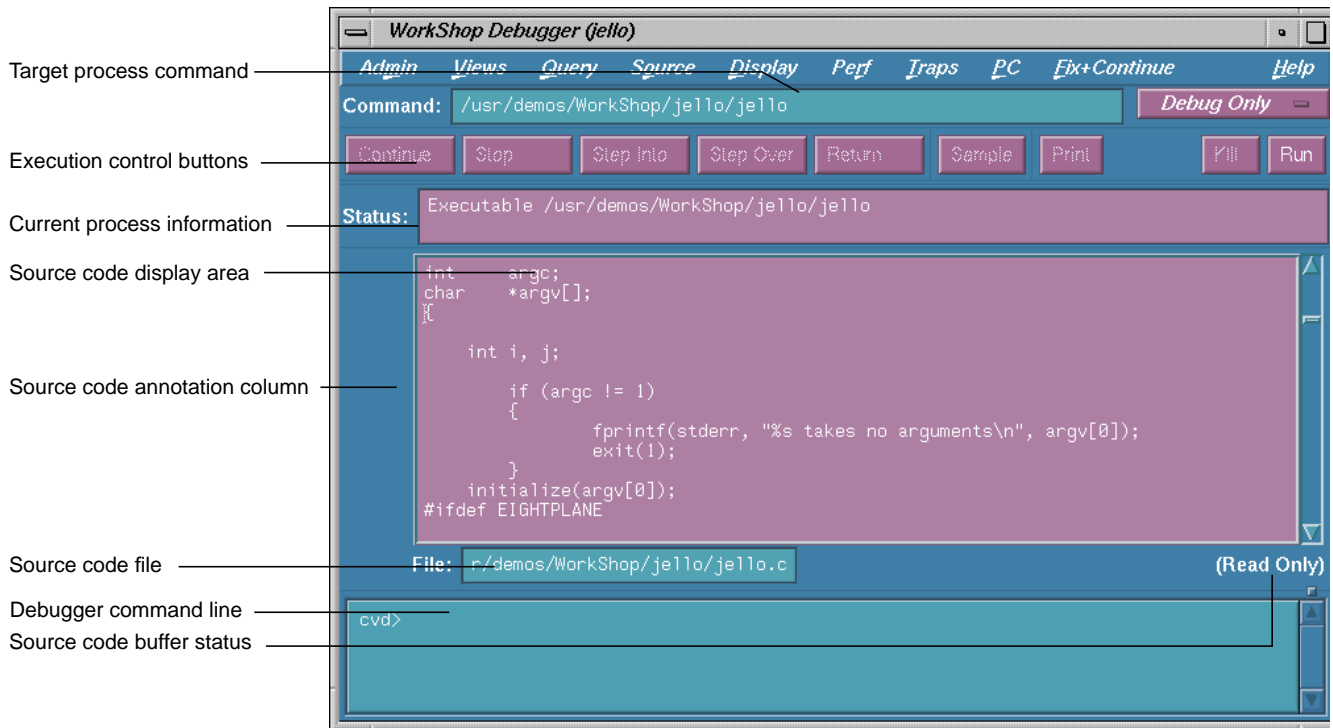


Figure 3-1 The Main View Window with *jello* Source Code

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).

4. 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 3-2). Enlarge this window to watch the program execute. The polyhedron is initially suspended in the center of the cube.

5. Click the left mouse button anywhere inside the *jello* window.
The polyhedron drops to the floor of the cube.

6. 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: 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 a normal side effect due to GL/X interaction.

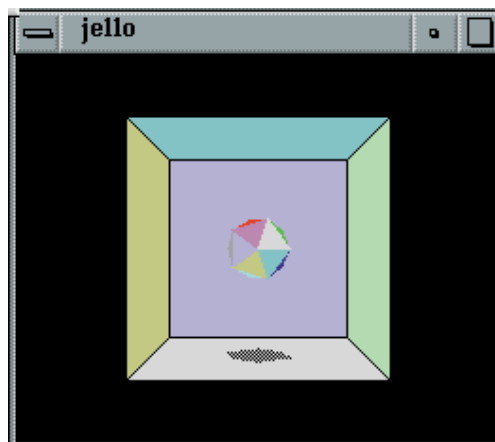


Figure 3-2 The *jello* Window

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 Search dialog box, as shown in Figure 3-3.

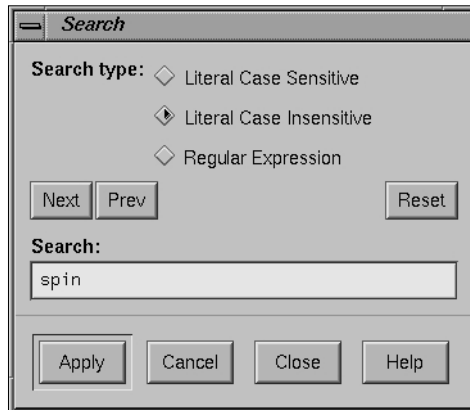


Figure 3-3 The Search Dialog Box

3. Click the *Apply* button in the Search dialog box.

The search takes place. Each instance of “spin,” the target string, is highlighted in the source code and flagged in the scroll bar to the right of the display area. Figure 3-4 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 “Source Menu” on page 145.

4. Click the *Close* button in the Search dialog box.

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.

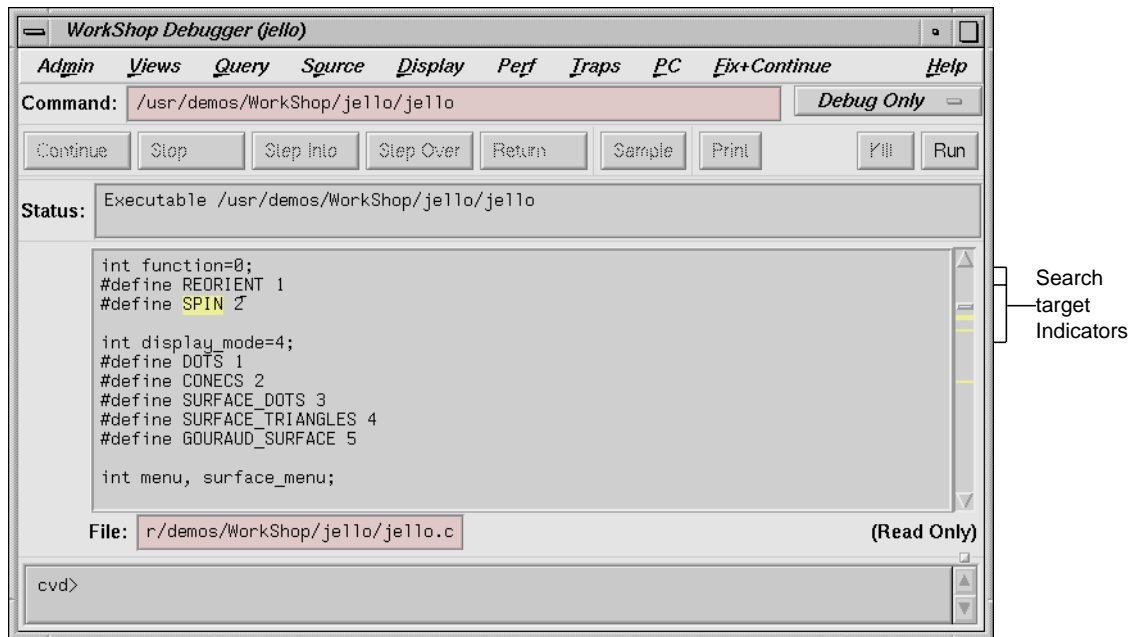


Figure 3-4 Search Target Indicators

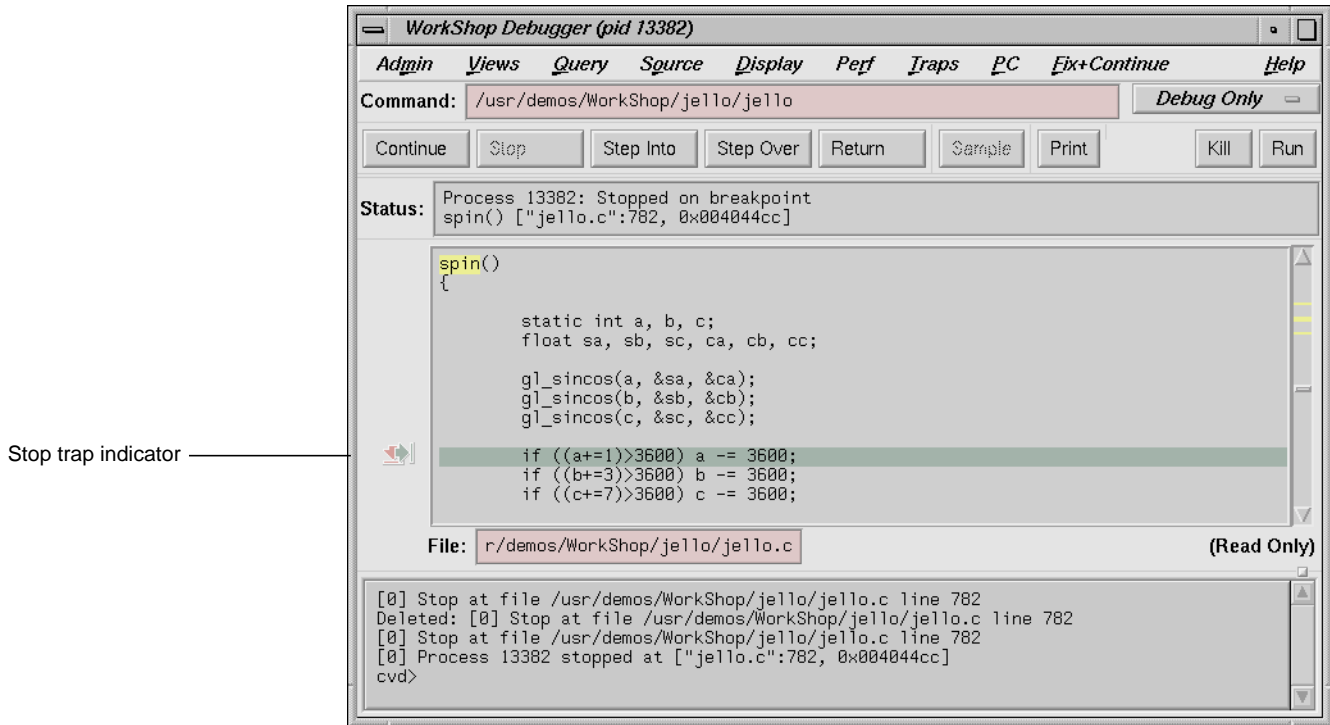
Setting Traps

Stop traps (also called breakpoints) stop the program's 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, "Setting Traps."

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 3-5. 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 PC appears and the line becomes highlighted.



Stop trap indicator

Figure 3-5 Stop Trap Indicator

2. Click the *Continue* button at the upper-left corner of the Main View window repeatedly 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 3-6.

Trap Manager lets you list, add, edit, disable, or remove traps in a process. You set one breakpoint in the **spin** function by clicking in the source code annotation column. This 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 from the top. The count information lets you specify the number of times a trap should be encountered before it fires. The trap controls let you manipulate traps. All traps (active and inactive) are shown in the trap display area.

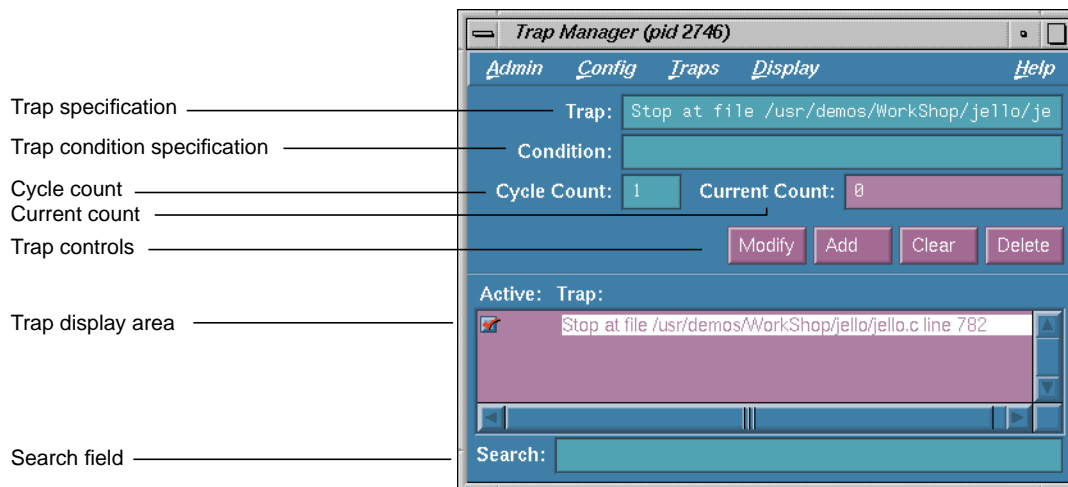


Figure 3-6 The Trap Manager Window

- 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.
- Click the *Clear* button, move the cursor to the *Trap:* field, 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 “display” and a different display option. 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 popup menu, and select “display” and then the “conecs” option with the right button.

This triggers the watchpoint and stops the process. If you were tracking the effects of changing display modes, you could bring up other views now.

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.

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 3-7. 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 “Tracing Through Call Stack View” on page 61.

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 examine 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.

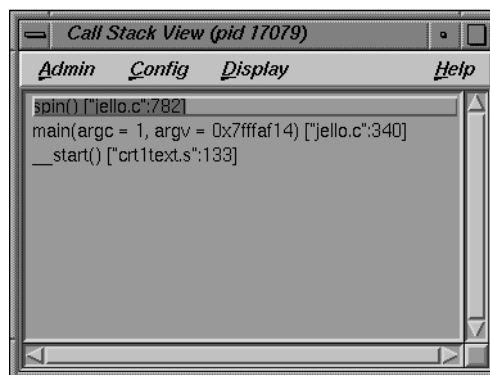


Figure 3-7 Call Stack View at **spin** Stop Trap

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.

5. 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 3-8, 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.

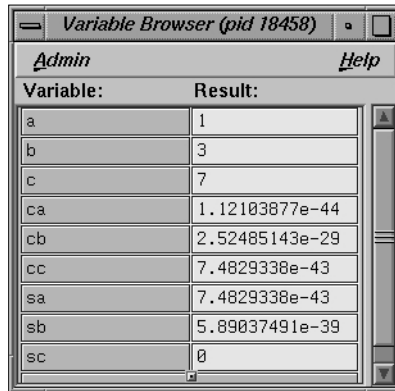


Figure 3-8 Variable Browser at **spin**

- 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 3-9). 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.

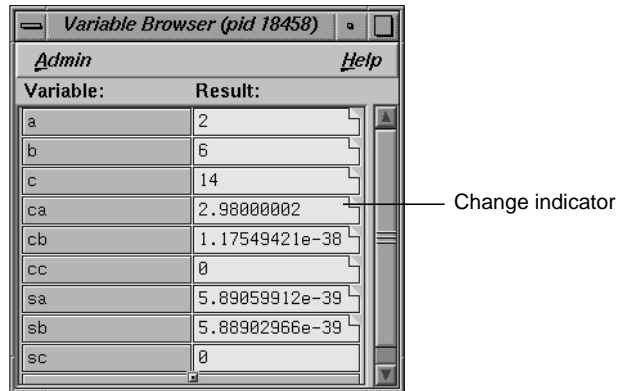


Figure 3-9 Variable Browser after Changes

7. Select "Close" from the Admin menu in Variable Browser and "Close" from the Admin menu in Call Stack View to close them.
8. 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 "Evaluating Expressions" on page 64.

9. 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 3-10) lets you apply the language semantics to the expression.

The Format menu (shown on the right side of Figure 3-10) lets you view the value, type, address, or size of the result. You can further specify the display format for the value and address.

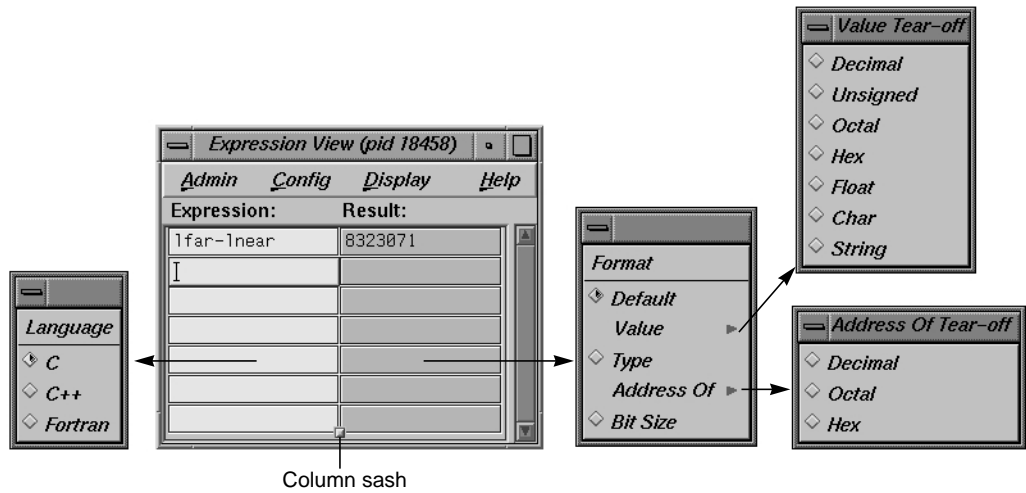


Figure 3-10 Expression View With Language and Format Menus Displayed

10. 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.

11. Select “Close” from the Admin menu in Expression View to close it.
12. Select “Structure Browser” from the Views menu in Main View.
13. Enter `jello_conec` in the *Expression* field and press **<Enter>**.

The Structure Browser 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, 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 3-11 now appears.

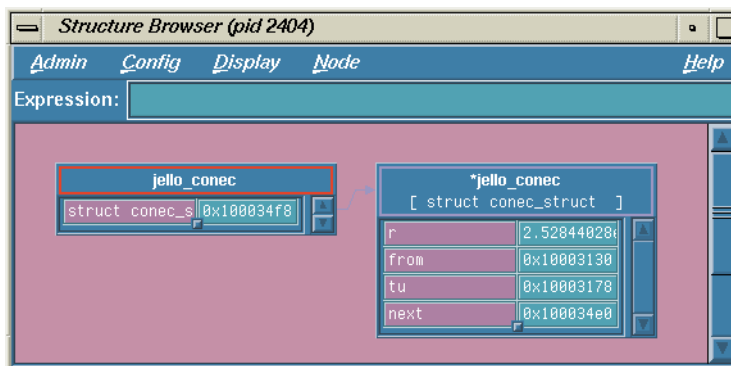


Figure 3-11 Structure Browser Window With *jello_conec* Structure

14. 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 3-12.)

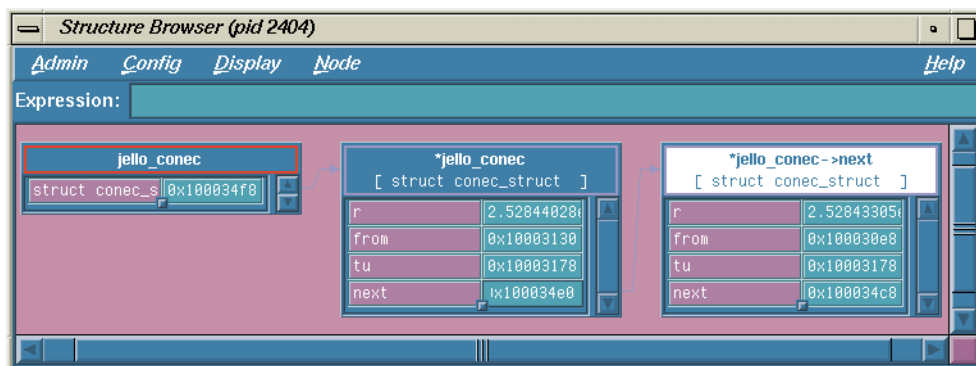


Figure 3-12 Structure Browser Window With Next Pointer Dereferenced

15. Select "Close" from the Admin menu in Structure Browser to close it.
16. 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.

17. 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 3-13 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.

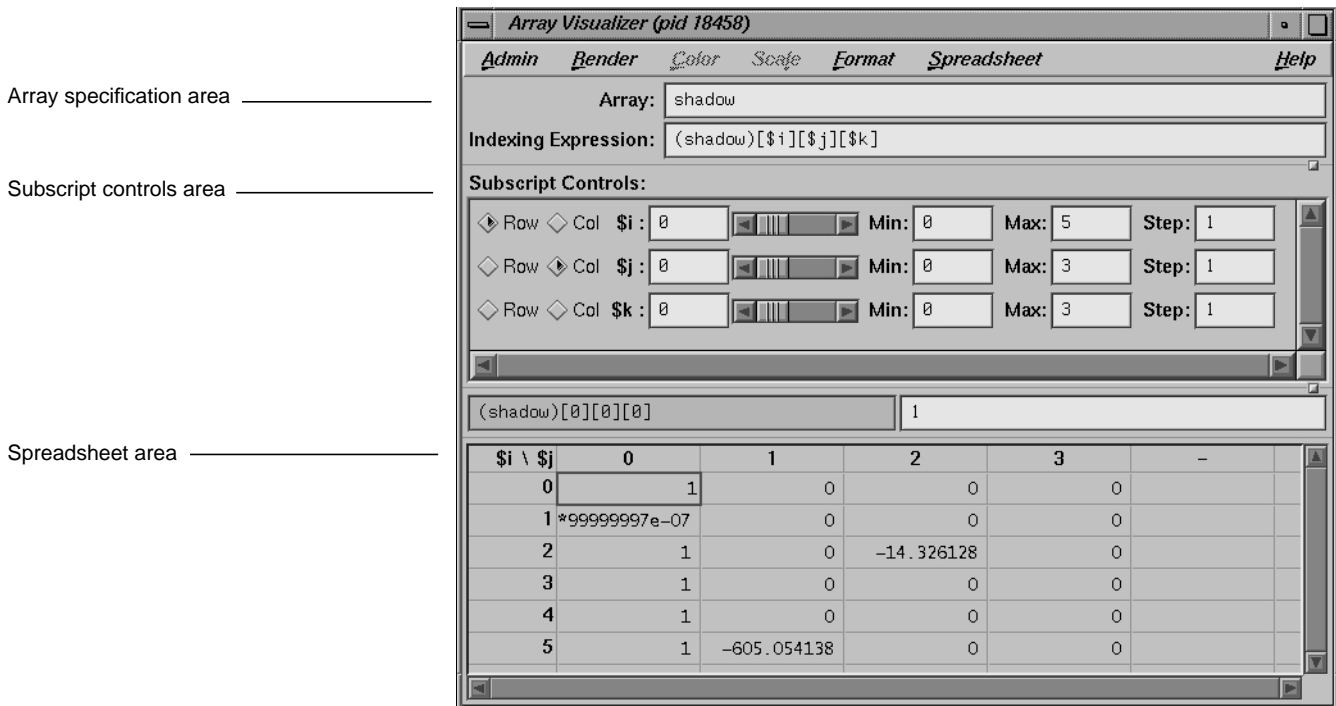


Figure 3-13 Array Browser Window for shadow Matrix

18. 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 $\$i$ by $\$k$.

Figure 3-14 shows a close-up view of the subscript control area.

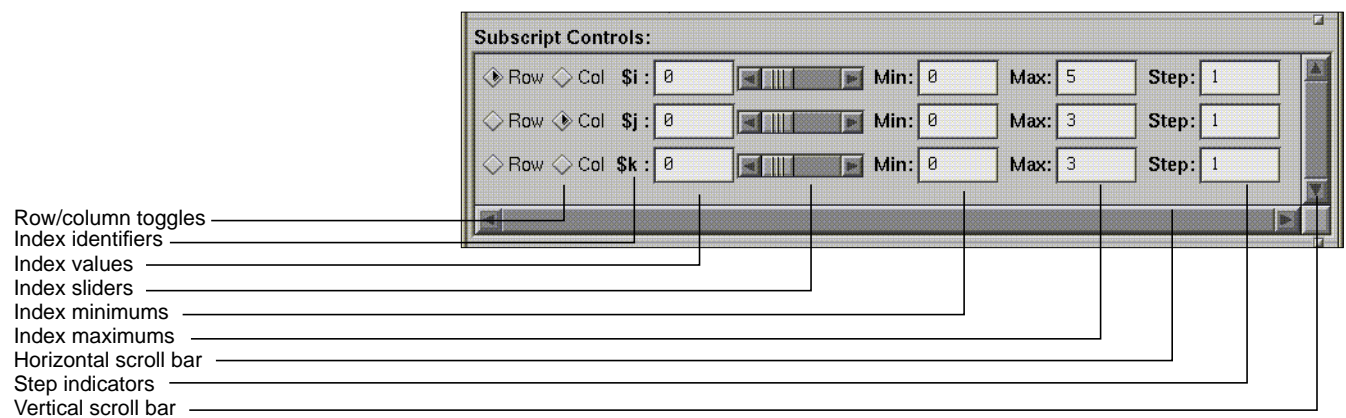


Figure 3-14 Subscript Control Area in Array Browser

The row/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 Browser window.

19. 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; the corresponding value is plotted on the z -axis. The data can be rendered as a surface, bar chart, multiple lines, or points.

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

Setting Traps

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 (Source View) or the Trap Manager. You can also specify traps in the Debugger command line at the bottom of the Main View window. 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 a program's performance. They let you collect data related to resource usage without stopping the process.

This chapter covers the following topics:

- "Trap Terminology"
- "Setting Traps in Main View and Source View"
- "Setting Traps in Trap Manager"
- "Setting Traps With Signal Panel and Syscall Panel"

For a tutorial on the use of traps, see "Setting Traps" on page 31.

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 taken when the trap fires, either stopping the process or capturing data.

When used as a verb, the term trigger refers to engaging a trap but not necessarily firing it. There are some circumstances where the process may hit a trap but not satisfy all the conditions necessary for firing it.

Trap Triggers

You can set traps at a specified location or when a specified event occurs. The triggers provided in WorkShop are

- at a given line in a file (traditionally referred to as “breakpoints”)
- at a given instruction address
- at the entry or exit for a given function
- after set time intervals (referred to as “pollpoints”)
- when a given variable or address is read, written, or executed (referred to as “watchpoints”)
- when a given signal is received
- when a given system call is entered or exited

Furthermore, you can specify a condition (as an expression) that must be met before a trap fires. You can also specify the cycle count, that is, a specific number of passes through a trap without firing it.

Trap Actions

Two actions can occur when a trap is fired:

- one or all processes can be stopped
- a sample of performance data can be taken

A trap that stops processes is called a stop trap, or a breakpoint. A trap that collects performance data is called a sample 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 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.

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.

Setting Traps With the Traps Menu in Main View

The Traps Menu in Main View is shown in Figure 4-1.

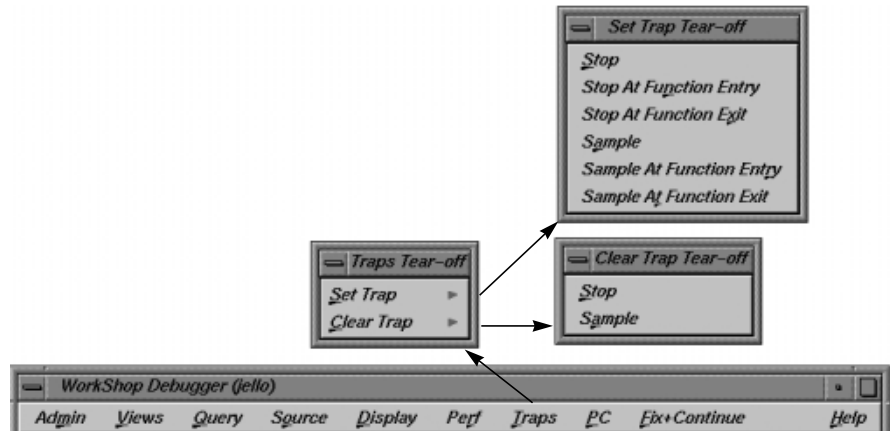


Figure 4-1 Traps Menu in Main View

To set a trap using the Traps menu, you need to identify where the trap is to be located and its 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 “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.

The traps are indicated by icons in the source annotation column (and also appear in Trap Manager if you have it open). Figure 4-2 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 “Enabling and Disabling Traps.”

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.

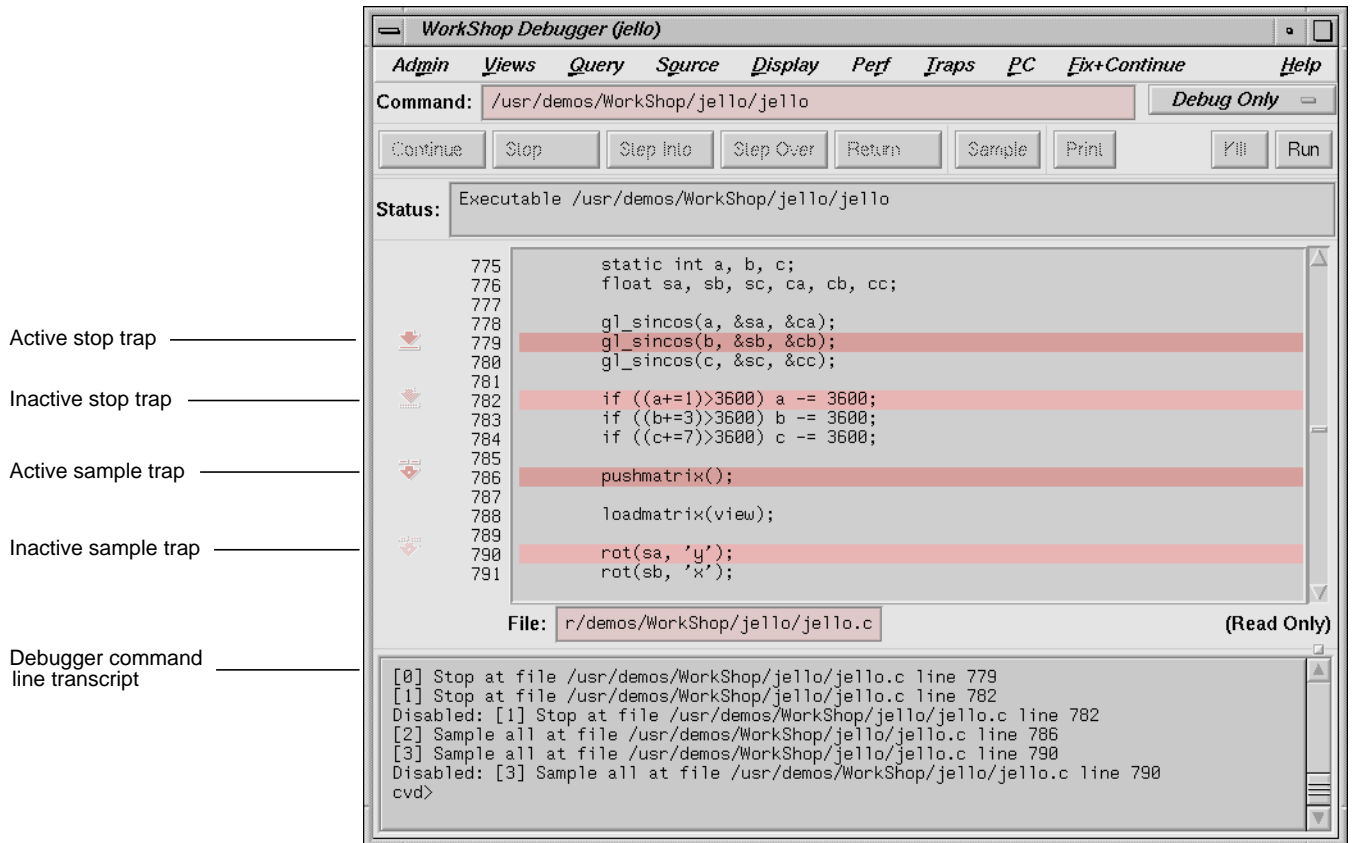


Figure 4-2 Typical Trap Icons

Setting Traps With the Mouse

The quickest way to set a trap is to click in the source annotation column in Main View and 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. (The way to tell if data collection is on is to look at the Debug option menu in the upper-right corner of the Debugger main view, and see whether it is set to "Debug Only," "Performance," or "Purify.")

Setting Traps in Trap Manager

The Trap Manager helps you manage all the traps involved with a process. Its 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 4-3 with the Config, Traps, and Display menus displayed.

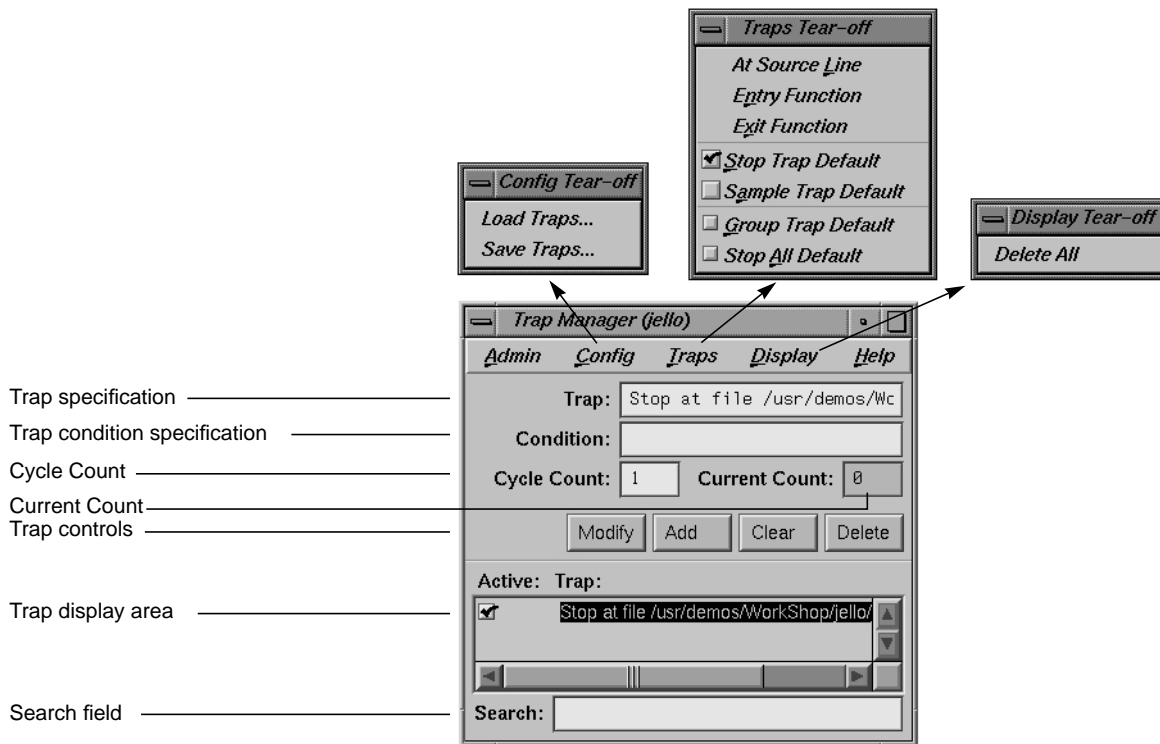


Figure 4-3 Config, Traps, and Display Menus in the Trap Manager

Setting Single-process and Multiprocess Traps in the Trap: Field

New or modified traps are entered in the *Trap:* field. They have the general form:

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

The entry `[stop | sample]` refers to the trap action. You can set a default for it in the Traps menu (“Stop Trap Default” or “Sample Trap Default”) and omit entering it at the beginning of the specification.

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. These will be entered by default if the “Stop All Default” and “Group Trap Default” toggles in the Traps menu are on.

The specific syntaxes for the location and condition are shown below. After you enter the trap (by the *Add* or *Modify* button or by `<Enter>`), the full syntax of the specification appears in the field. The *Clear* button clears the *Trap* and *Condition* fields and the cycle fields.

```
[stop | sample] [all] [pgrp] at [file filename] \  
    [line line-number]  
    Trap at the specified line in the specified file.
```

```
[stop | sample] [all] [pgrp] addr instruction-address  
    Trap on the specified instruction address.
```

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

```
[stop | sample] [all] [pgrp] in function [[file] \  
    filename]  
    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]  
    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]]  
    Set a data 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]]`
Set 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`
Trap on receipt of the given signal. Same as “catch” in *dbx*.

`[stop | sample] [all] [pgrp] syscall entry sys-call-name`
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`
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]`
Trap at regular intervals of *time* seconds. This is typically used for sampling only.

Some typical trap examples are provided in Figure 4-4. 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.

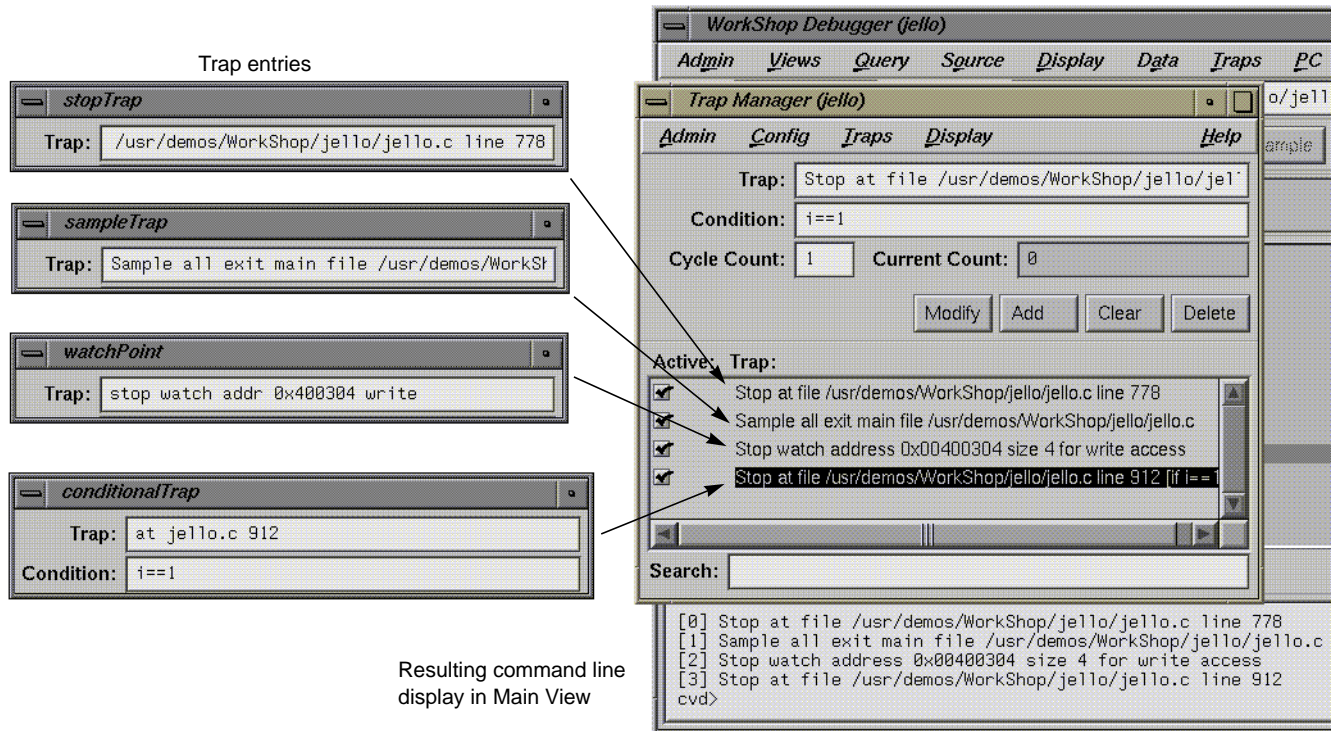


Figure 4-4 Trap Examples

Setting a Trap Condition

The *Condition*: field lets you specify a condition necessary for the trap to be fired. A condition can be any legal expression and will 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 4-4. 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, the 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.

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 at the *n*th time the trap is encountered and every *n* 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.

Setting a Trap With the Traps Menu and Source Display

The Traps menu in Trap Manager (see Figure 4-5) 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, you select the function name in the source display and then click “Entry Function” or “Exit Function.” The trap set in all of these cases is governed by the defaults you have set in the menu.

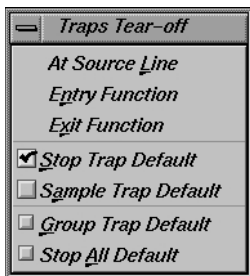


Figure 4-5
Traps Menu in Trap Manager

Moving around the Trap Display Area

The trap display area shows all traps set for the current process. You have vertical and horizontal scroll bars for moving around the display area. The *Search* field lets you incrementally search for any string in any trap.

Enabling and Disabling Traps

Each trap has an indicator to its left for toggling back and forth between its active and inactive states. This feature lets you accumulate traps and turn them on only as needed. Thus, when you don't need the trap, it won't get in your way. When you do need it, it is readily re-enabled.

Saving and Reusing Trap Sets

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

Setting Traps With Signal Panel and Syscall Panel

You can trap signals using Signal Panel and system calls using Syscall Panel (see Figure 4-6).

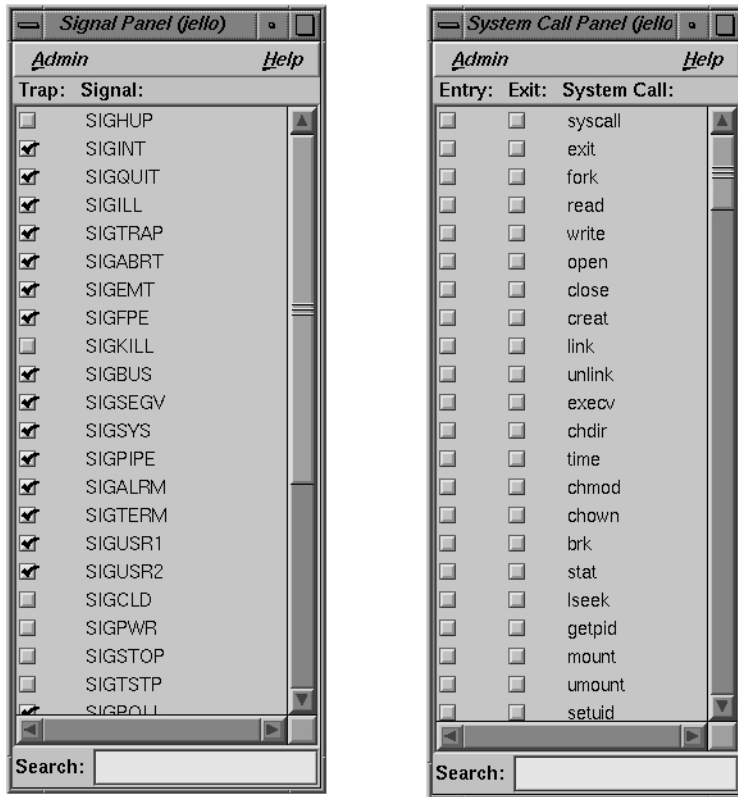


Figure 4-6 Signal Panel and Syscall Panel

Both are selected from the Views menu in Main View. Signal Panel sets a trap on receipt of the signal(s) selected. Syscall Panel sets a trap at the selected entry to or return from the system call.

Controlling Process Execution

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

- “Main View Control Panel”
- “Controlling Process Execution With PC Menu”
- “Execution View”

Main View Control Panel

Process execution is controlled using the top portion of the Main View window. See Figure 5-1.

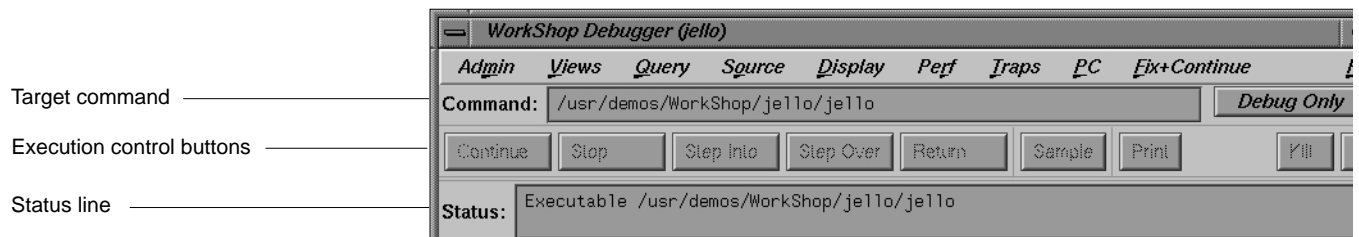


Figure 5-1 Main View Control Panel

The Main View window contains a row of execution control buttons that enable you to control program execution. The execution control buttons are located above the source display area. To use any of these commands, click on the appropriate button with the left mouse button. The Main View control panel is described below.

Status and Entry Fields in the Main View Control Panel

The 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.)

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 the program.
- Kill* Kills the active process.

The control buttons used for process interruptions are

- 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 popup menu shown in Figure 5-2. 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 5-2.

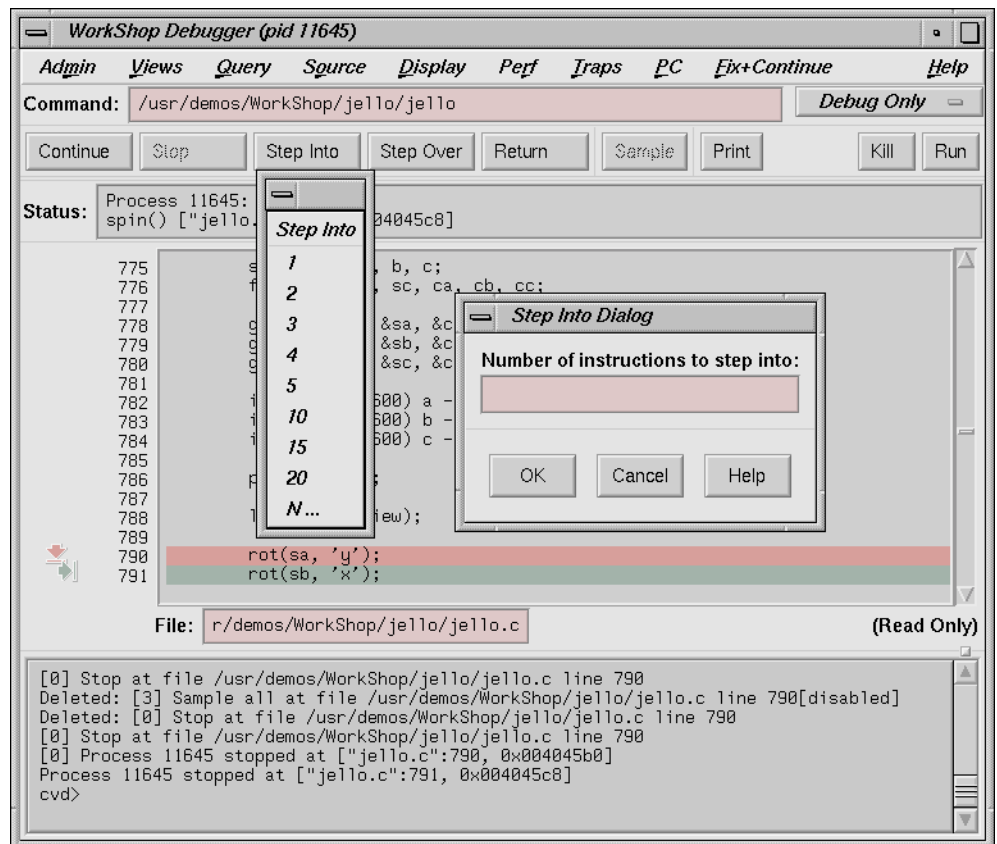


Figure 5-2 Step Into Popup Menu and Dialog Box

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 popup menu shown in Figure 5-3. 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 5-3.

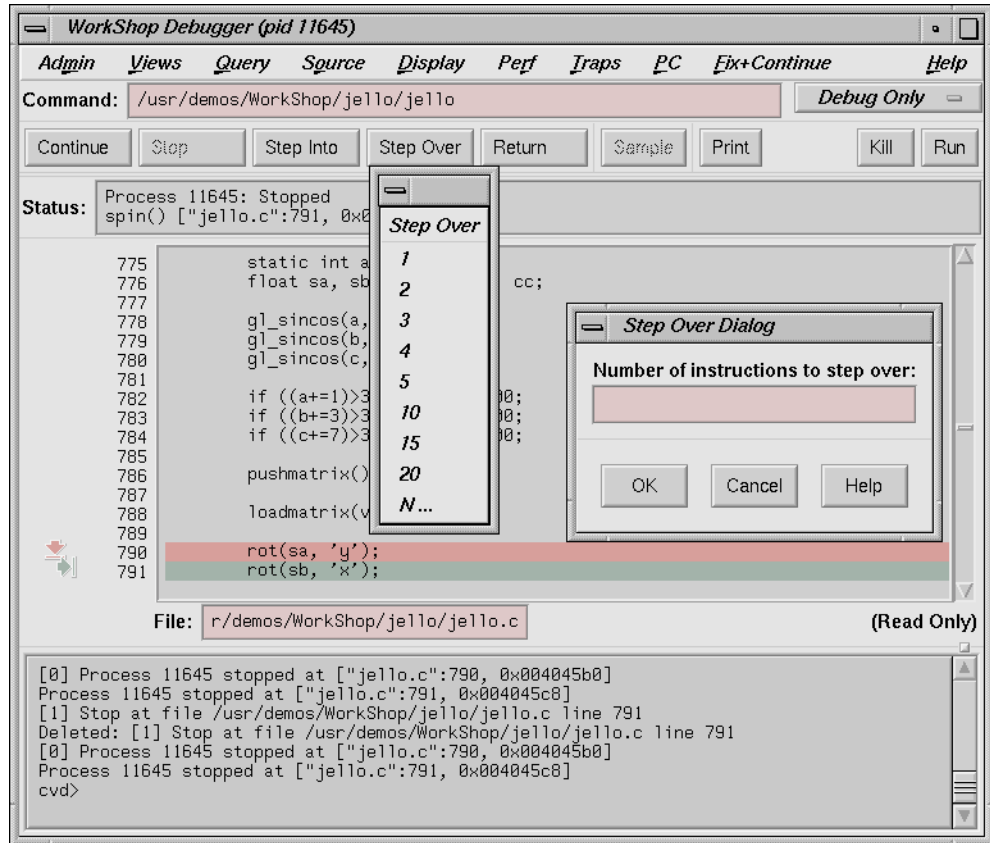


Figure 5-3 Step Over Popup Menu and Dialog Box

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.

Controlling Process Execution With PC Menu

The PC (program counter) menu in Main View provides a quick and informal means of controlling process execution. See Figure 5-4.

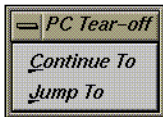


Figure 5-4
PC Menu in Main View

These 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 re-execute a portion of code.

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 is launched in iconified form.

Examining Debugger Data

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

- “Tracing Through Call Stack View”
- “Evaluating Expressions”

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, “Debugger Reference.”

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 6-1 with the major menus displayed.

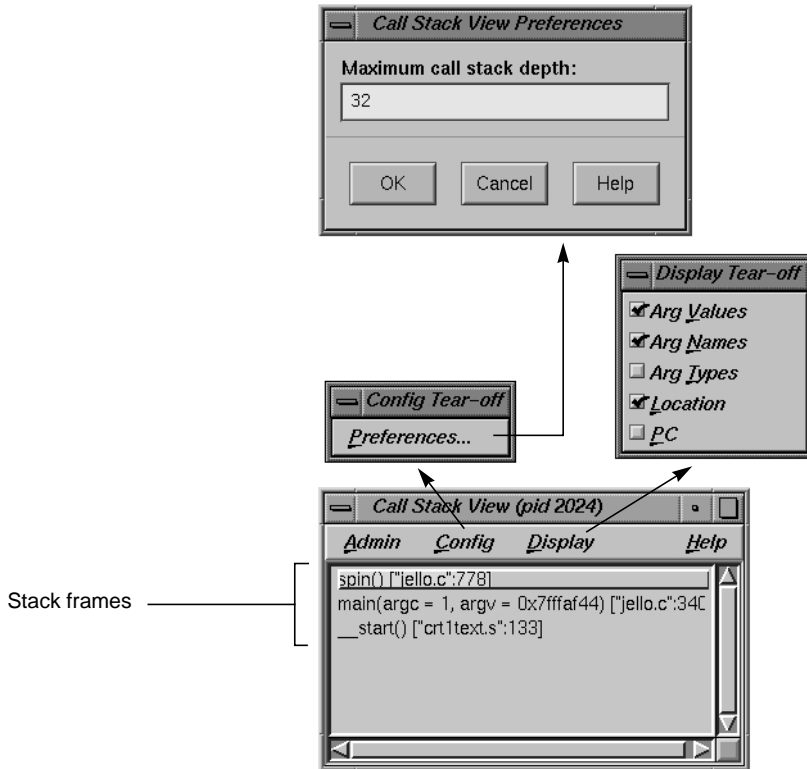


Figure 6-1 Call Stack View Window With Config and Display Menus and Preferences Dialog Box

In addition to the functions, the Call Stack View window lets you see the argument names, values, and types as well as the locations of the functions and the PC. If symbolic information for the arguments has been stripped from the executable, the label `<stripped>` will appear in place of the arguments, as in the function `_start` in the above example. You also have the option of setting the maximum depth of the Call Stack View by selecting “Preferences...” from the Config menu.

To move through the call stack, you simply 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 (such as the

Variable Browser or Structure Browser) will 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 6-2 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, then the window would have scrolled to the place where the function **main** calls **spin**.

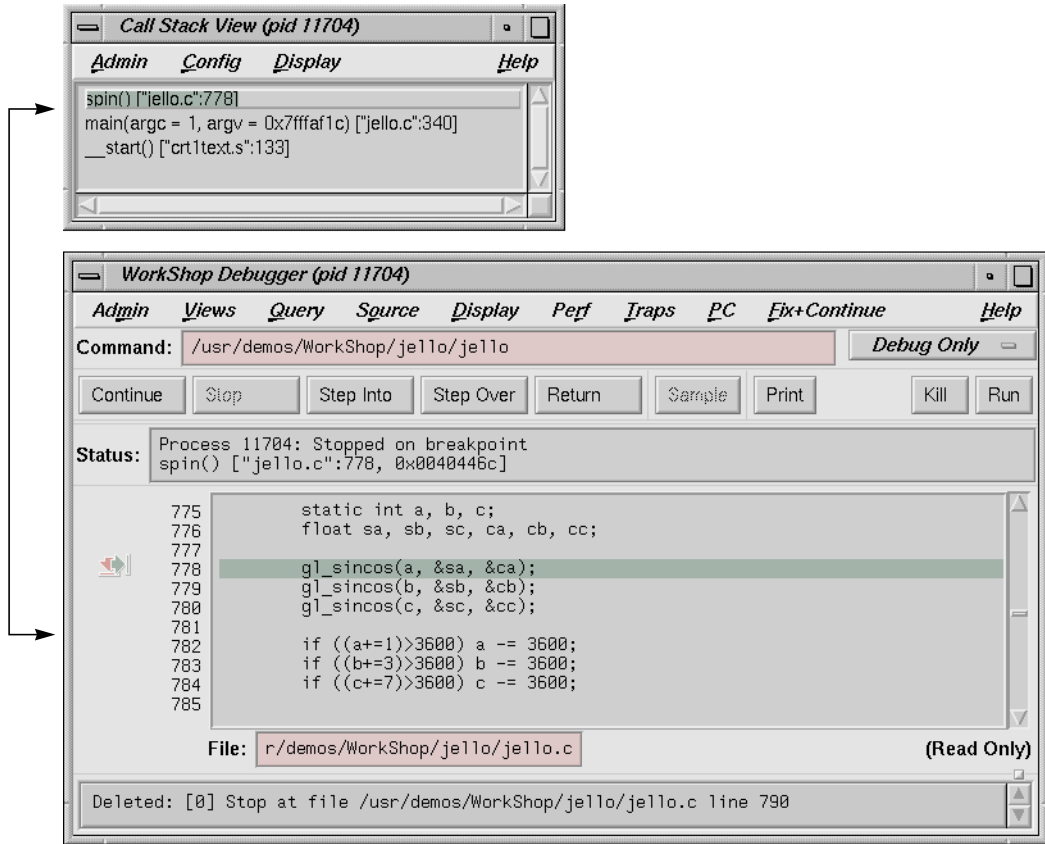


Figure 6-2 Tracing Through Call Stack View

Evaluating Expressions

You can evaluate any valid expression (in C, C++, or Fortran) 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 pre-processor, as in a **#define** directive or a macro.

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

Note: Expressions can also be evaluated from the command line.

Expression View

The Expression View window is shown in Figure 6-3 with its major menus displayed. Note that Expression View has two popup 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, on the left side of the window, 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: 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 right column.

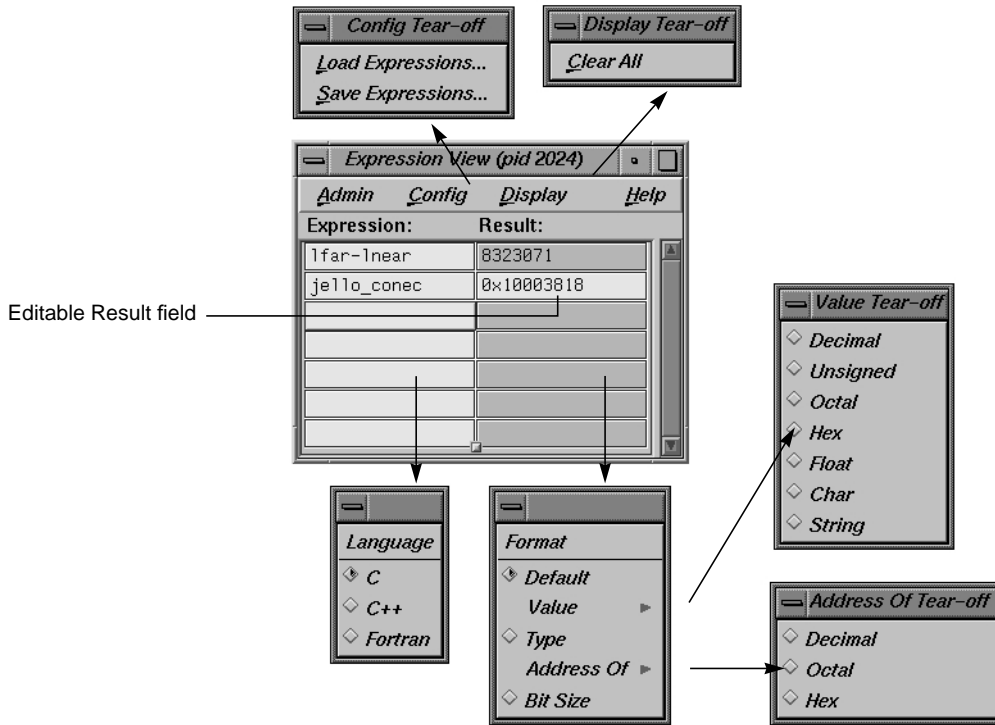


Figure 6-3 Expression View With Major Menus Displayed

If you want 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, or as decimal, unsigned, octal, hexadecimal, float, or characters. You can also display the type, the address (in decimal, octal, or hex), or the size of the result in bits.

Caution: The CASEVision Debugger uses the target program’s symbol table to determine the types of variables. 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 full-word integer. This gives the correct value for full-word integers or pointers, but the wrong value for non-full-word integers and for floating-point values.

To see the value correctly of a variable of unknown type, you can cast the address of the variable to a pointer to the correct type, using C type-cast syntax. 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 callstack). Like the other views involved with variables, Expression View has variable change indicators for value fields that let you see previous values, as shown in Figure 6-4.

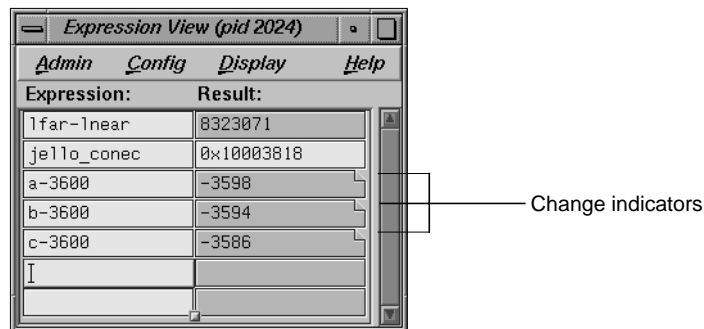


Figure 6-4 Change Indicators in Expression View

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...”

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. The valid expression operations are shown in the following paragraphs.

Evaluating Expressions in C

The valid C expressions are shown in Table 6-1.

Table 6-1 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	= += -= /= %= >>= <<= &x= ^= = (Note that a new assignment is made at each stepping point. Use assignments with caution to avoid inadvertently modifying variables.)

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 the user's program contains a successor function `succ`, the Debugger will evaluate the expression `succ(succ(succ(3)))` to 6.

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`.

Limitations

Constructors may be called from Expression View, just like any other member function. 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.

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 6-2 are supported on data of integral, real, and complex types.

Table 6-2 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.)

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.

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

This chapter provides an interactive sample session that demonstrates most of the Fix and Continue functions. The session outlines common tasks you can perform with Fix and Continue, using example C++ application source to illustrate the use of each function. For complete reference information on the Fix and Continue user interface, see “Fix+Continue Windows” on page 258.

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”
- “Redefining a Function”
- “Setting Breakpoints in Redefined Code”
- “Viewing Status”
- “Comparing Original and Redefined Code”
- “Ending the Session”

Setting Up the Sample Session

For this tutorial, use the demo files in the directory `/usr/demos/WorkShop/bounce`, which contain the complete source code for the C++ application *bounce*. To prepare for the session, you first need `tocreatefileset`, then launch Fix and Continue from the Debugger. You must enter the commands listed below:



Figure 7-1 Executive View Icon

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

The `cvd` command brings up the CaseVision Debugger, from which you can use the Fix and Continue utility. The Execution View icon (shown in Figure 7-1) and Main View (shown in [Figure 7-2](#)) appear. Note that the Debugger shows that the source code status indicator is (Read Only).

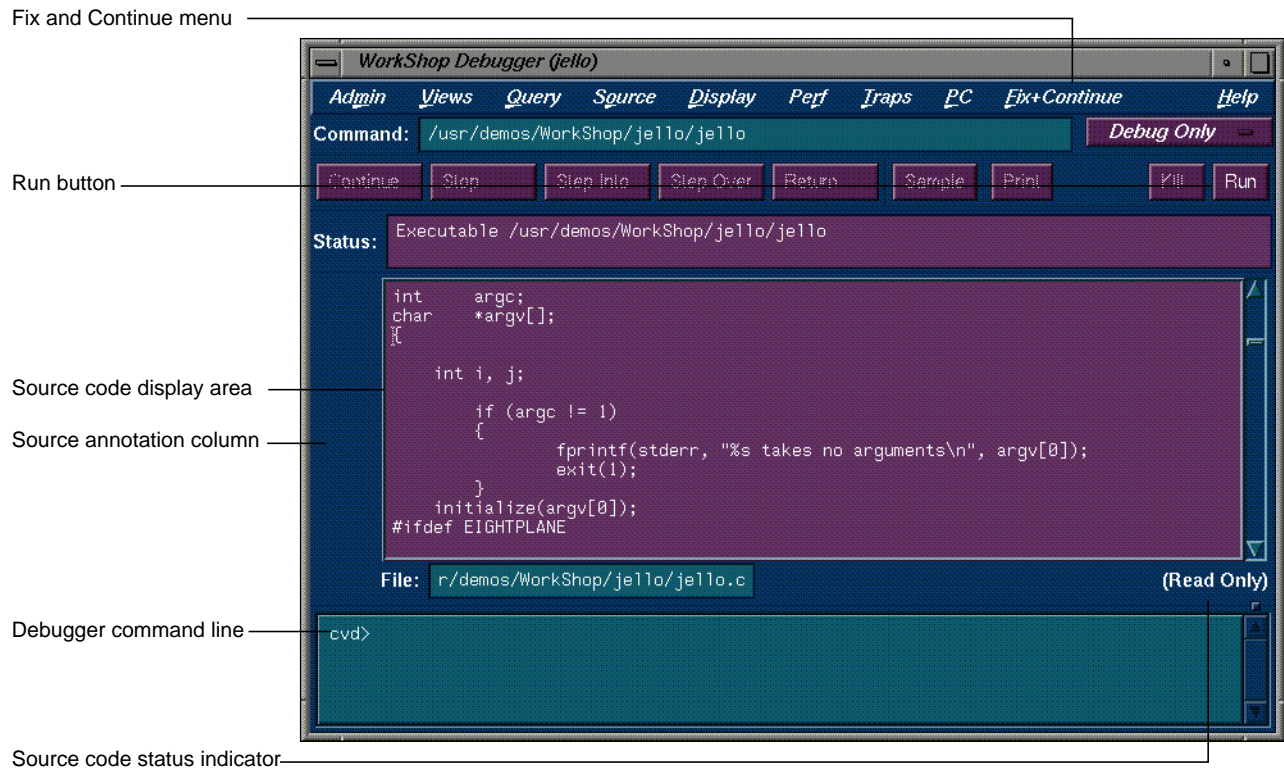


Figure 7-2 Debugger Main View With Fix and Continue Menu

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 7-3).

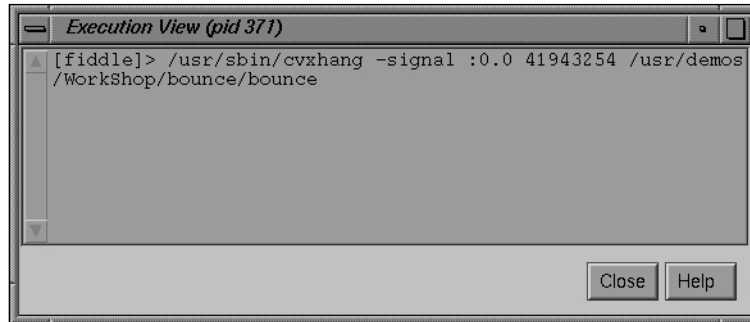


Figure 7-3 Program Results in Execution View

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

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

Editing a Function

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

```
cvd> func Clock::speedChanged
```

This opens the file *Clock.C*, and places the cursor at the beginning of the function **Clock::speedChanged**, as shown in [Figure 7-4](#).

```
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 7-4 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 7-5](#). Line numbers change 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 7-5 Redefined Function

Changing Code

1. To increase the speed of the ball, change the value of *_delta* from 1000 / value to 100 / value.
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.

If there are any errors, the Fix+Continue error messages window opens as shown in Figure 7-6. The Debugger command line also gives a report. If all went as planned, there are no errors or warnings.

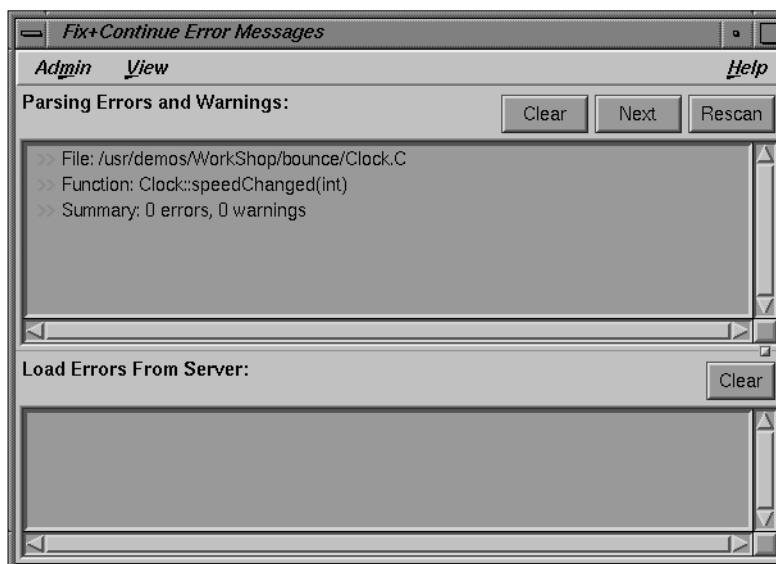


Figure 7-6 Checking Syntax Opens Fix and Continue Status Window

If you do have an error, correct it and repeat steps 1-3. You can go to the error location by double-clicking the appropriate line in the error message window. When you see the change ID and activated status, as shown in [Figure 7-7](#), continue with the next step.

When the parse and load has completed, the highlighting color of the function changes.

```
Debugger command line report —
cvd> func Clock::speedChanged
Change id: 1 redefined
Change id: 1 modified
Change id: 1 redefined
cvd>
```

Redefined function change id # Status

Figure 7-7 Report of Successful Redefinition

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 7-8](#)).

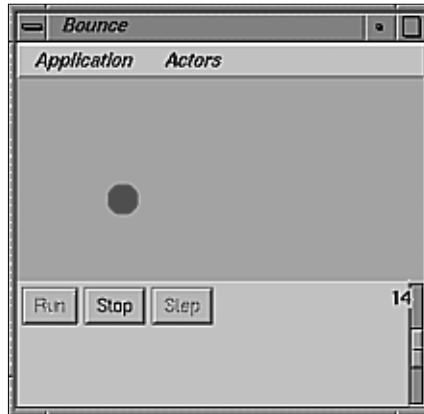


Figure 7-8 Bounce Window

Deleting Changed Code

If you make a mistake, there's a graceful way out. Suppose for example that you decided you didn't want to change the speed after all. To delete the change, you need merely select the "Delete Edits" option from the Fix+Continue menu in the Source view.

Changing Code From the Debugger Command Line

As an alternative to using the Fix and Continue menu, you can redefine and check syntax for a function from the Debugger command line. Try changing `_delta` to 100 by entering the following at the command line:

```
cvd> replace_source "Clock.C":83  
"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: 4 redefined  
Change id: 4 modified  
Process 5779 stopped at ["select.s":12, 0x0fac2010]  
Change id: 4 activated
```

```
Change id: 4 , build results:
 4 enabled /usr/demos/WorkShop/bounce/
Clock.C Clock::speedChanged(int)
cvd>
```

If you prefer to use the command line, experiment with `add_source` and `redefine` to get the same functionality described for the menu commands. For details on each command, refer to “Debugger Command Line” on page 271.

Saving Changes

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

1. Select “Save As...” from the Fix+Continue menu.
2. Look at the features of the dialog box (see [Figure 7-9](#)) that enable you to save your file. To save the changes back to the original source files, click that radio button and then click *Apply* or *OK*. To save to a different file, click the other radio button, choose a suffix, and click *Apply* or *OK*. Since you don’t want to save the changes, press *Cancel*.

Alternatively, on the Debugger command line, you could type `save_changes -file Clock.C Clock.C`. Either method saves all the changes to the file, replacing the compiled source code.

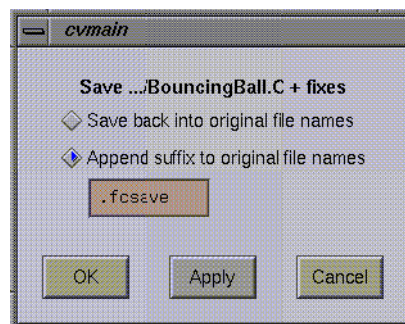


Figure 7-9 Saving a Function File

You usually want to wait until you are finished with Fix and 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. See “Fix+Continue Menu” on page 158 for more information.

Setting Breakpoints in Redefined Code

To see how the Debugger works with traps (breakpoints) in redefined code you'll set breakpoints, run the Debugger, and view the results (Figure 7-10).

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

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

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

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 your new *SIZE* definition by clicking in the source annotation column at line 35.5.

Alternatively, you can set a breakpoint through the command line by entering **stop at #** or **b #** where # 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.

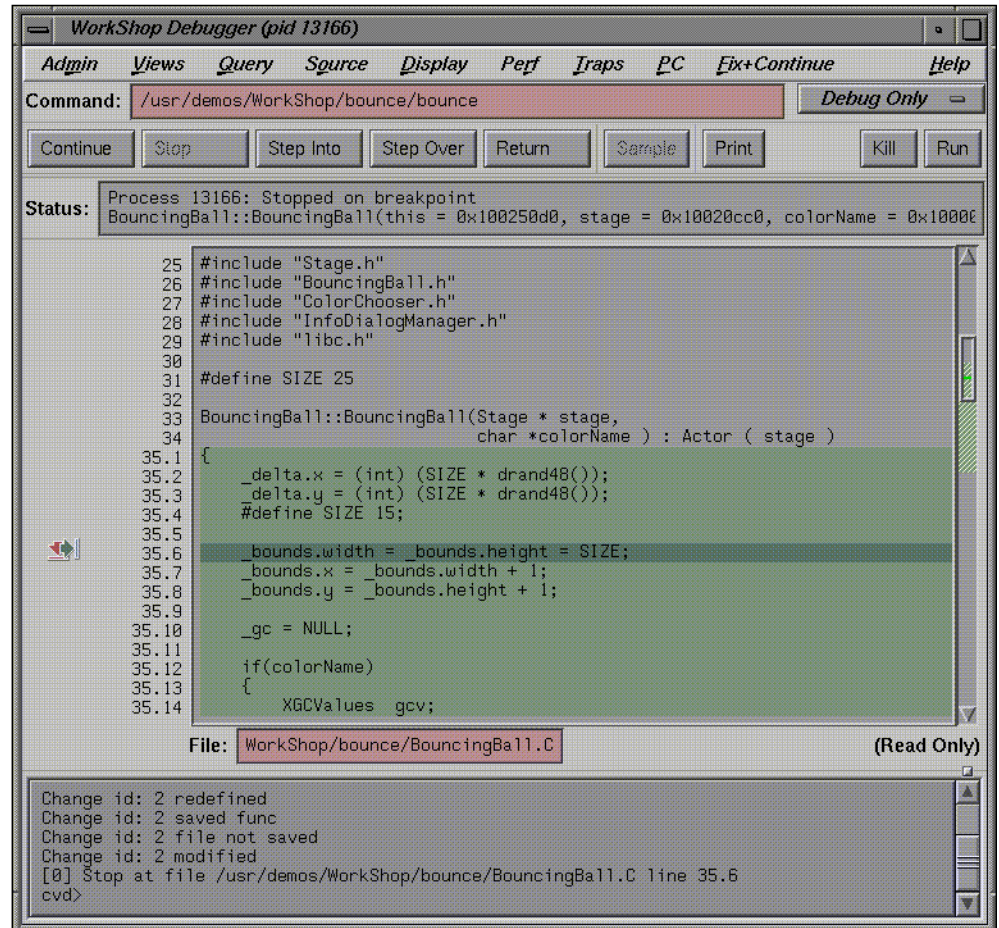


Figure 7-10 Stopping After Breakpoints in Redefined Code

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 (see Figure 7-11).

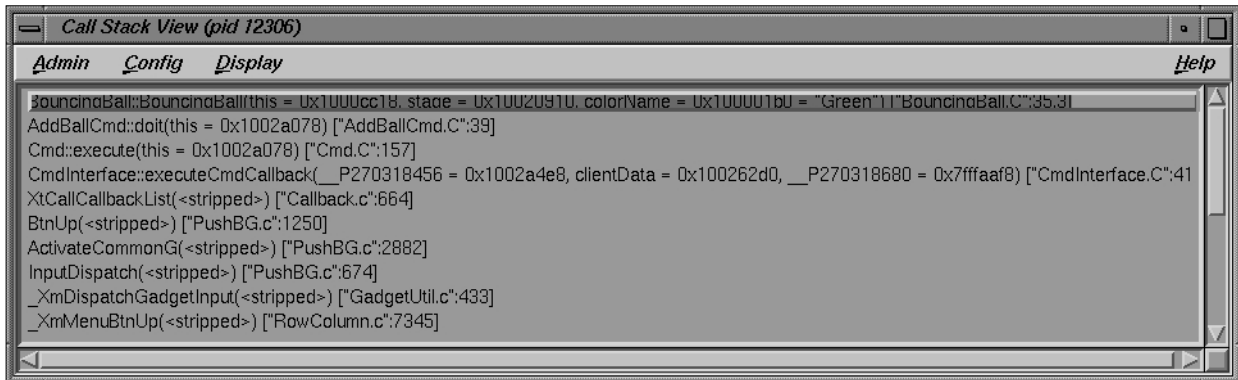


Figure 7-11 Call Stack BreakPoint Results

8. Select "Trap Manager" from the Views menu to view the locations of the traps (see Figure 7-12).

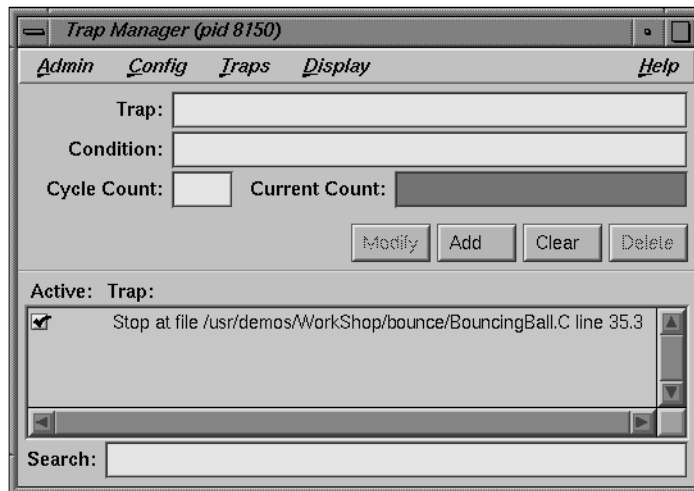


Figure 7-12 Trap Manager BreakPoint Results

9. Remove the breakpoint by clicking on it in the source annotation column.

Viewing Status

Pull down the Fix+Continue menu, choose the Views submenu, and select “Status Window”. The View Status window opens, as shown in Figure 7-13.

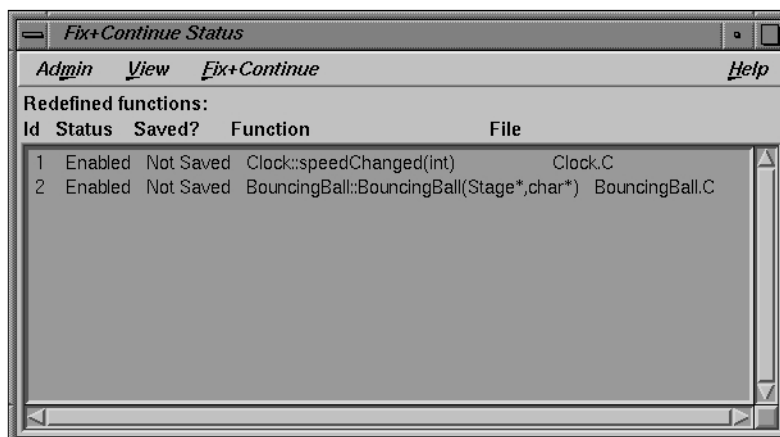


Figure 7-13 Using the View Status Window

Comparing Original and Redefined Code

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

Switching Between Compiled and Redefined Code

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

1. Select *Run* to view your redefined code. Notice that the balls you add are smaller in your modified version.
2. Place the insertion point in function **BouncingBall**.
3. Select “Edit<-->Compiled” from the Fix+Continue menu. This disables your changes.
4. Select *Continue*. 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 command `disable_changes #` from the Debugger command line, where # is the redefined function ID number.

To re-enable your changes, do the following:

5. Select *Stop*.
6. Select “Edit<-->Compiled” from the Fix+Continue menu. This re-enables your changes. The balls you add will now be smaller.

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

Comparing Function Definitions

1. Place the insertion point in the **BouncingBall** function body.
2. Pull down the Fix+Continue menu, choose the Show Difference submenu, and select “For Function”. A *xdiff* window opens as shown in Figure 7-14.

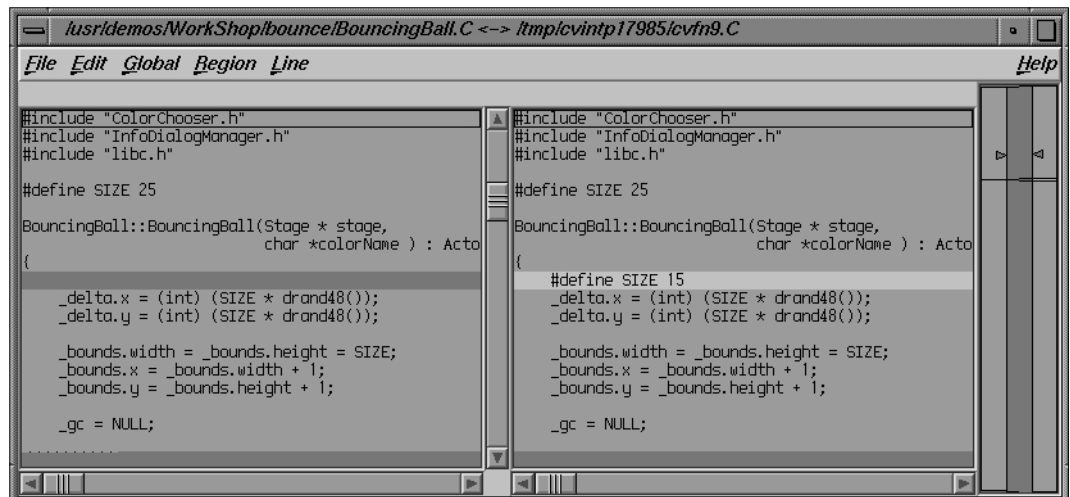


Figure 7-14 Comparing Compiled vs. Redefined Function Code: *xdiff*

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

If you don't 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...".

Comparing Source Code Files

When you have made several redefinitions to a file, sometimes you need a side-by-side comparison of the entire file. To see how your changes to the 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'll get the same result from the Debugger command line if you enter the following:

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

As an alternative to pulling down menus using the mouse, you can use mnemonics to select the menu item from the keyboard. After closing the difference window, you'll reopen it. With the insertion point anywhere in the file, enter the following:

Alt-f d f

Ending the Session

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

Detecting Heap Corruption

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

- “Typical Heap Corruption Problems”
- “Detecting Heap Corruption Errors”
- “Heap Corruption Detection Tutorial”

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.

Detecting Heap Corruption Errors

To detect heap corruption problems, you need to relink your executable with a special WorkShop *malloc* library (*-lmalloc_cv*) instead of the standard *malloc* library (*-lmalloc*). By default, the library always catches these 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**.

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 environment variable `_RLD_LIST` to link the `-lmalloc_cv` library. See the reference (man) page for `rld(1)`.

Setting the Environment Variables

After compiling, you 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 the data in any memory allocation freed by `free`. It also 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 the data in any memory allocation returned by `malloc`. It also 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 certain 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 “guard words”, that is, specific patterns, in front of and behind the area returned to the caller. When `free` or `realloc` is called on a block, its guard words 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 really made, and the process space and swap requirements can grow quite large.

MALLOC_TRACING

prints out all the *malloc* events including address and size of the *malloc* or *free*. Tracing is normally done in the course of a performance experiment; the variable need not be set in such cases, because the running of the experiment automatically enables it. If the option is enabled when the program is run independently, and *MALLOC_VERBOSE* is set to 2 or greater, the 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 reference page for *malloc_cv*.

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** which 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 *reallocated*.

When you run the program, it will halt 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 run your program again. 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 *-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.

Heap Corruption Detection Tutorial

This tutorial demonstrates how to detect corruption errors, using a program called *corrupt*. The *corrupt* program has already been linked with the WorkShop *malloc* library (*libmalloc_cv*). Its listing 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;
}
```

1. Go to the directory `/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 *Add*:

```
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 8-1 with the stop trap set.

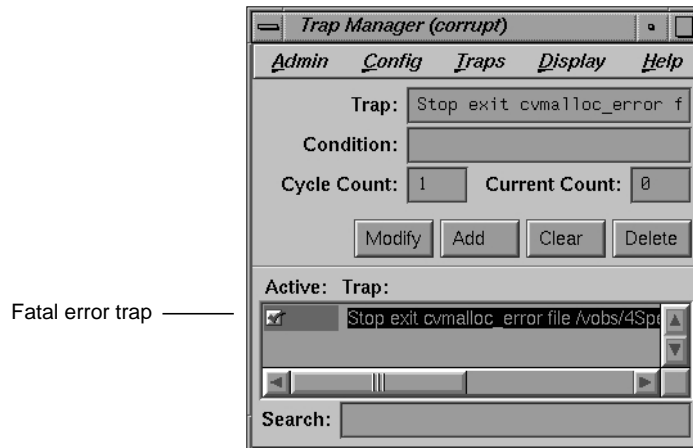


Figure 8-1 Setting Traps to Detect Heap Corruption

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 8-2.

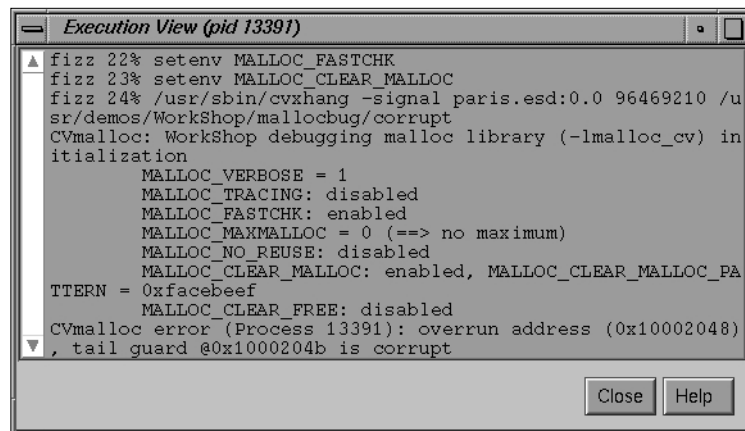


Figure 8-2 Heap Corruption Warning Displayed in Execution View

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 8-3).

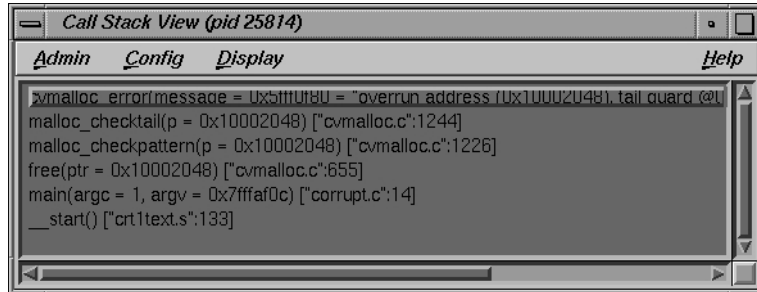


Figure 8-3 Call Stack at Boundary Overrun Warning

8. Click *Continue* in the Main View control panel and watch Execution View and Call Stack View.

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 *Continue* again and watch Execution View and Call Stack View.

This time the process stops at a bus error. The PC stops at the statement:

```
value=*bogus
```

because *bogus* was set to an uninitialized value.

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

This gives us the address for the variable *bogus* and has been done in Figure 8-4. We need the bad address so that we can set a watchpoint to find out when it is written to. (Note in this example that the address is 0x7fffaef4—your address will be different.)

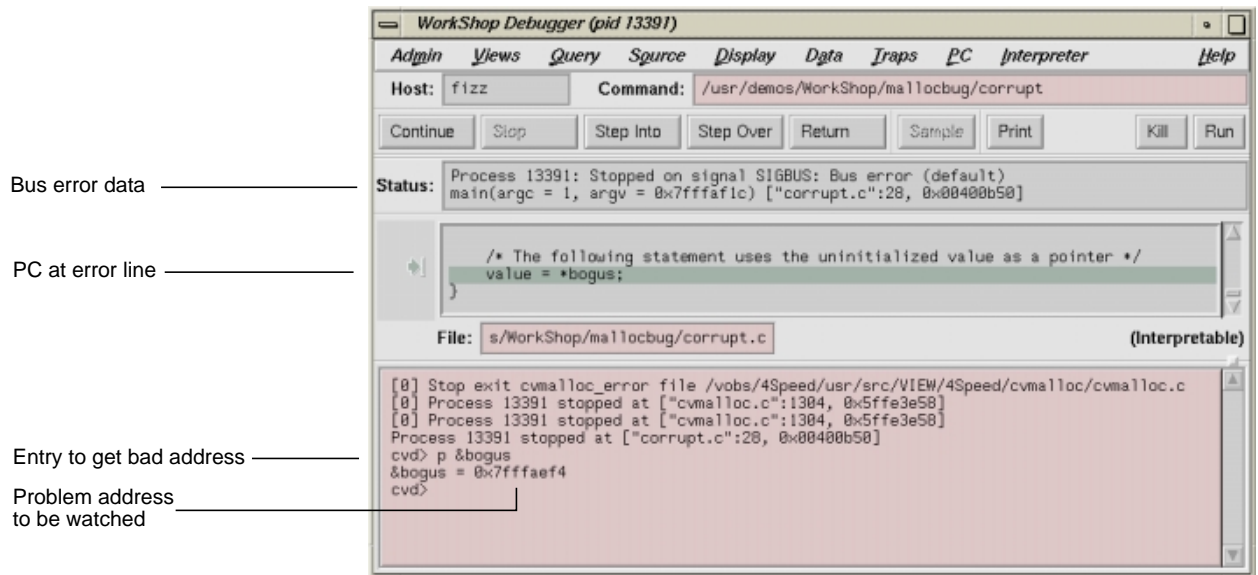


Figure 8-4 Main View at Bus Error

11. Deactivate the stop trap by clicking the toggle button next to the trap description in the Trap Manager window, and click *Kill* in Main View to kill the process.
12. Type the following command in the *Trap* field in the Trap Manager using the address you obtain from the Debugger command line (see Figure 8-4) and click *Add*:


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

Use the actual 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 *Run* and observe Main View.

The process stops at the point where the variable *bogus* gets a bad value. The details of the error display in the Main View *Status* field (see Figure 8-5).

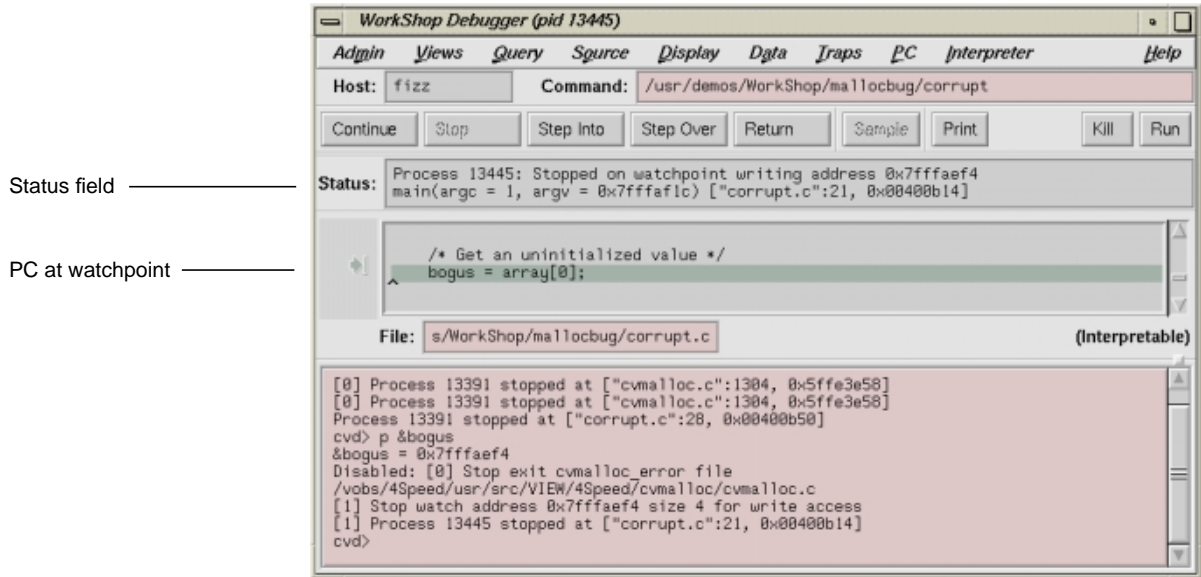


Figure 8-5 Watch Point Error Displayed in Main View

Multiple Process Debugging

WorkShop supports performance analysis and debugging of multiprocess applications, including processes spawned either with *fork* or *sproc*. 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.

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

- “Debugging With Multiprocess View”
- “Controlling Execution and Setting Traps in a Multiprocess Program”
- “Debugging a Multiprocess Fortran Program”

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. Note that a child that performs an *exec* with *setuid* (the user ID) enabled will not become part of the process group. 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.

Each process in the session can have a standard Main View session associated with it. All processes in a process group share a single Multiprocess View. Selecting “Multiprocess View...” from the Admin menu in Main View for any process in the group brings up the Multiprocess View

window. If the Multiprocess window exists, it will be raised to the front; otherwise, a new window will be created.

Currently, Multiprocess View handles these multiple process situations:

- *True multiprocess program*, which refers to a tightly integrated system of *sproc'd* processes, generated by POWER/Fortran or POWER/C.
- *Auto-fork application*, which is a process that spawns a child process and then runs in the background.
- *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.
- *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*.

Multiprocess View does not support remotely distributed applications.

Displaying the Multiprocess View

The first step in debugging multiple processes simultaneously is to invoke the Debugger with the parent process. Then select "Multiprocess View" from the Admin menu to bring up Multiprocess View. Main View is attached to the parent process. Figure 9-1 shows a typical Multiprocess View with Config and Process menus displayed.

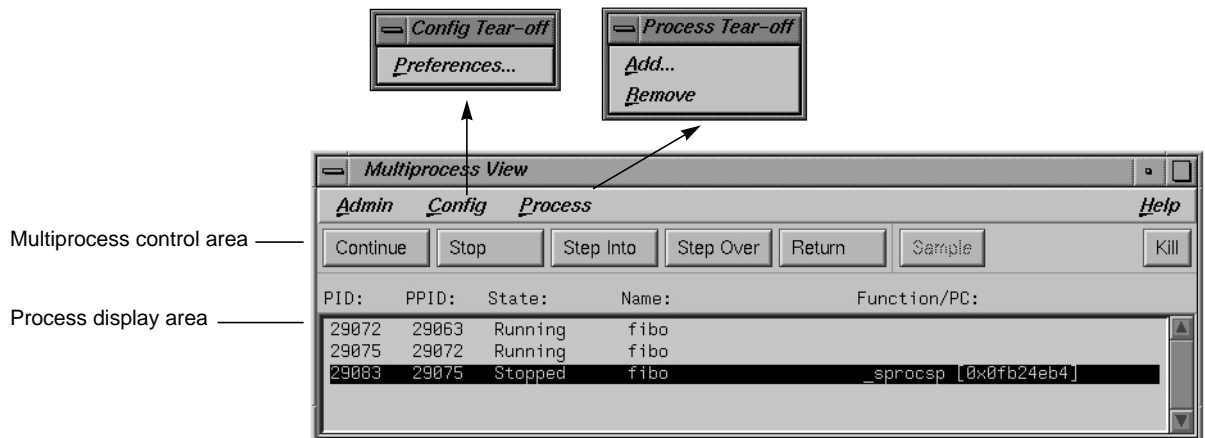


Figure 9-1 Multiprocess View With Config and Process Menus Displayed

To open a Main View (or other debugging views) for another process, double-click the desired process in Multiprocess View. A separate Main View window displays the selected process, and you can select any debugging views desired. If a set of views exists for that process, the views are raised to the foreground. To reuse views already displayed, select “Switch Process...” from the Admin menu in Main View. (If a process is currently highlighted in Multiprocess View, its ID is entered automatically in the *Process ID:* field in the Switch Process dialog box.)

Viewing Process Status

When Multiprocess View comes up, it lists the status of all processes in the process group. This information includes:

- PID:* shows the process identifier (ID).
- PPID:* lists the parent process IDs. Notice in Figure 9-1 that the first process PID#7748 is the parent process of the second.
- State:* represents the state of the process: stopped, running, or created, which appears just prior to running. Terminated processes are not displayed.
- Name:* identifies the process by filename.

Function/PC: indicates the current function and program counter (PC) for any stopped processes.

Multiprocess Control Buttons

Multiprocess View uses the same control buttons as MainView 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 buttons are:

- Continue* resumes program execution after a halt and continues until a stop trap or other event stops execution.
- Stop* 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* 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 popup menu displays that lets you select one of the fixed values or a specified number of steps.
- Step Over* 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 popup menu displays that lets you select one of the fixed values or a specified number of steps.
- Return* executes the remaining instructions in the current function. Program execution stops upon return from that procedure.
- Sample* collects performance data for each process (if performance data collection is enabled).
- Kill* terminates all processes in the group.

Multiprocess Traps

As discussed in Chapter 4, "Setting Traps," 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 the “Group Trap Default” and “Stop All Default” selections in the Traps menu in Trap Manager.

Note that the *Sample* button always samples all processes.

Adding and Removing Processes

The Process menu lets you manually add or remove a process from the process group (see Figure 9-2).

To remove a process, click the process and select “Remove” from the Process menu. Note that a process in a *sproc* share group cannot be removed from the process group.

To add a process, select “Add...”. The dialog box shown in Figure 9-3 displays. Enter the new process ID and click *OK*.

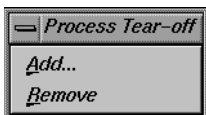


Figure 9-2
Process Menu in
Multiprocess View

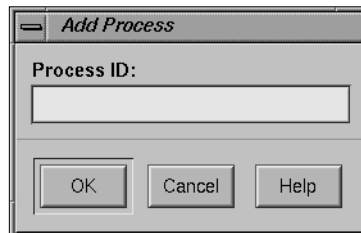


Figure 9-3 Add Process Dialog Box

Multiprocess Preferences

The “Preferences...” option in the Config menu brings up the Preferences dialog box. It lets you control when processes are added to the group, and it specifies their behavior (see Figure 9-4).

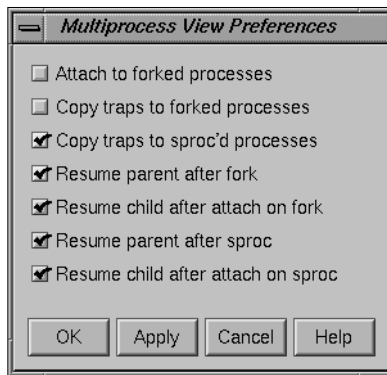


Figure 9-4 Multiprocess View Preferences Dialog Box

The Multiprocess View preference options are:

Attach to forked processes

attaches new processes spawned by the *fork* command to the group automatically. (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 sproced.

Resume child after attach on sproc

restarts the new sproced process automatically when it is attached. If this option is left off, a new process will stop as soon as it is attached.

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 of the tasks you'll be performing most often when using *cvd* to debug *mp* code. The tasks demonstrated are:

- 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 program *fibonacci* 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 *fibonacci.c* in the directory */usr/demos/WorkShop/mp*.

A listing of *fibonacci.c* 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 (; i) {
        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 (; i) {
        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 get started, compile the program and run the Debugger.

1. Compile *fibonacci.c*.

```
cc -g fibonacci.c -o fibo
```

2. Invoke the Debugger on *fibonacci*.

```
cvd fibo &
```

3. Bring up the multiprocess view by selecting "Multiprocess View..." from the Admin menu.

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

Using the 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 the Debugger's behavior on *sproc* by setting Multiprocess preferences.

1. Select "Preferences..." from the Config menu in Multiprocess View.
2. Deactivate *Resume child after attach on sproc*.

At the same time, you can turn off trap inheritance, so you can experiment with trap setting later.

3. Click OK to accept the change.

Now you're ready to run the process.

4. In the Main View, click *Run*.

If you watch Multiprocess View, 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. Double-click the child process in the Multiprocess View window.

You see a dialog box like the one in Figure 9-5, and the Debugger creates a new window.

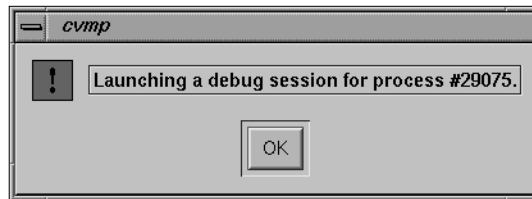


Figure 9-5 Launching a Debug Session Dialog Box

You can use the buttons in Multiprocess View to control all the processes simultaneously, or use the buttons in each of the Main Views to control each process separately.

Note: You'll probably get a warning that the *sproc.s* is missing. This is a reference to assembly code and can be ignored.

6. To send the child process on its way, click *Continue* in the Multiprocess View window.

The first child now spawns a grandchild process. The grandchild stops in *sproc*, as shown in Figure 9-6:

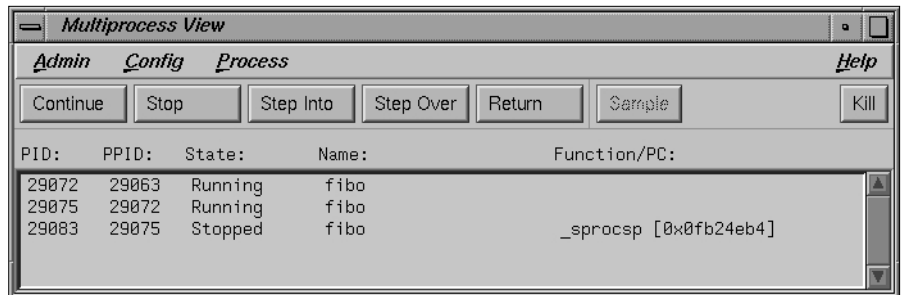


Figure 9-6 Using the Multiprocess View to Examine Process State

Using the Trap Manager to Control Trap Inheritance

This section shows you how to use the Trap Manager to set traps that affect one or all of the *fibonacci* process group. For complete information on using the Trap Manager, refer to Chapter 4, "Setting Traps."

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

Right now, traps set using the Traps menu in any of the Main View windows will affect only the process controlled by that 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 *fibonacci.c*.

Only the parent process halts. The child processes continue running, as a glance at Multiprocess View will confirm.

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 9-7.

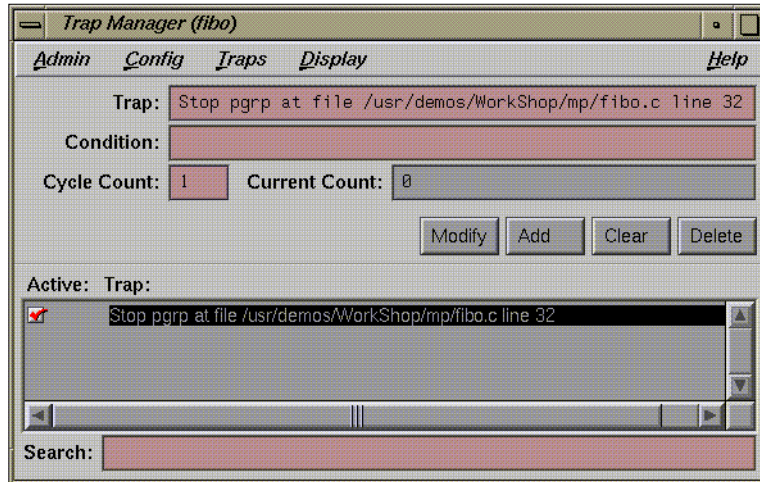


Figure 9-7 Modifying a Trap to Affect a Process Group

4. Click *Modify* to accept your change to the trap.

The trap affects the two child processes as well. Watch the Multiprocess View to see the whole process group stop at the trap on line 32.

You can set an option to make all traps affect the process group by default for those traps set using the Trap Manager.

5. Select “Group Trap Default” from the Traps menu (see Figure 9-8).

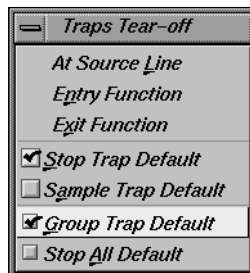


Figure 9-8 Setting the Group Trap Default

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.

The trap you've just set includes the modifier **pgrp**. It automatically affects both child processes.

You have now learned the basics of controlling the execution of multiple processes and setting traps.

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

Note that the Multiprocess View window must be closed explicitly. It does not close when the Main View windows do.

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 use the WorkShop Debugger to debug the sample program from Chapter 6 of the *Fortran 77 Programmer's Guide*.

General Fortran Debugging Hints

Debugging a multiprocessed program is more difficult than debugging a single-processor program; therefore, debug as much as possible on the single-processor version.

Try to isolate the problem as much as possible. If you can, reduce the problem to a single *C\$DOACROSS* loop.

Once you've isolated the problem to a specific *DO* loop, try changing the order of its 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 flags **-g** and **-mp_keep**. The **-mp_keep** flag saves the file containing the multiprocessed *DO* loop Fortran code. The compiler saves the code in a file named

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

where *user_subroutine_name* is the name of the subroutine containing the DOACROSS, *machine_name* is your machine name, and *pid* is the process ID number of the compilation.

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

Multiprocess Debugging Session

This section walks you through the process of using the Debugger to debug a small segment of incorrectly multiprocessed code. The example used in this section is also treated in Chapter 6 of the *Fortran 77 Programmer's Guide* with *dbx*. You can use *cvd* to perform the same tasks with less effort.

If you installed the demo programs, you can find the 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
```

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.

3. If you use the C shell, type

```
% setenv MP_SET_NUMTHREADS 2
```

4. Start the Debugger:

```
% 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:

```
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.

This sets the group default.

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

This selects the line.

9. From the Traps menu in Traps Manager, select “At Source Line.”

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

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

10. Bring up the Multiprocess View to keep tabs on the status of the two processes.

Now you’re ready to run the program.

11. Click *Run* in the Main View window.

As you watch the Multiprocess View, you’ll 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. Double-click the slave process listed in the Multiprocess View window, as in Figure 9-9.

This invokes a Main View debugging session on the slave process.



Figure 9-9 Launching a New Debugging Session From Multiprocess View

Now you can invoke the Variable Browser on each process. Look at *ii* and *jj* in Figure 9-10.

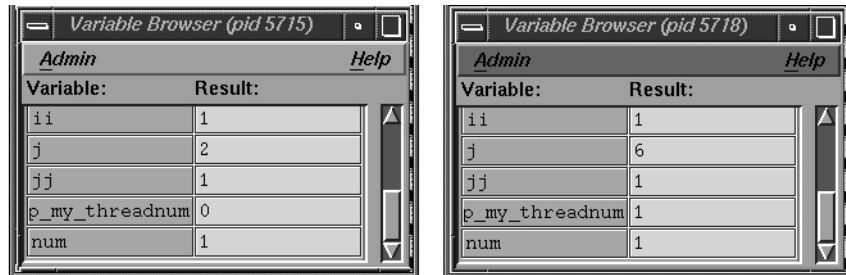


Figure 9-10 Comparing Variable Values From Two Processes

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've found the bug in your parallel Fortran program.

Using the X/Motif Analyzer: A Tutorial

This chapter provides an interactive 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”
- “Navigating the Widget Structure”
- “Examining Widgets”
- “Setting Callback Breakpoints”
- “Using Additional Features of the Analyzer”
- “Ending the Session”

Setting Up the Sample Session

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



Figure 10-1
Execution View Icon

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 CaseVision Debugger, from which you can use the X/Motif Analyzer. You see the Execution View icon (shown in Figure 10-1) and Main View (shown in Figure 10-2) appear. Note that the source code status indicator in the Debugger is (Read Only).

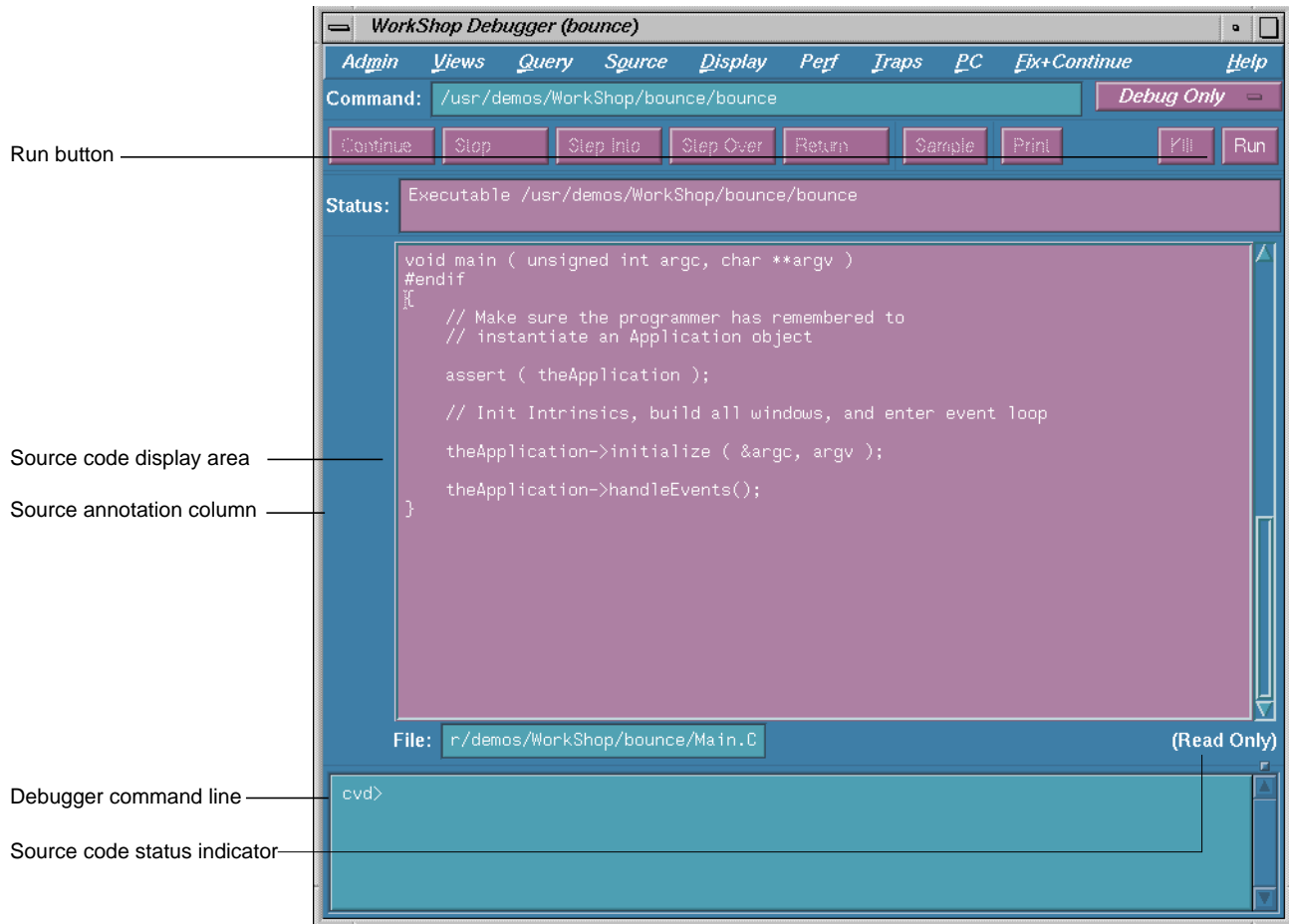


Figure 10-2 Debugger Main View

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, 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 10-3).

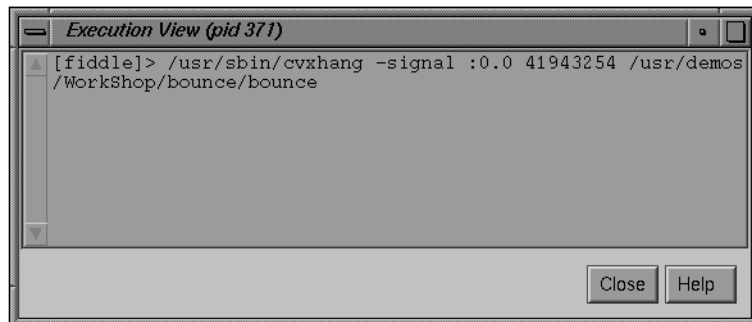


Figure 10-3 Program Results in Execution View

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

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 wish to change your `$LD_LIBRARY_PATH` environment variable to include `/usr/lib/WorkShop/Motif`. These are instrumented versions of the Silicon Graphics 5.3 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.

Navigating the Widget Structure

After being launched, the X/Motif Analyzer brings up an empty Widget examiner. The tab panel also shows the Breakpoints area, Trace examiner, and Tree examiner (see Figure 10-4).

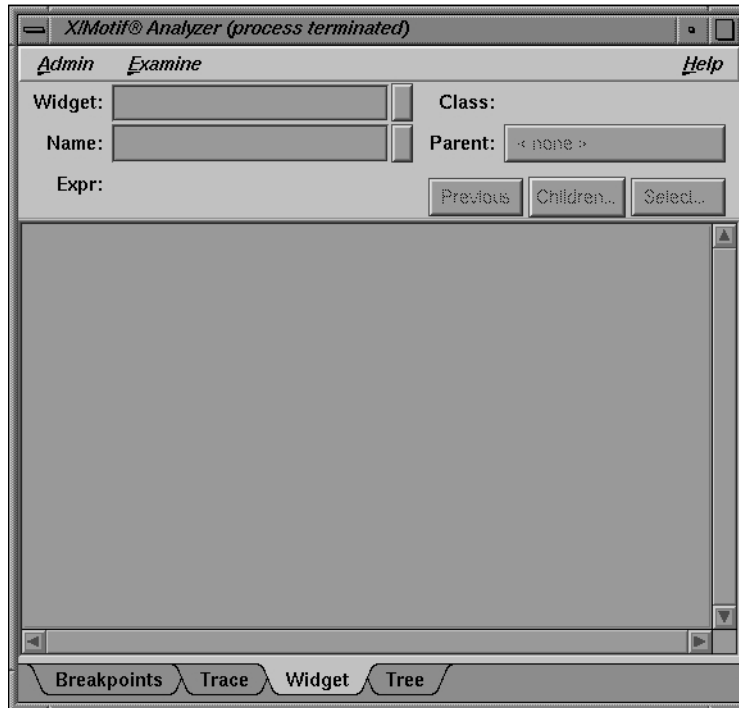


Figure 10-4 First View of the X/Motif Analyzer (Widget Examiner)

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* in the X/Motif Analyzer. This brings up an information dialog and changes the cursor to a +. Do not click *OK* in the information dialog. Click *Step* in the Bounce window as instructed by the dialog. The widget examiner displays the **Step** widget structure.

4. In the X/Motif Analyzer, click the Tree tab. The tree examiner displays the widget hierarchy of the target object (see Figure 10-5).

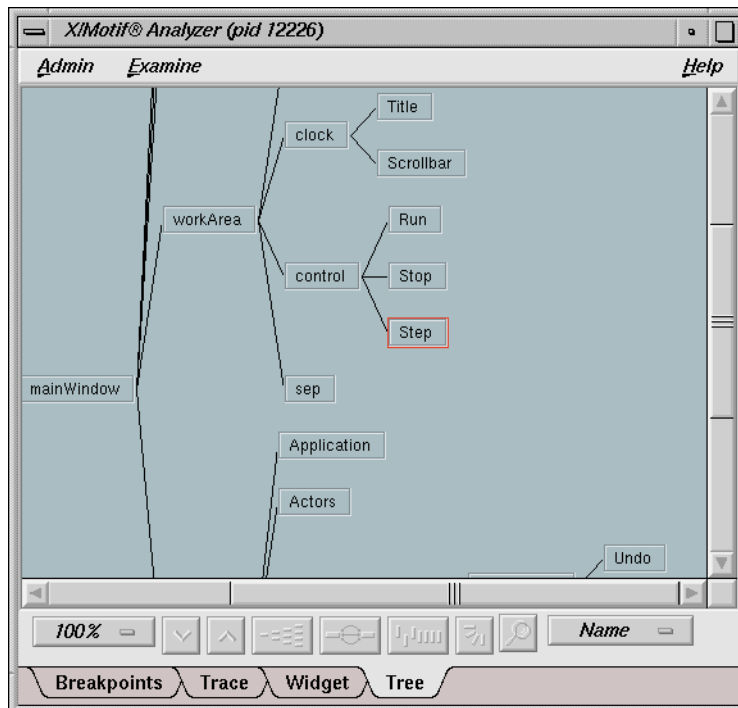


Figure 10-5 Widget Hierarchy Displayed in the Tree Examiner

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, which 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, 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.

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 *cod* 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. In the X/Motif Analyzer, 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 10-6.

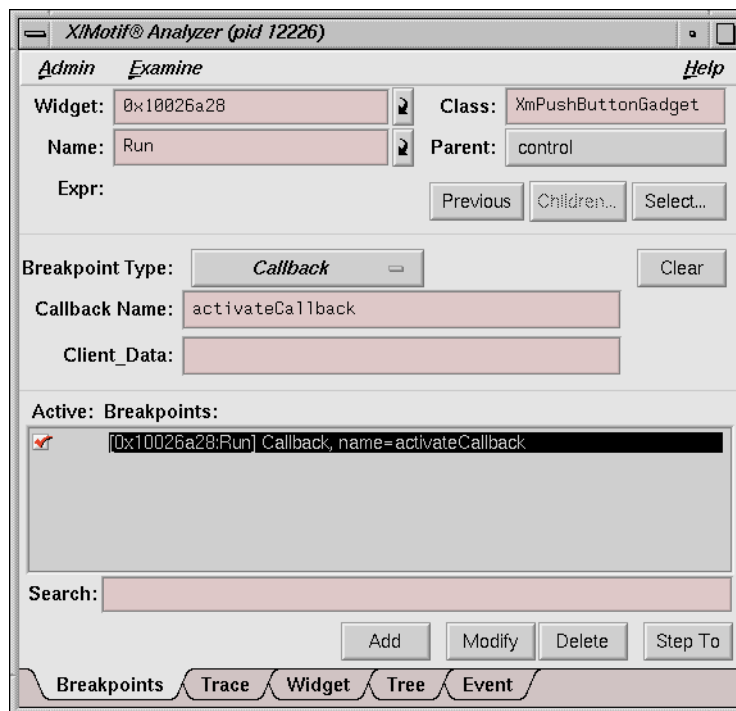


Figure 10-6 Adding a Breakpoint for a Widget

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.

Setting Callback Breakpoints

1. In the X/Motif Analyzer, click the Breakpoints list item to highlight the breakpoint.
2. In the X/Motif Analyzer, 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 10-7).

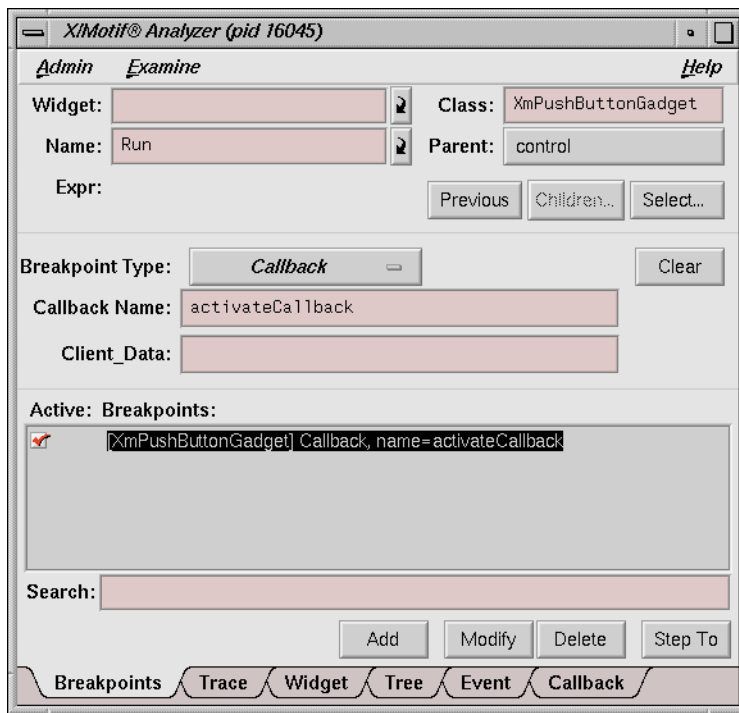


Figure 10-7 Setting Breakpoints for a Widget Class

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.

- In the X/Motif Analyzer, click the Callback tab to go to the callback examiner. This examiner displays the callback context and the appropriate `call_data` structure (see Figure 10-8).

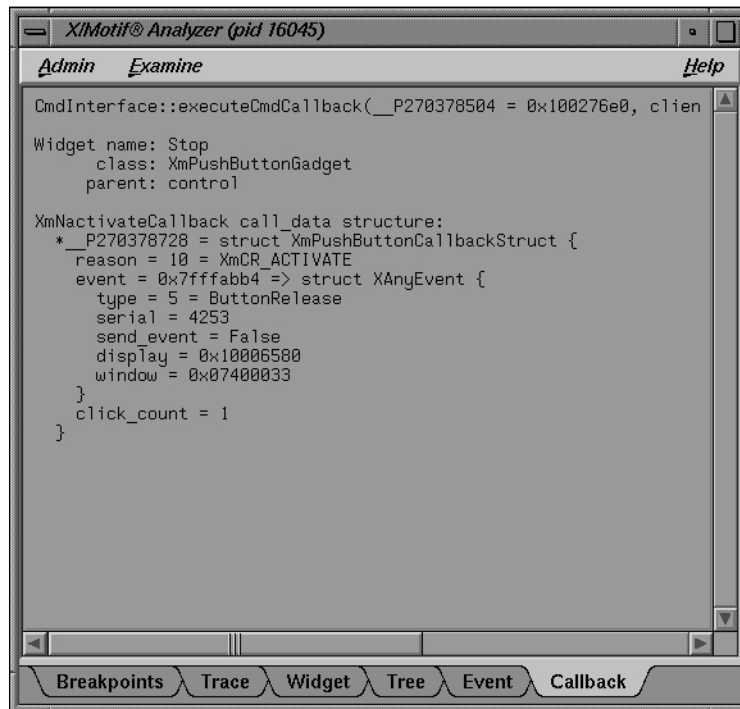


Figure 10-8 Viewing the Callback Context With the Callback Examiner

- Double-click the window value in the callback structure, fourth line from bottom.
- 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 10-9.

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.

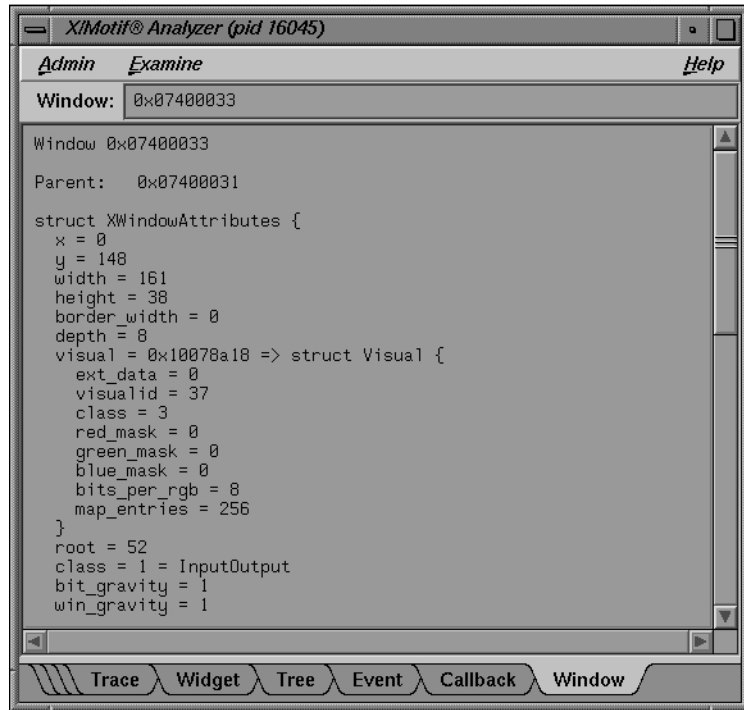


Figure 10-9 Viewing Window Attributes With the Window Examiner

Using Additional Features of the Analyzer

1. In the X/Motif Analyzer, 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 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 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 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 now displays the widget `control`, the parent of *Run*. This action produces the same results as clicking the *Parent* button.
8. In the X/Motif Analyzer, 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 10-10).

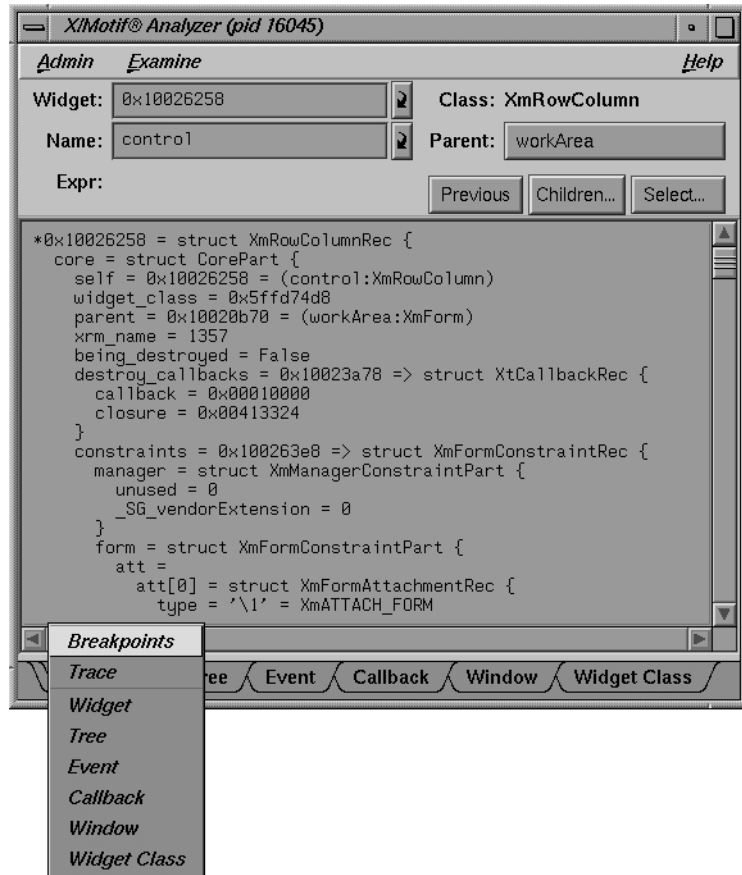


Figure 10-10 Selecting the Breakpoints Tab From the Overflow Area

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 in instances when the "sensitive" resource is changed for any push button gadget.
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 10-11).

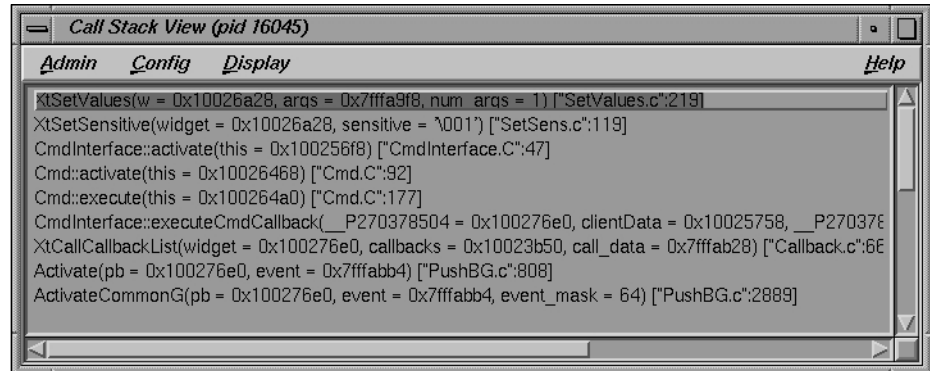


Figure 10-11 Viewing Breakpoint Results in the Callstack View

15. In the Callstack view, double-click the **Cmdinterface::activate** frame (just below **XtSetSensitive**). This is where the sensitive resource was changed.
16. In the X/Motif Analyzer, click the Widget tab.
17. In the X/Motif Analyzer, double-click the widget address in the “Widget” text field, press backspace, enter `_w`, and press <Enter>. The X/Motif Analyzer now 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 displays the **Step** widget, which is the widget now being changed.

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

This chapter describes in detail the function of each window, menu, and display in the Debugger's graphical user interface (GUI). In addition, the chapter describes the Debugger commands available on the Debugger command line (see "Debugger Command Line" on page 267). Most commands are available from either interface. You can move from one to the other as you prefer.

This chapter contains the following sections:

- "Main View"
- "Basic Windows"
- "X/Motif Analyzer Windows"
- "Project Session Management Windows"
- "Data Examination Windows"
- "Machine-level Debugging Windows"
- "Multiple Process Debugging Windows"
- "Fix+Continue Windows"
- "Debugger Command Line"

Main View

The major areas of the Main View window are shown in Figure A-1.

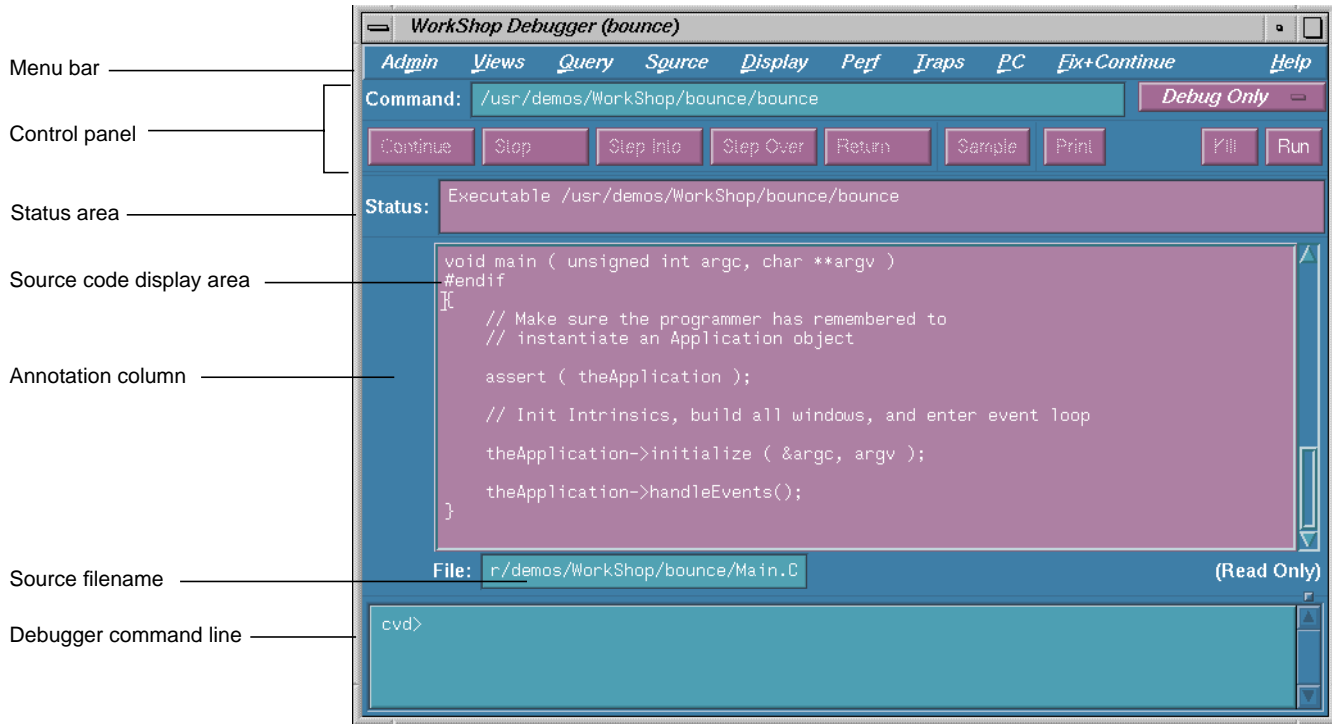


Figure A-1 Major Areas of the Main View Window

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 items, which are discussed in detail in later pages:

- "Admin Menu"
- "Views Menu"
- "Query Menu"
- "Source Menu"
- "Display Menu"

- “Perf Menu”
- “Traps Menu”
- “PC Menu”
- “Fix+Continue Menu”
- “Help Menu”

In addition, the Main View contains the following items:

“Command” text field

Displays the full pathname of the executable that you are currently debugging.

Debug option menu

Allows you to conduct performance experiments using either the built-in WorkShop performance tools, or the Purify memory corruption analysis tool. The option menu choices are:

- “Debug Only,” which runs the Debugger in Debug mode only, with no performance tools enabled.
- “Performance,” which causes performance data to be gathered and instrumented code to be generated for performance analysis while using the debugger.
- “Purify,” which causes the Purify memory corruption analysis tool is active during your debugger run. The code that you view with the debugger (in Main View, Source View, and so on) is Purify code. For further information on Purify, see the Purify documents.

Note: Purify is not part of the standard debugger package; it must be ordered separately.

Continue

Continues the execution of the current process. This command is legal only if the running process is stopped. If the program has not been run or has been killed, the Continue button is desensitized (grayed-out). If the target program has not yet started executing, use the Run command to start execution.

Stop Stops the execution of the current process while it is running. This command is valid only when a process is running; otherwise the command button is desensitized (grayed-out). Traps can also be planted to stop the program at a specific location or on a particular condition. See the Trap Manager for more details.

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 is legal only if the running process is stopped; otherwise the command button is desensitized (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. The step value menu selections consist of "1, 2, 3, 4, 5, 10, 15, 20, N..." If you choose the last menu entry "N...", 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. The step value menu selections consist of "1, 2, 3, 4, 5, 10, 15, 20, N..." If you choose the last menu entry "N...", a dialog window is opened to allow you to enter a step value.

<i>Return</i>	Continues the 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 is legal only if the running process is stopped; otherwise the command button is desensitized (grayed-out). This command is not allowed if the executable is instrumented for performance analysis.
<i>Sample</i>	Allows you to manually sample the state of a process for evaluation by the Performance Analyzer. This command is legal only if the process is running and the Enable Data Collection mode is set on the Performance panel; otherwise the command button is desensitized (grayed-out).
<i>Print</i>	Prints the value of the currently selected expression.
<i>Kill</i>	Kills the currently running process that you are debugging by sending it the equivalent of a “kill -9” signal. This command is legal if the process is running or stopped; otherwise the command button is desensitized (grayed-out).
<i>Run</i>	Runs the program that you are currently debugging. After the initial run, <i>Run</i> allows you to rerun the program, 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

Where such things as stop points are displayed.

“File” text field

Displays the name of the file that you are currently debugging.

Command line area

Area of the Main View where you can enter command-line Debugger commands.

Show/Hide annotations button

This button (see Figure A-2) only becomes visible if you run or load a performance experiment (see the *Performance Analyzer User’s Guide* for more information on the performance tools). This is a toggle button that shows or hides performance related annotations.

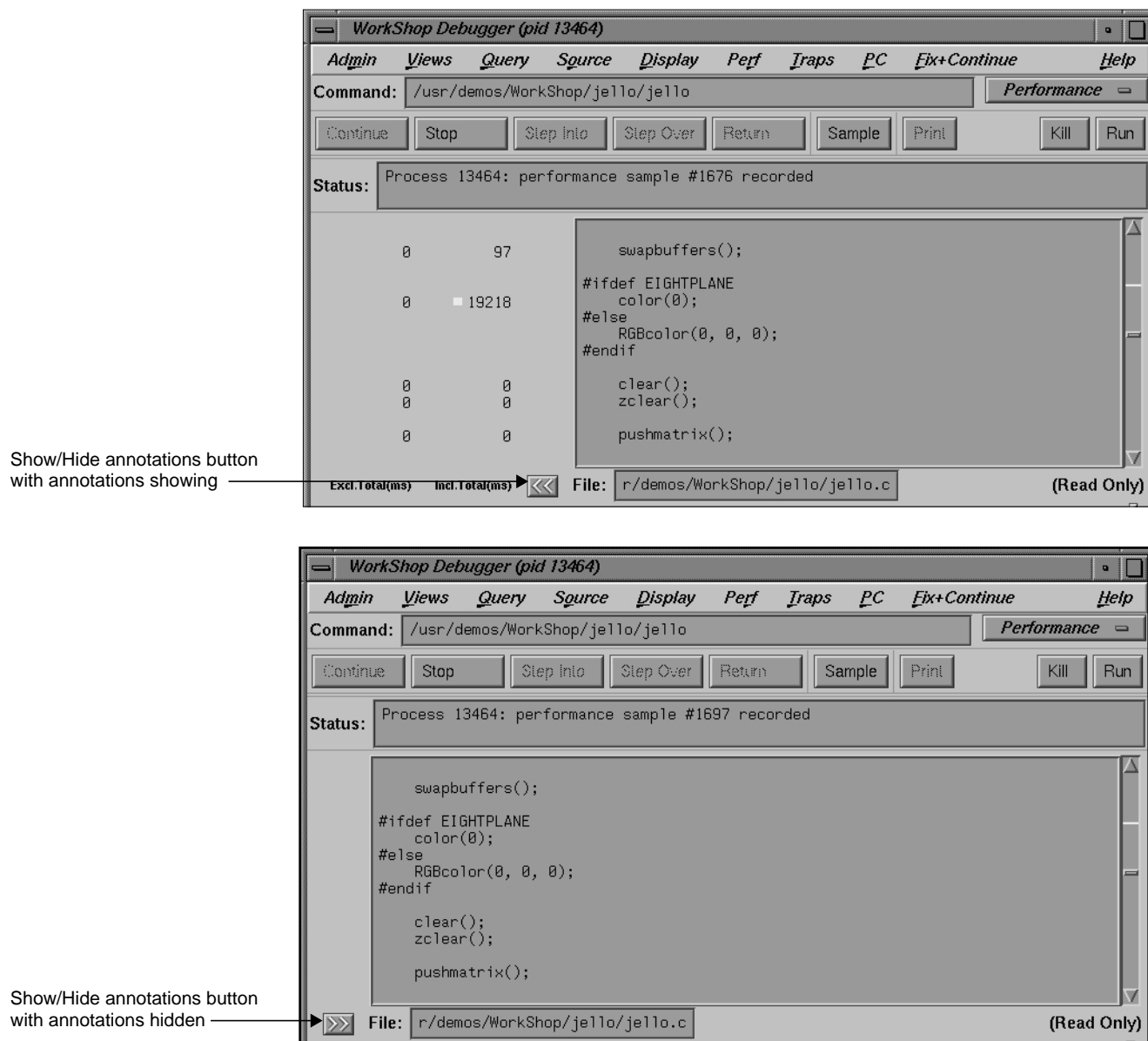


Figure A-2 Show/Hide Annotations Button in Main View

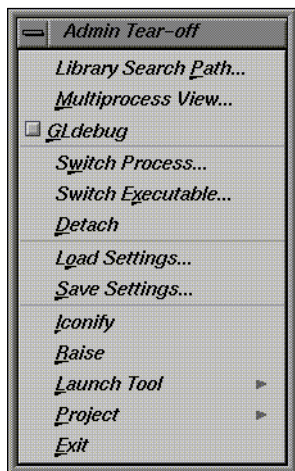


Figure A-3 Admin Menu in Main View

Admin Menu

The Admin menu in Main View performs general management functions dealing with processes, windows, and user preferences (see Figure A-3). The Admin menu provides these 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 box allows you to reset the environment variables `LD_LIBRARY_PATH` and `_RLD_ROOT`. You can also reset `_RLD_LIST` to control the set of DSOs that will be used by the program. See the reference dpage for `rlld` for more information on these variables. Any changes you make to these variables are propagated into the Execution View shell when you run the program.

The Library Search Path dialog is opened automatically when you invoke the Debugger on an executable or core file and the Debugger is unable to find all of the required DSOs. You may also open the Library Search Path dialog box by selecting “Library Search Path...” from the Admin Menu (see Figure A-4). 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.

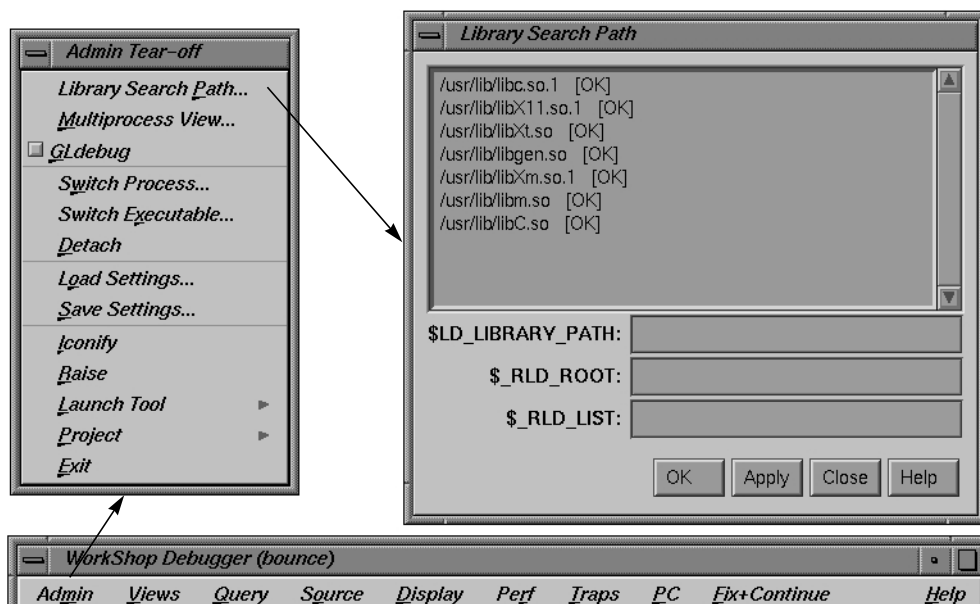


Figure A-4 The Library Search Path Dialog Box

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 helps you debug several processes at once.

“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, as shown in Figure A-5. You can type it in

or paste it from another window, if desired. Switching processes changes the session. If you select a process in Multiprocess View, it is used as the default value.

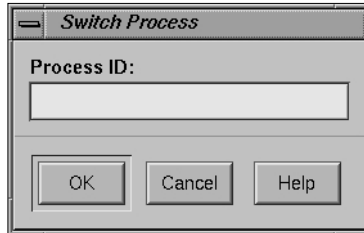


Figure A-5 The Switch Process Dialog Box

“Switch Executable...”

Changes the current executable. This option also lets you debug a different core file. It brings up the dialog box shown in Figure A-6.

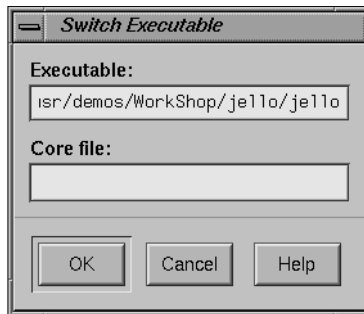


Figure A-6 The Switch Executable Dialog Box

“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. See the description of “Save Settings...”, the following item.

- “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 of the session’s views.
- “Raise”** Brings all the session’s view windows to the foreground and redisplay any iconified windows.
- “Launch Tool”** Lets you run the WorkShop tools. See Figure A-7. 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 buy WorkShop Pro MPF (for multi-process debugging), the “Parallel Analyzer” selection is enabled.



Figure A-7 “Launch Tool” Submenu

- “Project”** Lets you control the WorkShop tools operating on the same executable as a group. See Figure A-8. For more information on “Project View,” a facility for managing CASEVision tools operating on a common target, see “Project View” on page 203.



Figure A-8 "Project" Submenu

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

Views Menu

The Views menu in Main View (see Figure A-9) provides these selections for viewing the process(es) and their corresponding data:

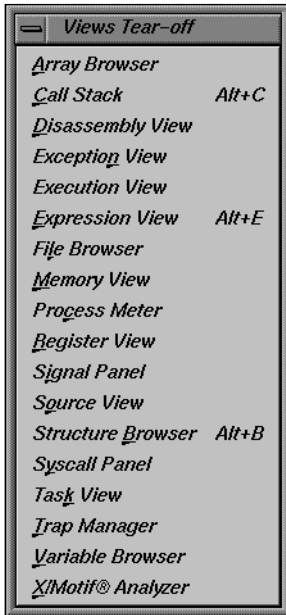


Figure A-9 Views Menu in Main View

"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 a frame in the stack, you can switch the current context to the invocation of that frame and check the state of variables.

"Disassembly View"

Displays assembly code corresponding to the source code.

"Exception View"

Displays the Exception View, and 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. (Used in both the Debugger and Performance Analyzer.)

“Variable Browser”

Displays the 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.

Query Menu

The Query menu (see Figure A-10) 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.

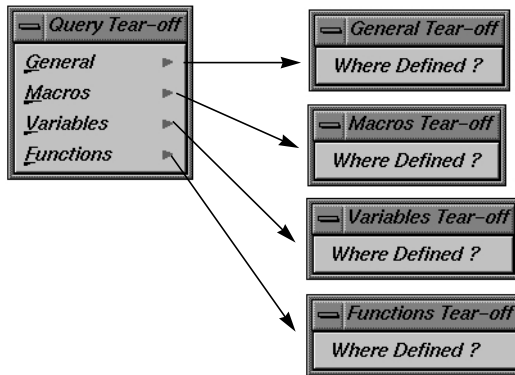


Figure A-10 Query Menu With Submenus

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.

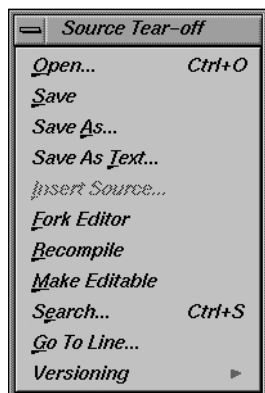


Figure A-11 Source Menu in Main View

Source Menu

The Source menu in Main View (see Figure A-11) provides these selections to deal with 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.
- “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 A-12). 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.

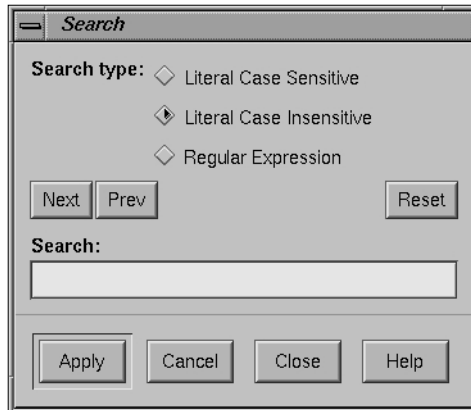


Figure A-12 The Search Dialog Box

“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 A-13.

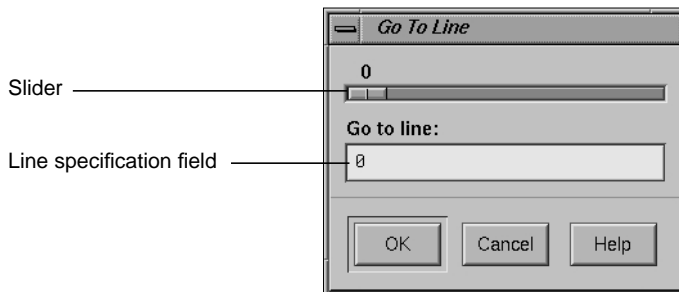


Figure A-13 Go to Dialog Box

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 CASEVision/ClearCase, RCS, or SCCS. Type:

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

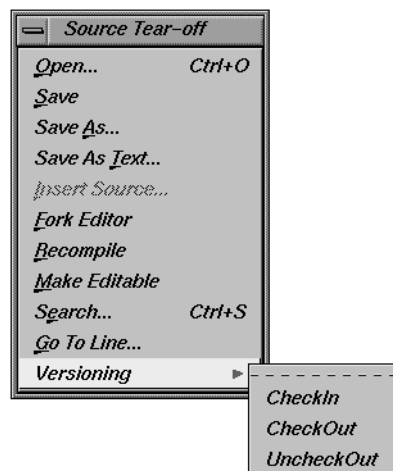


Figure A-14 Versioning Submenu



Figure A-15 Display Menu in Main View

The "Versioning" submenu appears in Figure A-14.

Selecting any of these options displays a shell in which you can access the configuration management tool. The selections in the submenu are:

- "Versioning": "CheckIn": Saves the source file and checks it into the database as a new version.
- "Versioning": "CheckOut": Recalls the source file from the tool's database if you have the proper authority, locks it, and makes it editable.
- "Versioning": "UncheckOut": Cancels the checkout, with no changes registered.

Display Menu

The Display menu in Main View (see Figure A-15) provides these selections to annotate the source code displayed:

"Show Line Numbers"/"Hide Line Numbers"

Displays or hides line numbers in the annotation column corresponding to the source code.

"Preferences..."

Displays the Preferences dialog box (see Figure A-16), 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/MP, you can display and manipulate loop indicators. The Performance Analyzer displays experiment statistics. The Tester module (if purchased) lets you see coverage statistics. Turning off the Performance toggle deletes the performance annotations from the Source View.

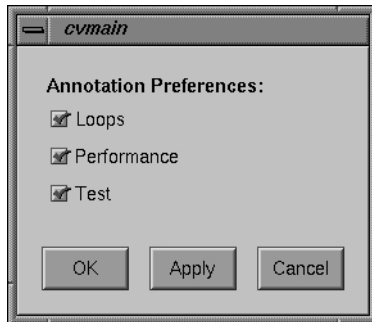


Figure A-16 Preferences Dialog Box

“Hide Icons” / “Show Icons”

Removes or displays the annotation column next to the source code display area.

Perf Menu

The Perf (Performance) menu (see Figure A-17) 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 A-17. You may only select one task per performance analysis run. If none of the given tasks satisfy your requirements, you can choose the “Custom task,” which will bring up the Custom Task dialog, which allows you to design your own task requirements (see Figure A-19).

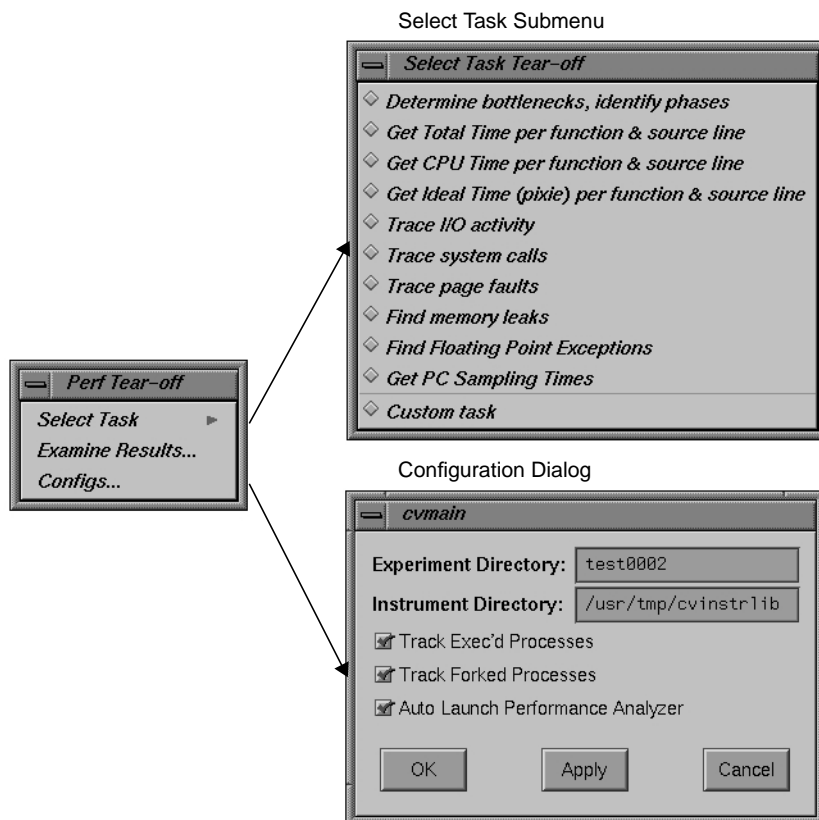


Figure A-17 Perf Menu and Subwindows

Examine Results...

Launches the Performance Analyzer (Figure A-18). For complete information on the Performance Analyzer, see the *Performance Analyzer User's Guide*.

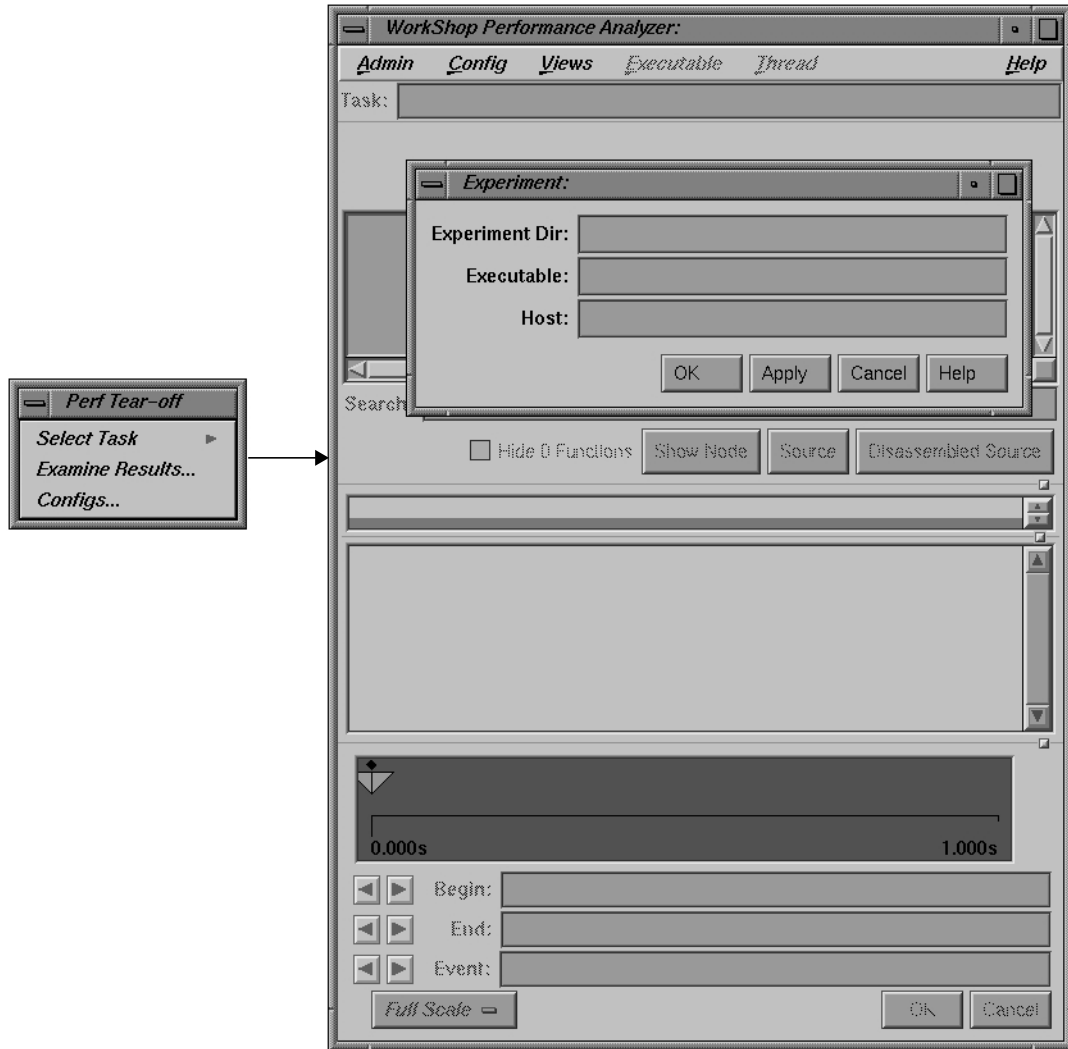


Figure A-18 Launching Performance Analyzer From Perf Menu

Configs...

Brings up the configurations dialog, which contains the following items:

- “Experiment Directory” text field, which allows you to specify the directory where the data captured during

the next experiment is stored. 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.

- “Instrument Directory,” which lets you reuse a previously instrumented executable. This technique avoids the processing necessary for a new instrumentation. Often in a series of experiments, you collect the same type of data while stressing the target executable in different ways. Reusing the instrumented executable lets you do this conveniently.
- “Track Exec’d Processes” toggle, which enables the Executable menu, which will contain selections for any exec’d processes. These selections let you see the performance results for the other executables.
- “Auto Launch Performance Analyzer” toggle, which automatically launches the Performance Analyzer when the experiment is completed.

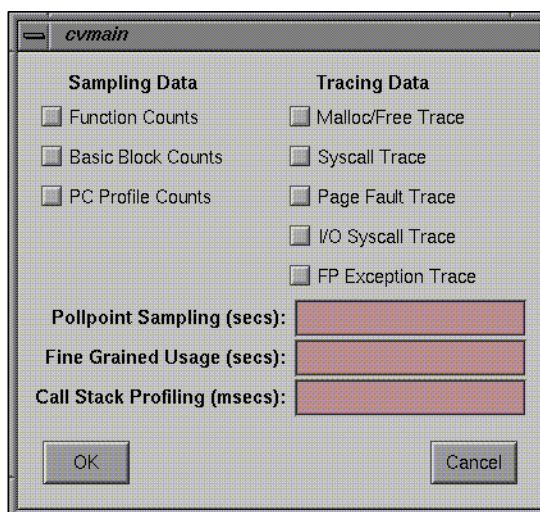


Figure A-19 Custom Task Dialog

The Custom Task dialog contains the following items:

“Sampling Data” toggles

These toggles specify which type of sampling data is collected and recorded during instrumentation. The available choices are

- “Function Counts”
- “Basic Block Counts”
- “PC Profile Counts”

Tracing Data” toggles”

These toggles the type of data recorded at tthe time at which an event of the selected type occurred. The available choices are

- “Malloc/Free Trace”
- “Syscall Trace”
- “Page Fault Trace”
- “I/O Syscall Trace”
- “FP Exception Trace”

“Pollpoint Sampling” text field

Allows you to specify a regular time interval for capturing performance data, including resource usage and any enabled sampling or tracing functions. Pollpoint is best used with call stack data only rather than other profiling data. Its primary utility is to enable you to identify boundary points for phases.

“Fine Grained Usage” text field

Allows you to set a time in to record resource usage data more frequently, at the specified time intervals. Fine grained usage helps you see fluctuations in usage between sample points.

“Call Stack Profiling” text field

Allows you to set the interval for which the the call stack of the target executable is sampled.

For further information on the Performance Analyzer, see the *Performance Analyzer User's Guide*.

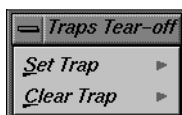


Figure A-20 Traps Menu



Figure A-21 Set Trap Submenu

Traps Menu

The Traps menu (see Figure A-20) offers the following menu selections:

“Set Trap” Allows you to set a trap in your source code. You can set a trap in a number of ways, depending on which selection you make from the submenu (see Figure A-21).

“Set Trap”:“Stop” Allows you to set a stop trap at a designated line in your source code. 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, pull down the “Set Trap” submenu, and select “Stop.”

“Set Trap”:“Stop At Function Entry” Allows you to set a stop trap at the beginning of a function. To set, highlight the function name in the source code display area and select “Set Trap,” then “Stop At Function Entry.”

“Set Trap”:“Stop At Function Exit” Allows you to set a stop trap at the end of a function. To set, highlight the function name in the source code display area and select “Set Trap,” then “Stop At Function Exit.”

“Set Trap”:“Sample” Allows you to set a sample trap at a designated line in your source code. To set a sample 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, pull down the “Set Trap” submenu, and select “Sample.”

“Set Trap”:“Sample At Function Entry” Allows you to set a sample trap at the beginning of a function. To set, highlight the function name in the source code display area and select “Set Trap,” then “Sample At Function Entry.”



Figure A-22 Clear Trap Submenu

“Set Trap”:“Sample At Function Exit”

Allows you to set 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.”

“Clear Trap”

Deletes the 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 (see Figure A-22).

“Clear Trap”:“Stop”

Designates the “Stop” trap type.

“Clear Trap”:“Sample”

Designates the “Sample” trap type.

PC Menu

The PC (program counter) menu in Main View (see Figure A-23) provides these 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.

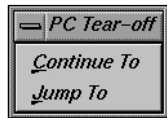


Figure A-23 PC Menu in Main View

Fix+Continue Menu

The Fix+Continue menu (see Figure A-24) 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...” popup menu in the Admin menu of the Status window. See “Fix+Continue Status Window” on page 255 for further information.

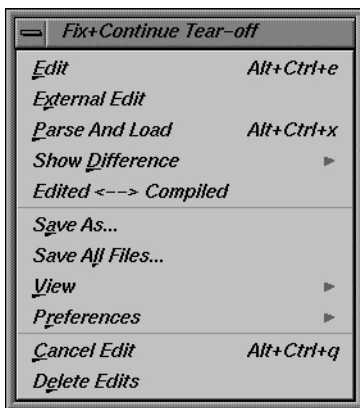


Figure A-24 Fix+Continue Menu

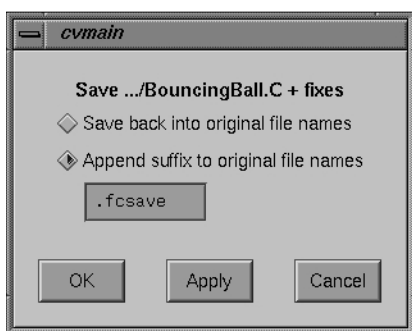


Figure A-25 "Save File+Fixes As..." Popup Window

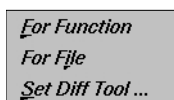


Figure A-26 Show Difference Submenu

"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.

"Show Difference" submenu

Allows you to see the difference between the original code and your modifications. See "Show Difference Submenu" on page 155 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 (see Figure A-25). You can save the changes to the current source file (the default), or to a separate file.

"Save All Files..."

Launches the "Save File+Fixes As..." dialog (see Figure A-25), which 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 and Continue supports status, message, and build environment windows. See "View Submenu" on page 156 for further information.

"Preferences" submenu

Allows you to set your Fix+Continue preferences. See "Preferences Submenu" on page 156 for further information.

"Cancel Edit" Takes you out of edit mode.

"Delete Edits" Deletes any changes that you made to functions.

Show Difference Submenu

This submenu (see Figure A-26) allows you to see the difference between the 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 Preference dialog (see Figure A-29), which allows you to set the tool that displays the differences between the two sets of code. The default is *xdiff*. For further information on the Preference dialog, see “Preferences Submenu” on page 156.

View Submenu

This submenu (see Figure A-27) allows you to open different Fix+Continue view windows. It contains the following options:

“Status Window”

Launches the Fix+Continue Status window. See “Fix+Continue Status Window” on page 255 for more information.

“Message Window”

Launches the Fix+Continue Message window. See “Fix+Continue Message Window” on page 260 for more information.

“Build Environment Window”

Launches the Fix+Continue Build Environment window. See “Fix+Continue Build Environment Window” on page 262 for more information.



Figure A-27 View Submenu

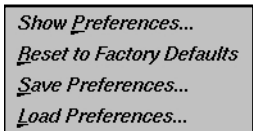


Figure A-28 Preferences Submenu

Preferences Submenu

The Preference Menu (see Figure A-28) allows you to set various options for the Fix and Continue environment, such as the difference tool, the external editor command, and so on. The menu contains the following options:

“Show Preferences”

Launches the Preference dialog (see Figure A-29), which displays the preferences that are currently enabled for the session, and allows you to change the settings.

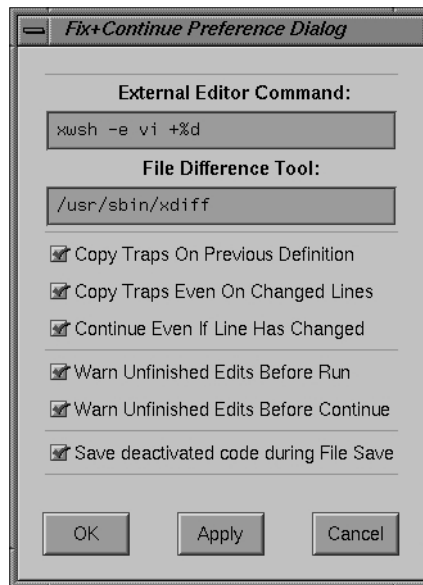


Figure A-29 Fix+Continue Preferences Dialog

The preferences available through the dialog are

- “External Editor Command” text field, which allows you to choose your text editor. The default is *vi*.
- “File Difference Tool” text field, which allows you to choose the tool that you 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 that 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 F&C will look at the switch.

- “Copy Traps Even On Changed Lines” toggle, which causes the debugger to copy traps onto a mapped line.
- “Continue Even If Line Has Changed” toggle. When you edit and parse 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, which pops up a warning dialog before a Run if you have unfinished edits.
- “Warn Unfinished Edits Before Continue” toggle, which 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 won’t get compiled. You can manually edit the source to use the old definition in any way you desire.

“Reset Factory Defaults”

Sets the preferences to the installed defaults.

“Save Preferences”

Allows you to save your preferences to a file. This item brings up the File dialog. See Figure A-139.

“Load Preferences...”

Allows you to load preferences from a file. This item brings up the File dialog. See Figure A-139.

Keyboard Accelerators

Use the accelerators in Table A-1 to issue Fix+Continue commands directly from the keyboard. The accelerators are listed alphabetically by command.

Table A-1 Fix and Continue Keyboard Accelerators

Command	Ctrl + key
Cancel Edit	U
Edit	E
External Edit	X
Parse And Load	P

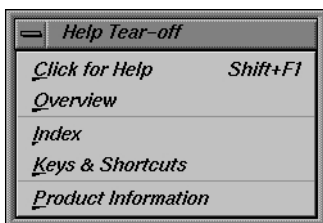


Figure A-30 Help Menu

Help Menu

The Help menu (see Figure A-30) provides these options:

- “Click for Help” Provides information on the selected window’s feature.
- “Overview” Provides overview 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 general copyright and version number information on the current tool.

Basic Windows

This section discusses some of the basic additional views that are available through the Debugger; the Execution View, Source View, and Process Meter.

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 A-31) is launched automatically with the Debugger.

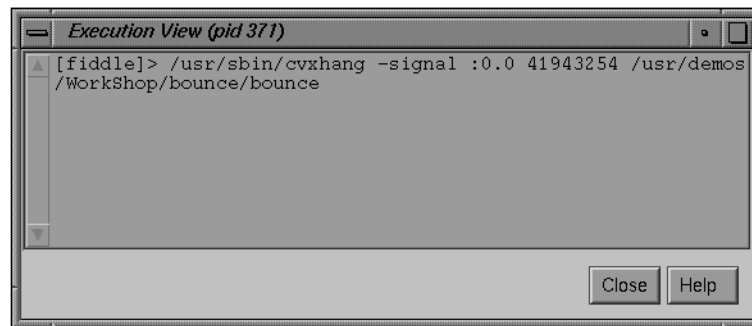


Figure A-31 Execution View

Source View

The Source View (see Figure A-32) displays your source, opening your *Main* program file by default.

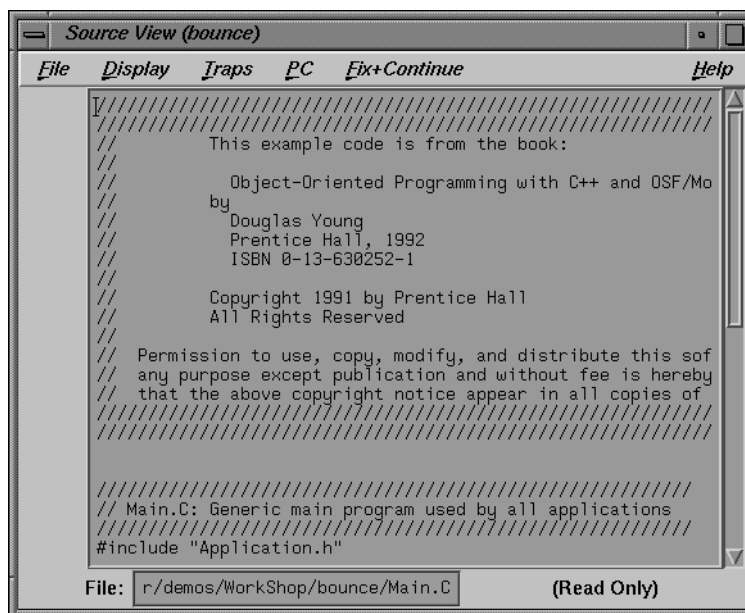


Figure A-32 Source View

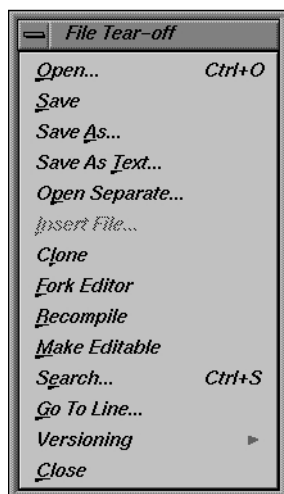


Figure A-33 Source View File Menu

Menu Bar

The Source View menu bar contains five items that are 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 “Main View” on page 132). The only new menu bar item is the file menu (Figure A-33), described below:

- “Open...” Launches the file dialog (see Figure A-139), allowing you to choose a file to load into the source view.
- “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.
- “Save As Text...” records the information in the display area as a text file.

- “Open Separate...” launches the File dialog, allowing 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, which lets you compile the source code associated with the current executable.
- “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 A-12). 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.
- “Go To Line...” launches the Go To Line dialog, which 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 CASEVision/ClearCase from SGI, RCS or SCCS. Type:

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

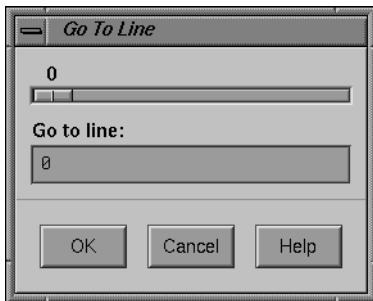


Figure A-34 Go To Line Dialog

Selecting any of these options displays a shell in which you can access the configuration management tool. The selections in the submenu are:

- “Versioning” : “CheckIn”
saves the source file and checks it into the database as a new version.
- “Versioning” : “CheckOut”
recalls the source file from the tool’s database if you have the proper authority, locks it, and makes it editable.
- “Versioning” : “UncheckOut”
cancels the checkout, with no changes registered.
- “Close”
Dismisses the Source View window.

Process Meter

The Process Meter monitors the resource usage of a running process without saving the data. Figure A-35 shows the Process Meter in its default configuration (with only the User/Sys Time chart active).

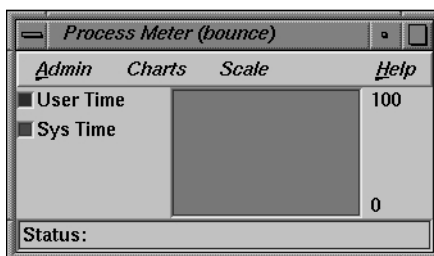


Figure A-35 Process Meter

The Process Meter contains its own menu bar, which contains the Admin, Charts, Scale, and Help menus. The Admin menu is the same as that described in “Admin Menu” on page 167. The Help menu is the same as that described in “Help Menu” on page 159. The other menus are described in the following sections.

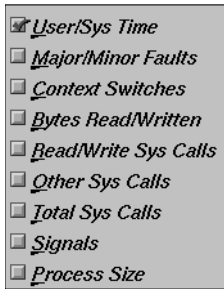


Figure A-36 Process Meter Charts Menu

Charts Menu

The Charts Menu contains a set of toggles that allow you to choose which charts are displayed in the Process Meter. You can display as many charts simultaneously as you wish. The choices available are:

- 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

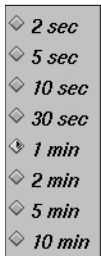


Figure A-37 Process Meter Scale Menu

Scale Menu

The Scale Menu is a radio button panel that allows you to set the time scale for the processes displayed in the Process Meter. The choices available are:

- 2 seconds
- 5 seconds
- 10 seconds
- 30 seconds
- 1 minute (the default)
- 2 minutes
- 5 minutes
- 10 minutes

Ada-specific Windows

This section discusses the Debugger windows that are specific to Ada: the Task View and Exception View.

Task View

The Task View is an Ada-specific view that provides you with task and callstack information. If you do not have Ada installed on your system, the Task View menu item in the Views menu will be grayed-out.

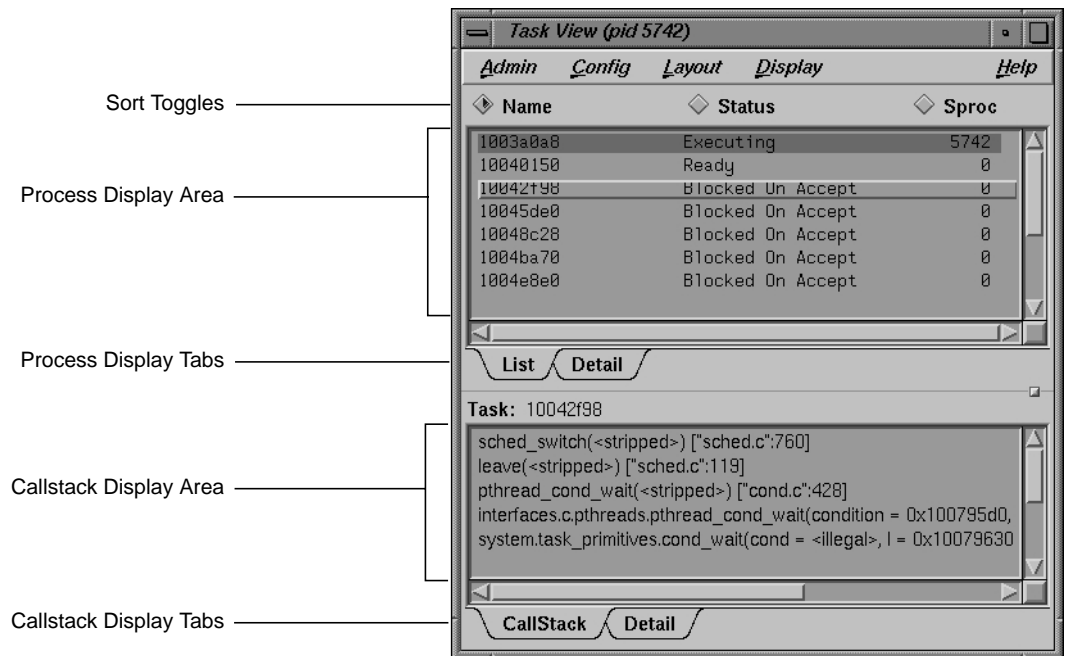


Figure A-38 Task View

The Task View contains its own menu bar, which contains the Admin, Config, Display, and Help menus. The Help menu is the same as that described in "Help Menu" on page 159. The other menus are described in the following sections.

In addition, the Task View contains the following items:

Process Sort toggles

Allow you to sort the process list in one of three ways, depending on which of the following radio buttons are active:

- "Name"
- "Status"
- "Sproc"

Process Display tabs

Allows you to view either a list of the tasks, or the details of the currently running (and highlighted) task. (See Figure A-39.)

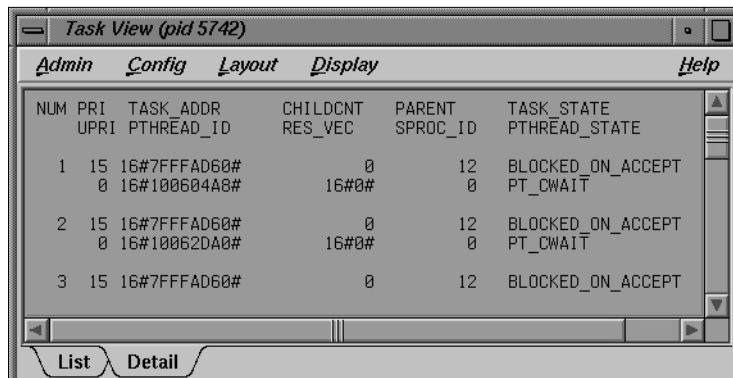


Figure A-39 Task View Process Detail View

Task Display tabs

Allows you to view the callstack information, or the callstack details of the currently selected process. (See Figure A-40.)



Figure A-40 Task View Callstack Detail View

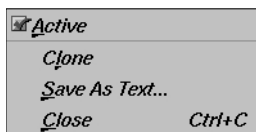


Figure A-41 Task View Admin Menu

Admin Menu

The Admin Menu (see Figure A-41) contains the following items:

“Active” toggle

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 (see Figure A-49). This dialog allows you to save your current session as text in a file you designate.

“Close”

Closes the current window.

Config Menu

The Config Menu (Figure A-42) contains the following items:

“Preferences...”

launches the preference dialog (Figure A-42), which allows you the option of setting the maximum depth of the Task View.

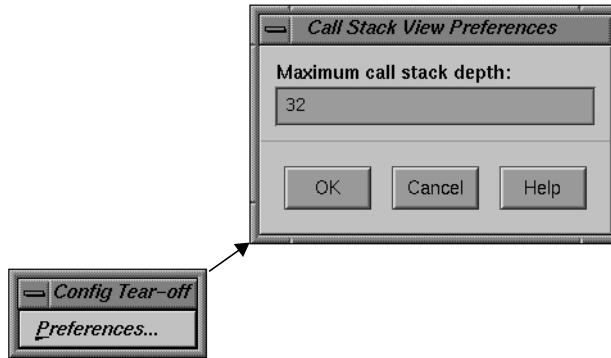


Figure A-42 Task View Config Menu

Layout Menu

The Layout Menu (Figure A-41) contains the following toggles:



Figure A-43 Task View Layout Menu

- “Task List” Causes only the Callstack Display to be shown.
- “Single Task” Causes only the Process Display to be shown.

Display Menu

The Display Menu (Figure A-41) 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 toggles control what radio buttons are made available in the toggle sort list, as well as what information is displayed in the Process Display area.

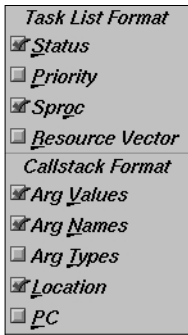


Figure A-44 Task View Display Menu

The Task View Display menu contains the following toggles:

- “Status” Displays the status of the process. This toggle is active by default.
- “Priority” Displays the priority of the process.
- “Sproc” Displays the sproc value of the process. This toggle is active by default.
- “Resource Vector” Displays the resource vector value of the process.

“Arg Values”	Allows you to set the argument values in the Task View. This toggle is active by default.
“Arg Names”	Allows you to set the argument names in the Task View. This toggle is active by default.
“Arg Types”	Allows you to set the argument types in the Task View.
“Location”	Allows you to set the function location in the Task View. This toggle is active by default.
“PC”	Allows you to set the PC in the Task View.

Exception View

The Exception View is an Ada-specific view that allows you to set traps on exceptions. This view only functions if Ada is installed.

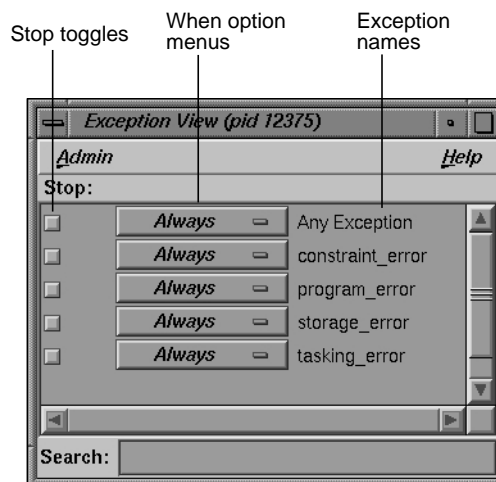


Figure A-45 Exception View

The Exception View contains the following items:

Admin menu Contains the following items:

- “Active” toggle: 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 (see Figure A-49). This dialog allows you to save your current session as text in a file you designate.
- “Close”: Closes the current window.

“Stop” toggle Indicates when a trap is active.

“When” option menu

Allows you to select when an exception trap fires. Contains the following choices:

- “Always:” stop any time the exception is raised.
- “Catch-All:” stop when caught by a catchall rather than an explicit handler, or when unhandled.
- “Unhandled:” stop when the exception is unhandled.

The process is always stopped at the point of a raise.

“Search” text field

Allows you to search for an exception.

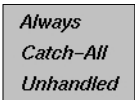


Figure A-46 “When”
Exception Option Menu

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 graphics contexts, that might be difficult or impossible to inspect using ordinary debugger functionality.

To access the X/Motif analyzer window, you must pull down the Views menu and select "X/Motif Analyzer" (see Figure A-47).

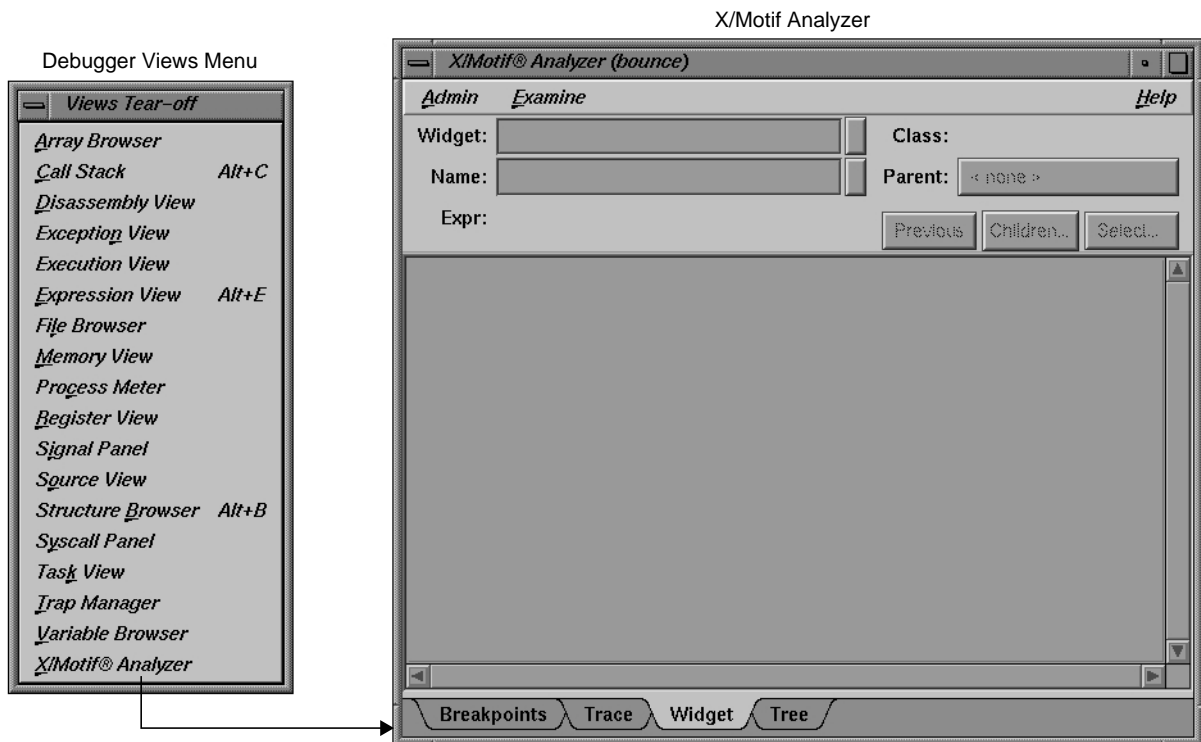


Figure A-47 Launching the X/Motif Analyzer

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 examiner windows available are

- "Breakpoints Examiner"
- "Trace Examiner"
- "Widget Examiner"
- "Tree Examiner"
- "Callback Examiner"
- "Window Examiner"
- "Event Examiner"
- "Graphics Context Examiner"
- "Pixmap Examiner"
- "Widget Class Examiner"

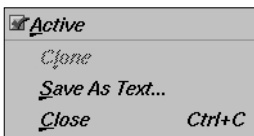


Figure A-48 Admin Menu

Admin Menu

The Admin menu (see Figure A-48) offers the following menu selections:

"Active" toggle

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 (see Figure A-49). This dialog allows you to save your current session as text in a file you designate.

"Close"

Closes the current window.

<i>S</i> election	<i>Alt+S</i>
<i>W</i> idget	<i>Alt+W</i>
<i>W</i> idget <i>T</i> ree	
<i>W</i> idget <i>C</i> lass	
<i>W</i> indow	
<i>X</i> Event	
<i>X</i> Graphics Context	
<i>X</i> Pixmap	

Figure A-50 Examine Menu

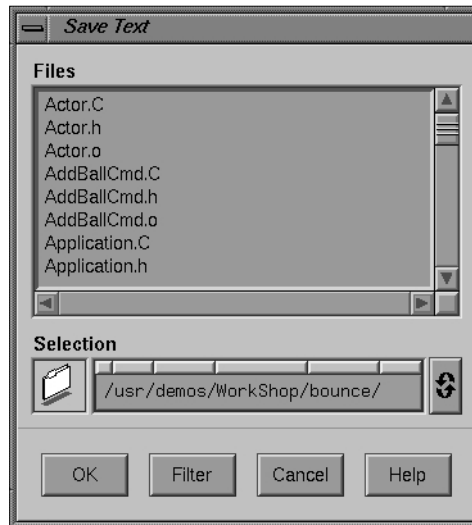


Figure A-49 "Save Text" Dialog

Examine Menu

The Examine menu (see Figure A-50) offers the following menu selections:

- "Selection" Selects the currently highlighted object for examination.
- "Widget" Uses the current selection as input to the widget examiner, then opens that examiner (see "Widget Examiner" for information).
- "Widget Tree" Switches the window view to the widget tree examiner (see "Tree Examiner" for information).
- "Widget Class" Switches the window view to the widget class examiner (see "Widget Class Examiner" for information).
- "Window" Switches the window view to the window examiner (see "Window Examiner" for information).
- "X Event" Switches the window view to the X Event examiner (see "Event Examiner" for information).

“X Graphics Context”

Switches the window view to the X graphics context examiner (see “Graphics Context Examiner” for information).

“X Pixmap”

Switches the window view to the X pixmap examiner (see “Pixmap Examiner” for information).

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 A-51).



Figure A-51 Examiner Tabs

When first launched, the X/Motif Analyzer has only four tabs: Breakpoints, Trace, Widget, and Tree. As you use new 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” (see Figure A-52). The initial four tabs may not be removed.

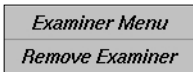


Figure A-52 Removing Tabs

Return Button

The “Widget” and “Name” text fields both have return buttons (see Figure A-53) just to their right. Clicking these buttons causes the X/Motif Analyzer to respond exactly as if you had pressed **Return** on your keyboard.

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 A-53):

- The widget specification area, which 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.

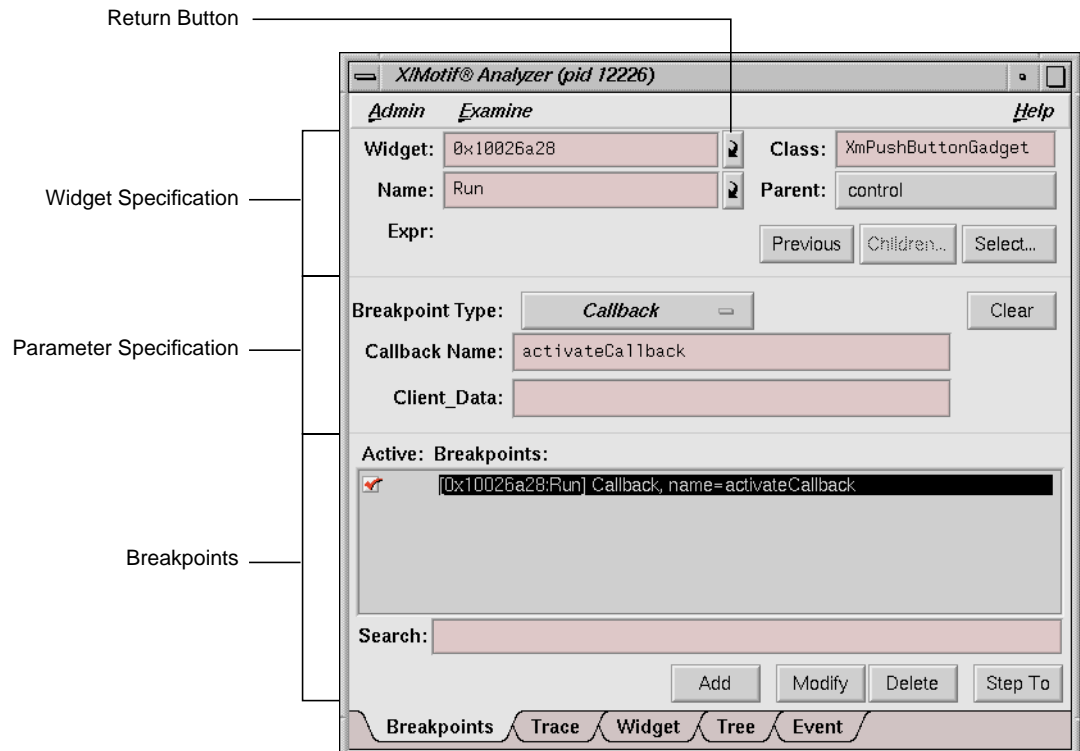


Figure A-53 Breakpoints Examiner

The control area has eight different breakpoint types that it can examine. These types are set through the "Breakpoint Type" option button (see

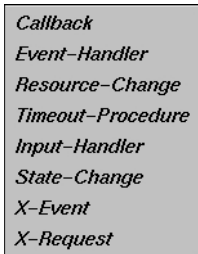


Figure A-54 Breakpoint Type Option Button

Figure A-54). The options for the “Breakpoint Type” option button are described below:

- “Callback:” Widget callback installed by **XtAddCallback**. Parameters include callback name and **client_data** *XtPointer* value. See “Callback Breakpoints Examiner” on page 176 for more information.
- “Event-Handler:” Widget event handler installed by **XtAddEventHandler**. Parameters include X event type and **client_data** *XtPointer* value. See “Event-Handler Breakpoints Examiner” on page 178 for more information.
- “Resource-Change:” Resource change caused by **XtSetValues** or **XtVaSetValues**. Parameters include resource name and resource value, both strings. See “Resource-Change Breakpoints Examiner” on page 180 for more information.
- “Timeout-Procedure:” Timeout callback installed by **XtAppAddTimeOut**. Parameters include **client_data** *XtPointer* value. See “Timeout-Procedure Breakpoints Examiner” on page 182 for more information.
- “Input-Handler:” Input callback installed by **XtAppAddInput**. Parameters include **client_data** *XtPointer* value. See “Input-Handler Breakpoints Examiner” on page 184 for more information.
- “State-Change:” Various widget state changes (for example, “managed” or “realized”). Parameters include widget state. See “State-Change Breakpoints Examiner” on page 185 for more information.
- “X-Event:” X event received by target application. Parameters include X event type. See “X-Event Breakpoints Examiner” on page 188 for more information.
- “X-Request:” X request received by target application. Parameters include X request type. See “X-Request Breakpoints Examiner” on page 189 for more information.

Callback Breakpoints Examiner

When the “Callback” option of the “Breakpoint Type” option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure A-55.

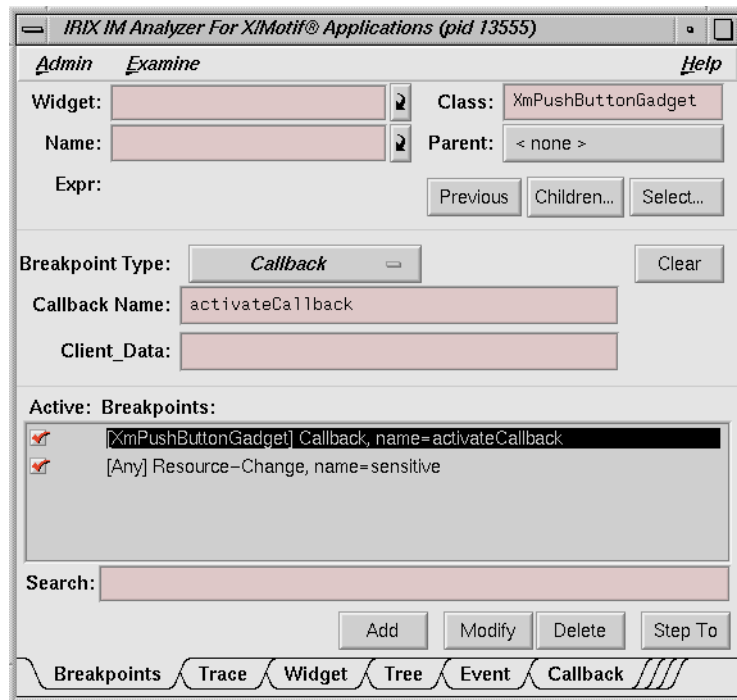


Figure A-55 Callback Breakpoints Examiner

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 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.
- "Callback Name" text field Allows you to set the name of the callback for the breakpoint.
- "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.

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 A-56.

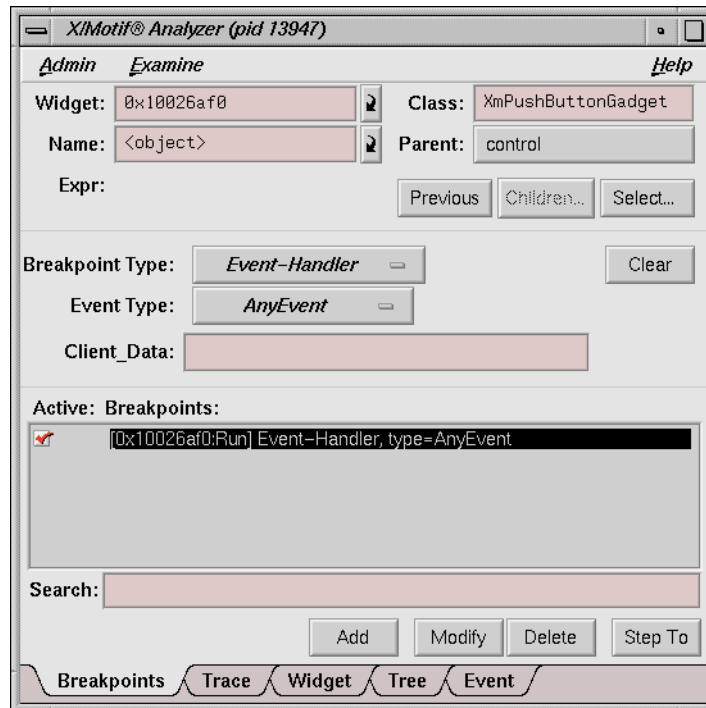


Figure A-56 Event-Handler Breakpoints Examiner

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 button

Allows you to move the parent of the currently selected widget.

<i>AnyEvent</i>
<i>KeyPress</i>
<i>KeyRelease</i>
<i>ButtonPress</i>
<i>ButtonRelease</i>
<i>MotionNotify</i>
<i>EnterNotify</i>
<i>LeaveNotify</i>
<i>FocusIn</i>
<i>FocusOut</i>
<i>KeymapNotify</i>
<i>Expose</i>
<i>GraphicsExpose</i>
<i>NoExpose</i>
<i>VisibilityNotify</i>
<i>CreateNotify</i>
<i>DestroyNotify</i>
<i>UnmapNotify</i>
<i>MapNotify</i>
<i>MapRequest</i>
<i>ReparentNotify</i>
<i>ConfigureNotify</i>
<i>ConfigureRequest</i>
<i>GravityNotify</i>
<i>ResizeRequest</i>
<i>CirculateNotify</i>
<i>CirculateRequest</i>
<i>PropertyNotify</i>
<i>SelectionClear</i>
<i>SelectionRequest</i>
<i>SelectionNotify</i>
<i>ColormapNotify</i>
<i>ClientMessage</i>
<i>MappingNotify</i>

Figure A-57 Event Type Option Button

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. The types available are shown in Figure A-57.

“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.

Resource-Change Breakpoints Examiner

When the “Resource-Change” option of the “Breakpoint Type” option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure A-58.

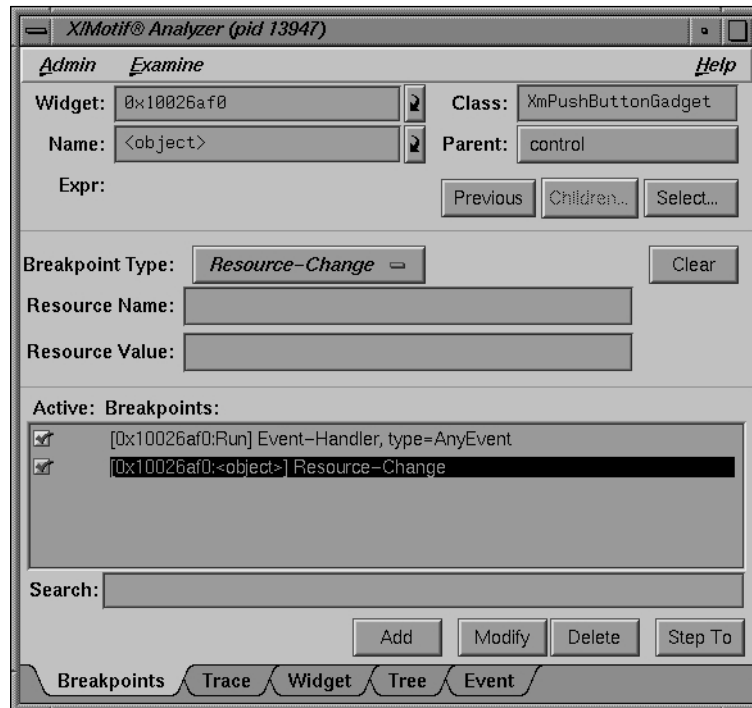


Figure A-58 Resource-Change Breakpoints Examiner

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.
- "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.

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 A-59.

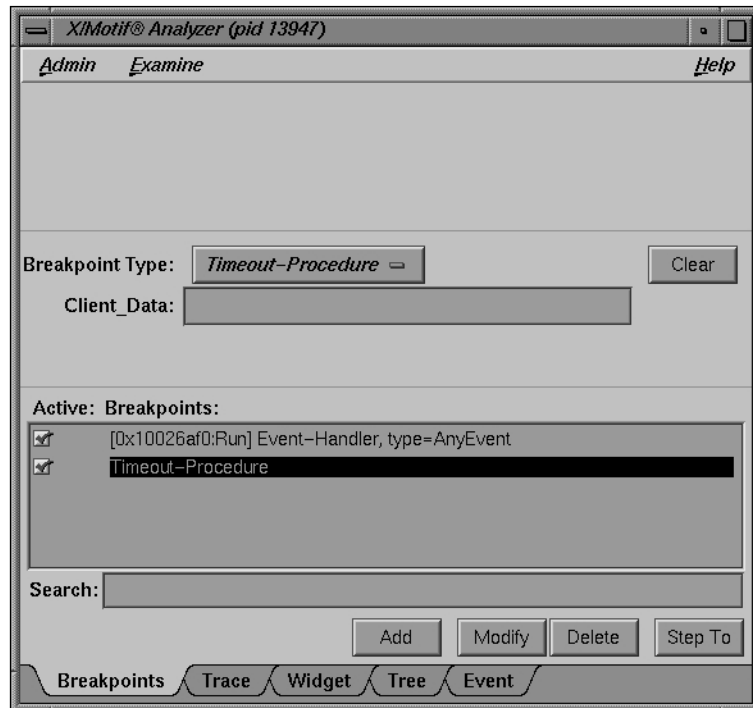


Figure A-59 Timeout-Procedure Breakpoints Examiner

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.

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 A-60.

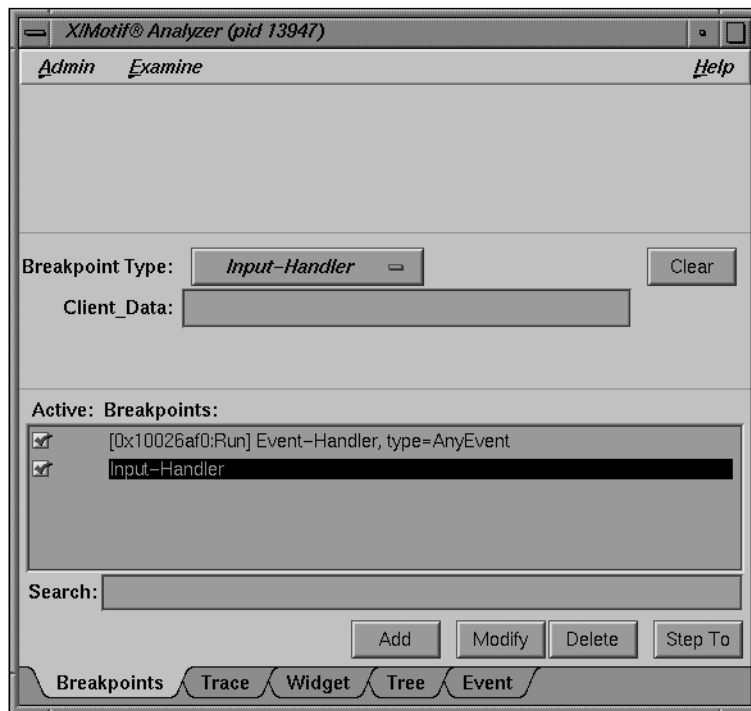


Figure A-60 Input-Handler Breakpoints Examiner

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.

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 A-61.

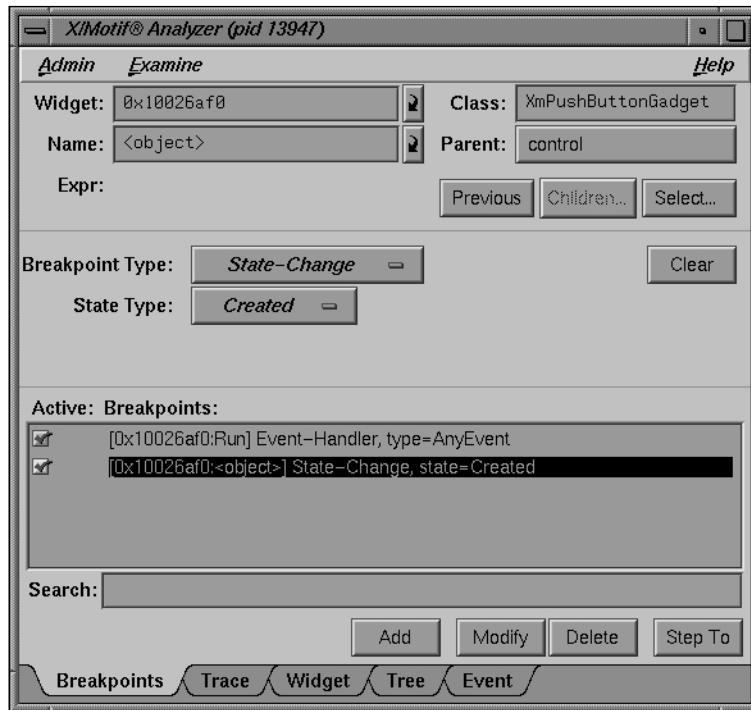


Figure A-61 State-Change Breakpoints Examiner

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. The types available are as follows (see Figure A-62):

- Created
- Destroyed
- Managed
- Realized
- Unmanaged
- Any

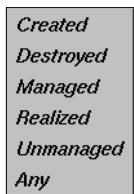


Figure A-62 State Type
Option Button

"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.

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 A-63.

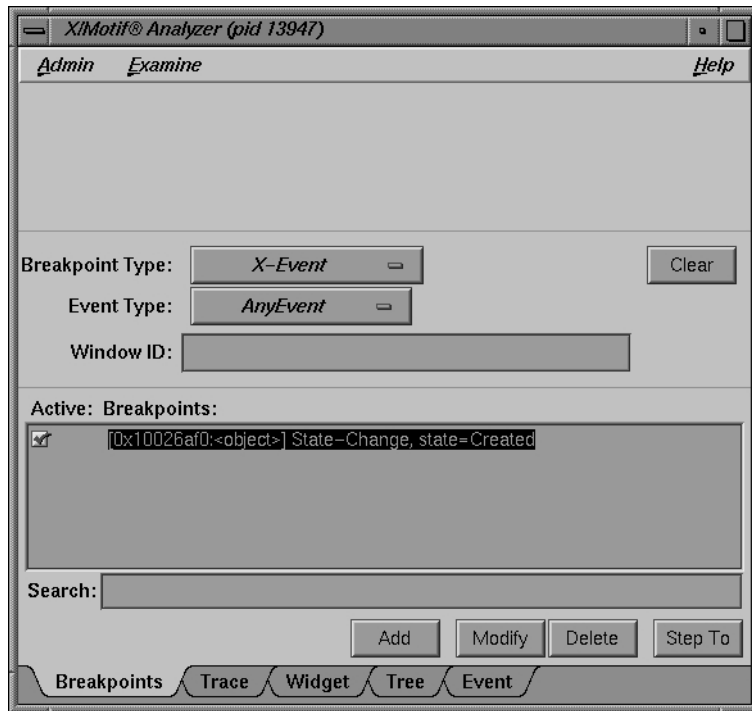


Figure A-63 X-Event Breakpoints Examiner

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.

<i>AnyEvent</i>
<i>KeyPress</i>
<i>KeyRelease</i>
<i>ButtonPress</i>
<i>ButtonRelease</i>
<i>MotionNotify</i>
<i>EnterNotify</i>
<i>LeaveNotify</i>
<i>FocusIn</i>
<i>FocusOut</i>
<i>KeymapNotify</i>
<i>Expose</i>
<i>GraphicsExpose</i>
<i>NoExpose</i>
<i>VisibilityNotify</i>
<i>CreateNotify</i>
<i>DestroyNotify</i>
<i>UnmapNotify</i>
<i>MapNotify</i>
<i>MapRequest</i>
<i>ReparentNotify</i>
<i>ConfigureNotify</i>
<i>ConfigureRequest</i>
<i>GravityNotify</i>
<i>ResizeRequest</i>
<i>CirculateNotify</i>
<i>CirculateRequest</i>
<i>PropertyNotify</i>
<i>SelectionClear</i>
<i>SelectionRequest</i>
<i>SelectionNotify</i>
<i>ColormapNotify</i>
<i>ClientMessage</i>
<i>MappingNotify</i>

Figure A-64 Event Type Option Button

“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. The types available are shown in Figure A-64.

“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.

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 A-65.

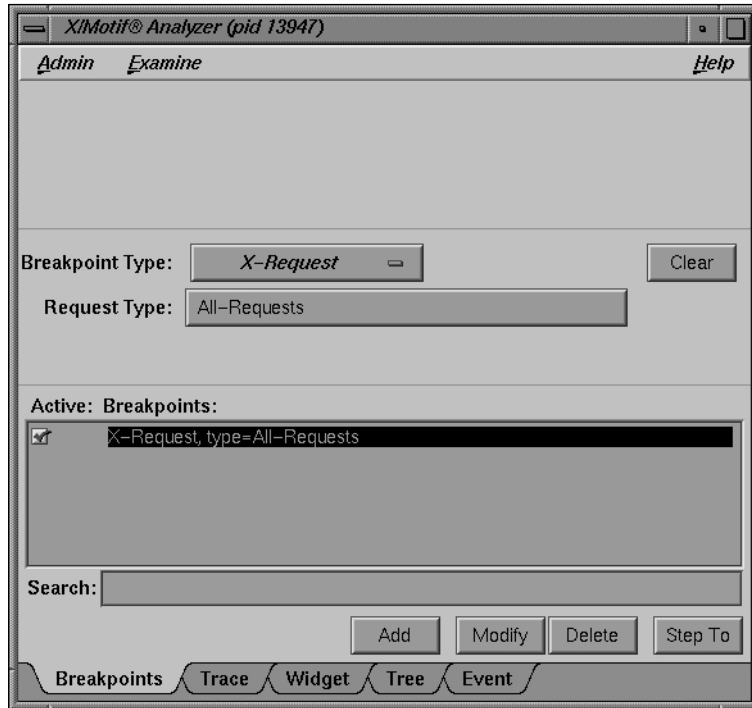


Figure A-65 X-Request Breakpoints Examiner

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 A-66). 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.

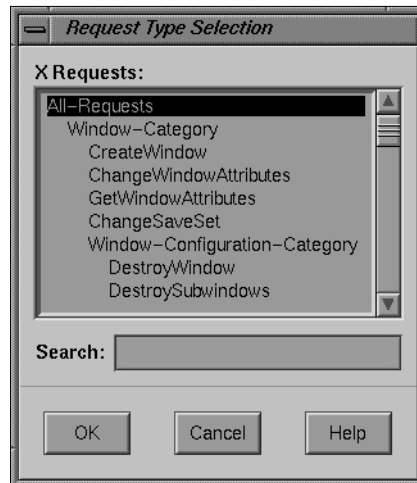


Figure A-66 “Request Type Selection” Dialog

“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.

Trace Examiner

The Trace examiner (see Figure A-67) 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, unmanage)
- Xt Callbacks (widget, event handler, work proc, timeout, input, signal)

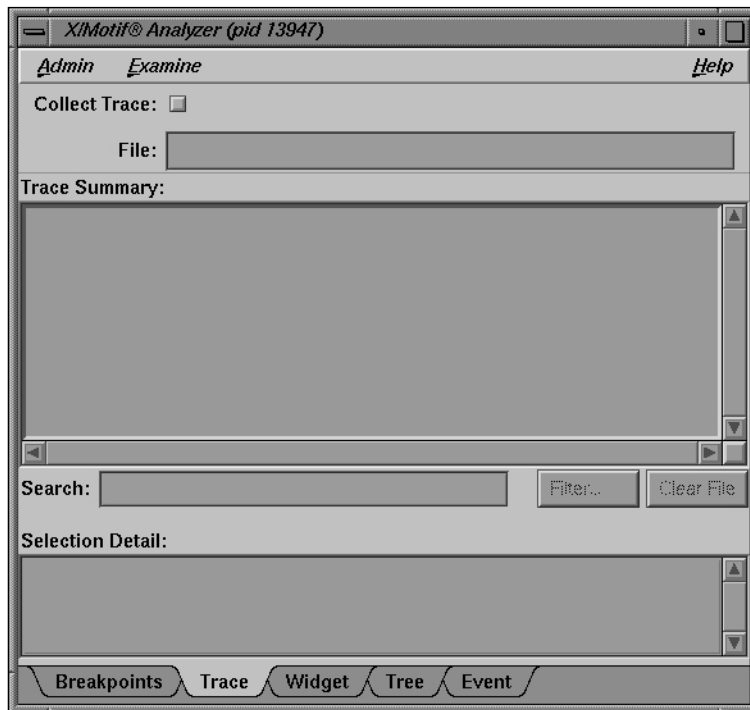


Figure A-67 Trace Examiner

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, textual 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.

Widget Examiner

The Widget examiner (see Figure A-68) displays the internal Xt widget structure, as well as the Xt inheritance implementation using nested C structs.

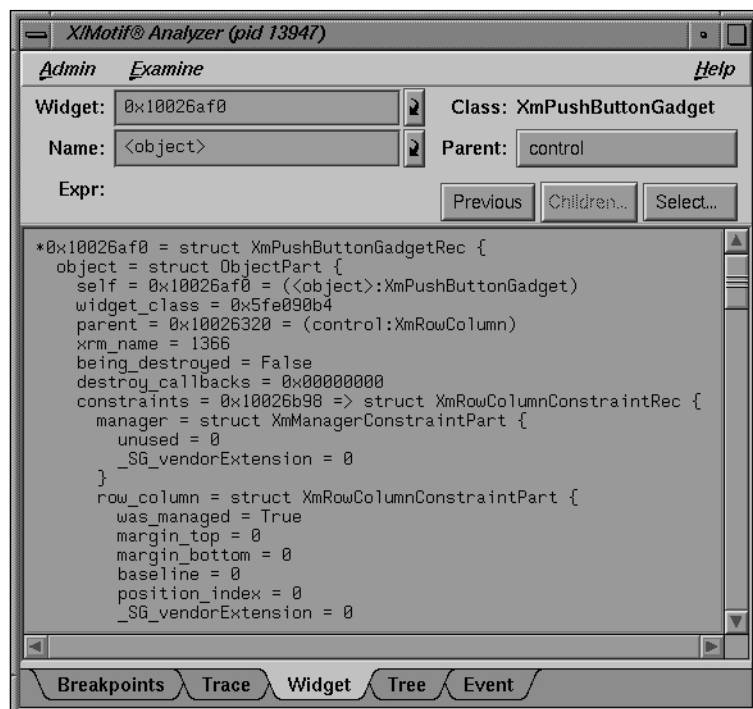


Figure A-68 Widget Examiner

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.

Tree Examiner

The Tree examiner (see Figure A-69) displays the widget hierarchy.

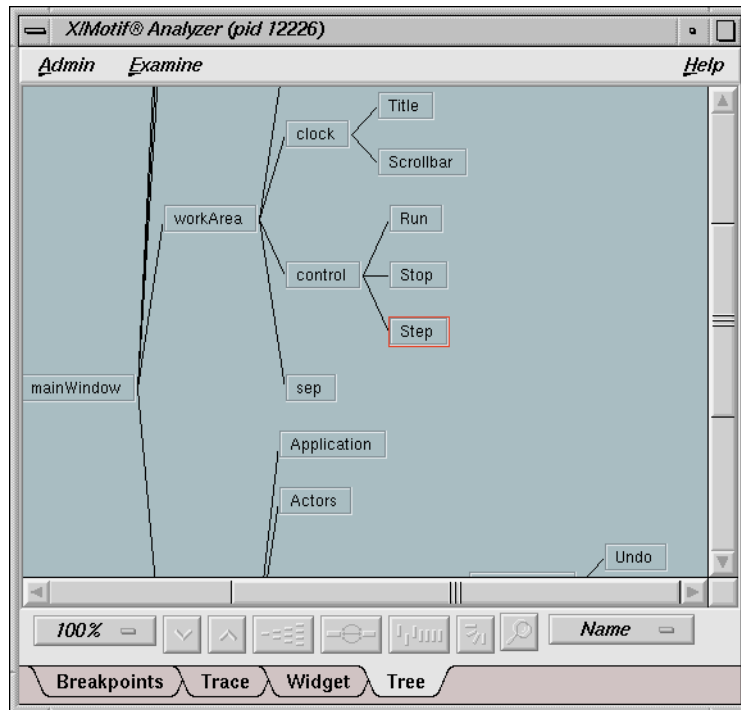


Figure A-69 Tree Examiner

You may double-click a node to view that widget in the Widget examiner. Use the option menu in the bottom-right corner to switch the display among widget names, class names, and IDs.

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.

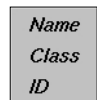


Figure A-70 Widget View Type Option Button

You may display the tree according to widget name, class, or ID value. You can select this by choosing the appropriate option from the widget view type option button (see Figure A-70) in the lower-right portion of the examiner.

Callback Examiner

The Callback examiner (see Figure A-71) 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.

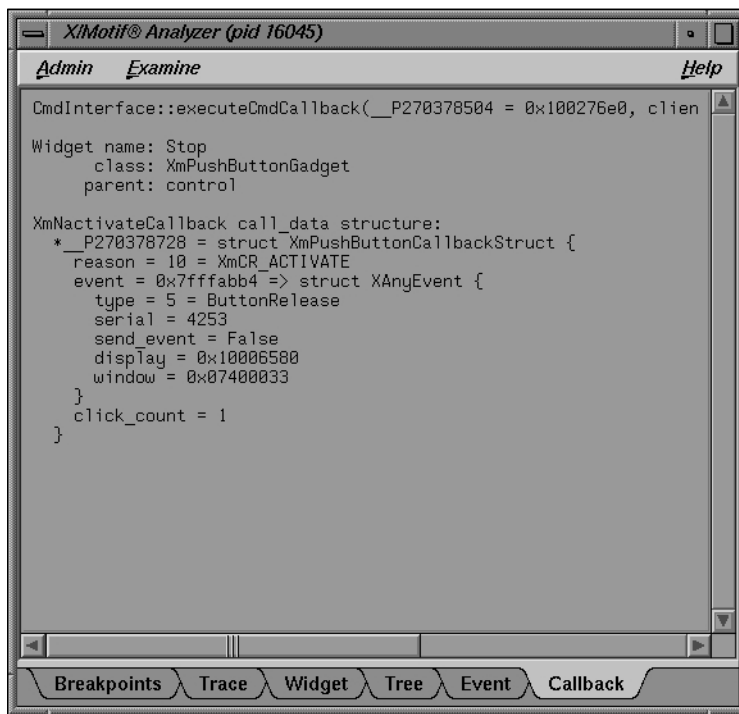


Figure A-71 Callback Examiner

Window Examiner

The Window examiner (see Figure A-72) 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.

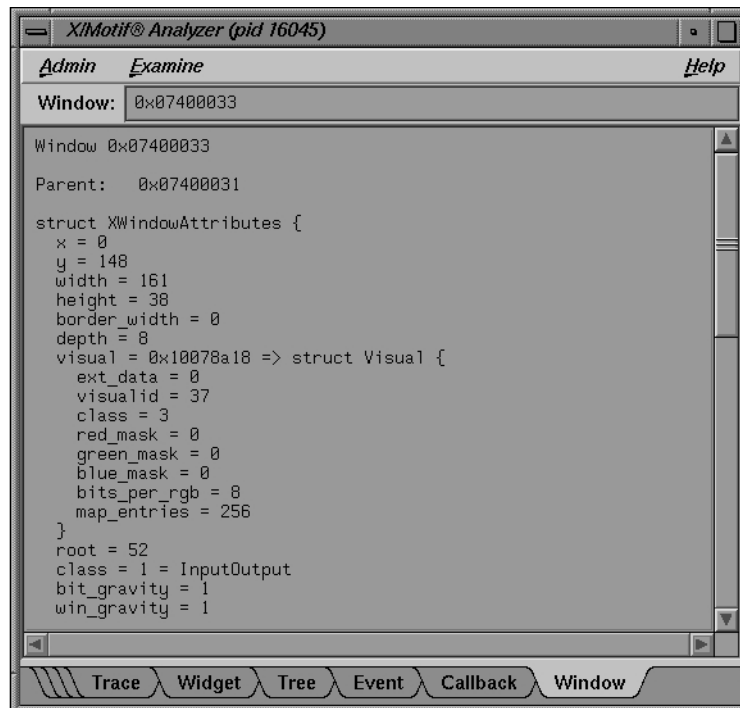


Figure A-72 Window Examiner

The Window examiner contains the “Window” text field, which displays the address of the window that is being examined. You may change to a different window by entering a new address and pressing the **ENTER** key.

Event Examiner

The Event examiner (see Figure A-73) displays the event structure for an XEvent pointer. The proper XEvent union member is used, and enums and masks are decoded.

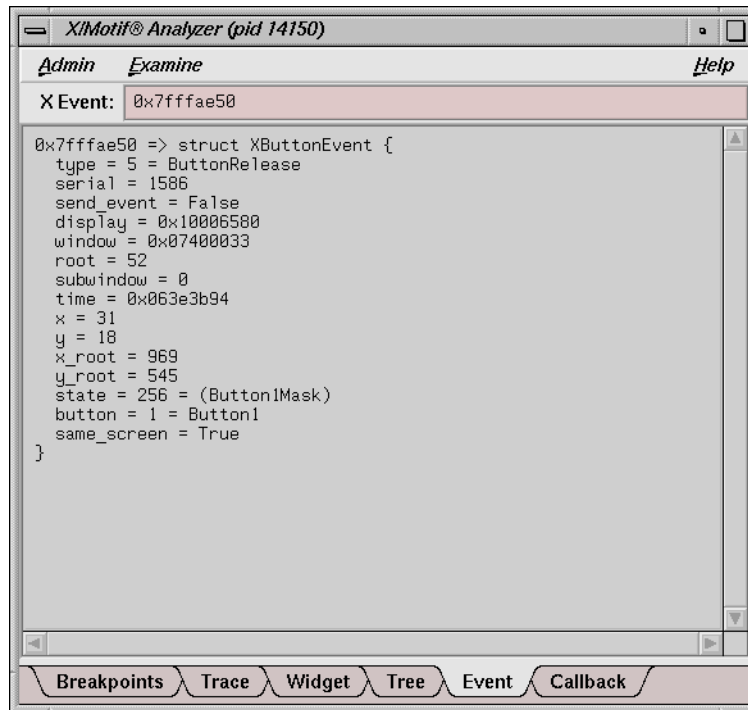


Figure A-73 Event Examiner

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 the **ENTER** key.

Graphics Context Examiner

The Graphics Context examiner (see Figure A-74) displays the X graphics context attributes that are cached by Xlib in the form of an XGCValues structure. Enums and masks are decoded.

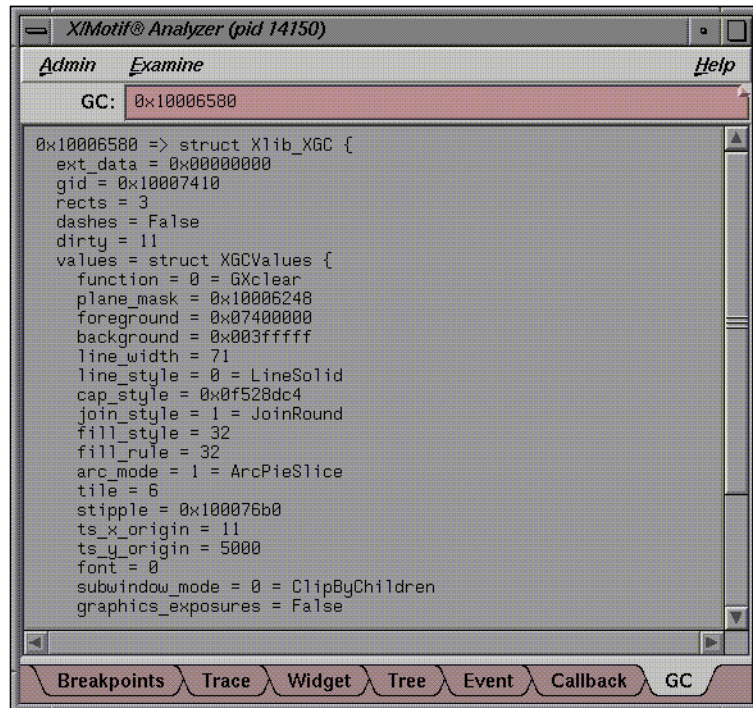


Figure A-74 Graphics Context Examiner

The Graphics Context examiner contains the “GC” text field, which 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 the **ENTER** key.

Pixmap Examiner

The Pixmap examiner (see Figure A-75) 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.

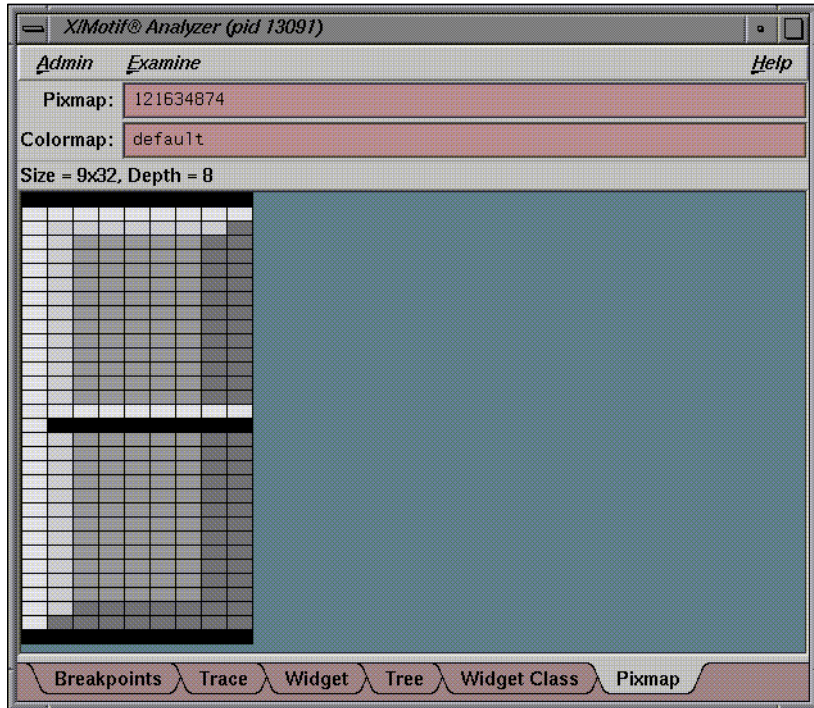


Figure A-75 Pixmap Examiner

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.

Widget Class Examiner

The Widget Class examiner (see Figure A-76) displays the X_t widget class structure, as well as the X_t inheritance implementation using nested C structs.

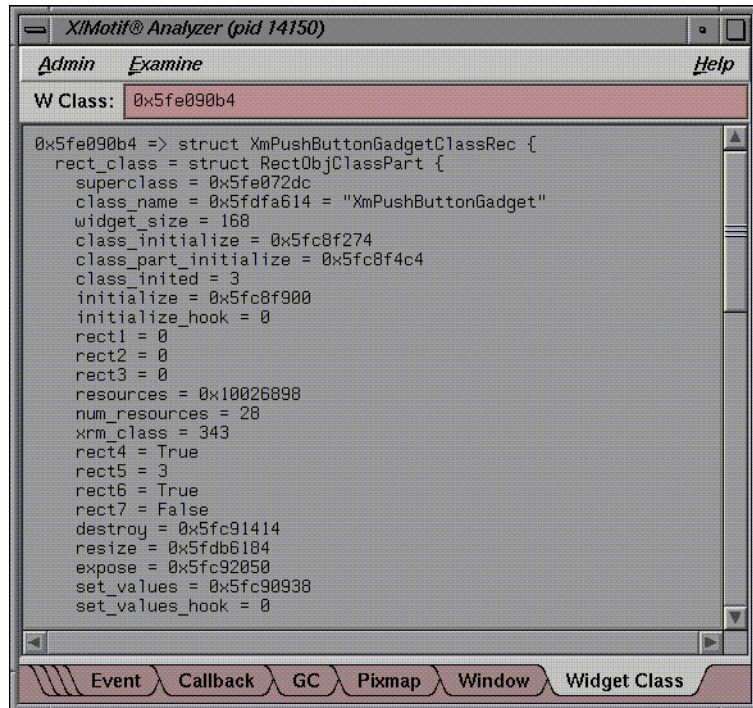


Figure A-76 Widget Class Examiner

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 the **ENTER** key.

Project Session Management Windows

A project is a single CASEVision work session focused on a common task. Its purpose is to let you perform operations conveniently on selected project components or on the project as a whole. A project includes all CASEVision tools and windows invoked from the command line (unless invoked with the `-privateProject` switch) and from the graphical user interface of one of those tools.

When you are working on a single project, with only a few windows open simultaneously, you can keep track of your activities with relative ease. In situations where you have many windows open from one or more projects, it is easy to get confused. To simplify the use of multiple windows, WorkShop provides facilities for iconifying and raising windows in

- Main View Admin menu
- the “Project” submenu
- the Project View window

Figure A-77 shows where these facilities are located.

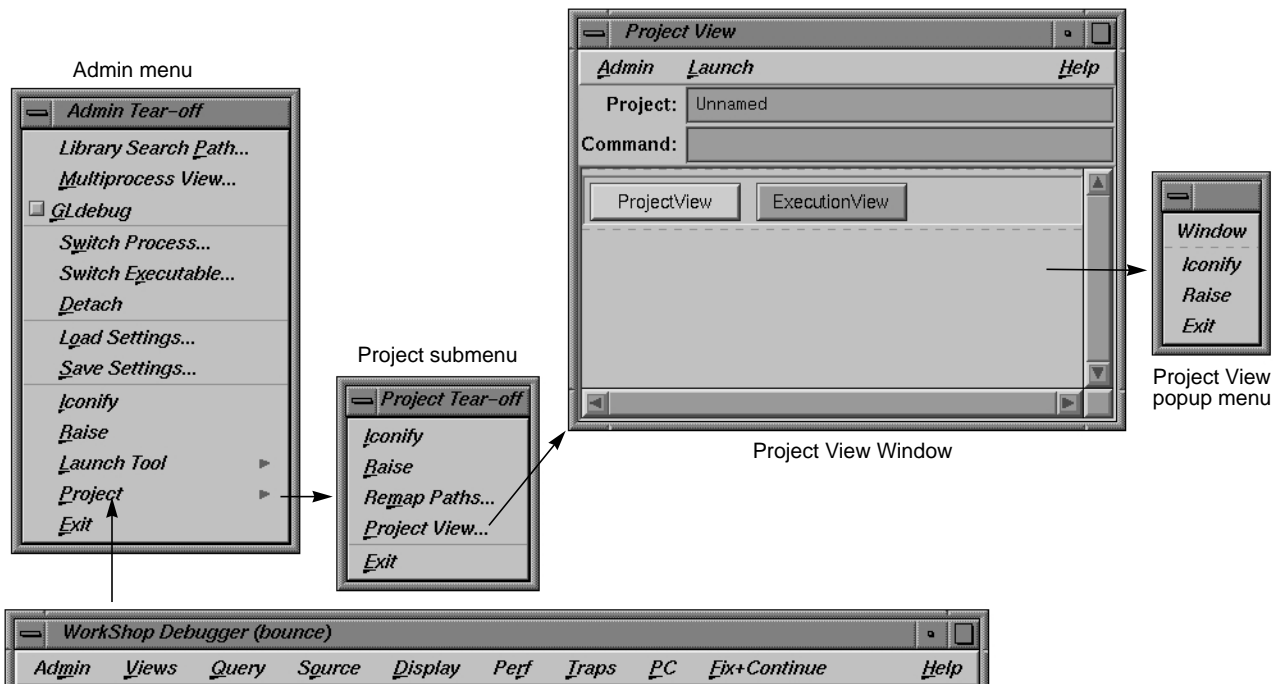


Figure A-77 Iconify and Raise Facilities

The window facilities in the Main View Admin menu apply to the current application and its windows only. “Iconify” iconifies all of the application’s views. “Raise” brings all the application’s view windows to the foreground and redisplays any iconified windows.

The “Project” submenu selections “Iconify,” “Raise,” “Remap Paths...,” and “Exit” operate the same way as their counterparts do in the MainView Admin menu, except that they are applied to all tools and windows in the current project.

Project View

To display the Project View window, in the Main View, pull down the Admin menu, select the “Project” submenu, and select “Project View...” The Project View window is shown in Figure A-78 with its Admin menu and right-button popup menu. Project View represents the components of a project (tools or windows depending on the toggles in the Admin menu) as buttons. Elements from the same project are grouped within a rectangle. A dashed-line rectangle indicates the currently selected project. When a project is selected, you can change its name or change the command.

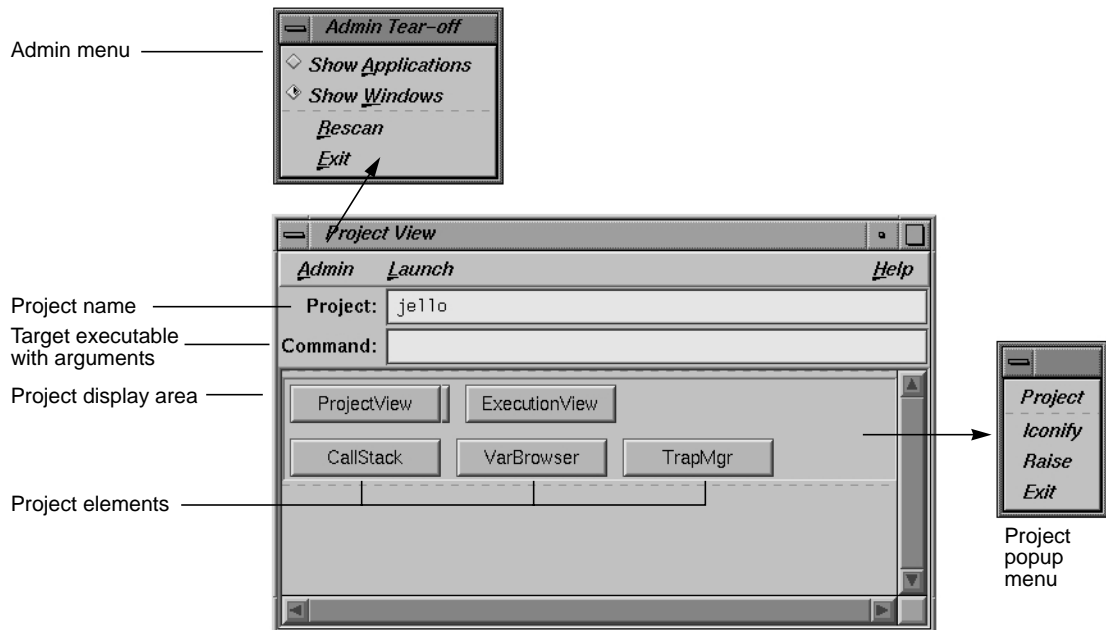


Figure A-78 Project View Window with Menus

Project View Admin Menu

The “Show Applications” and “Show Windows” toggle selections 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 redisplay the current elements. “Exit” closes the Project View window.

Project View Text Fields

The *Project* field lets you enter a name to identify the current project.

The *Command* field lets you invoke other tools to be included in the project. These can be WorkShop tools or integrated tools, such as CASEVision/ClearCase.

Project Display Area

The elements of a project are represented by buttons. When a button protrudes from the screen, the item is currently iconified; when it is recessed, the item is displayed. Clicking the button toggles it between display and iconify modes.

Project Popup Menu

When you hold down the right mouse button inside a project rectangle, the Project popup menu displays. It lets you “Iconify,” “Raise,” or “Quit” the item under the cursor or all items in the project as a whole, if the cursor is within the rectangle but not over an item.

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
- Signal Panel
- Syscall Panel

Trap Manager

The Trap Manager allows you to set, edit, and manage traps (used in both the Debugger and Performance Analyzer). The Trap Manager is shown in Figure A-79.

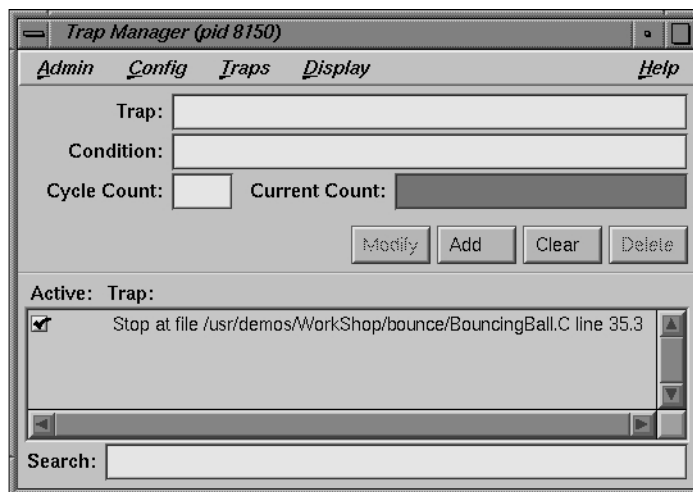


Figure A-79 Trap Manager

The Trap Manager 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, textual search for the trap list.

The Trap Manager has a menu bar which contains the Admin, Config, Traps, Display, and Help menus. The Admin menu is the same as that described in “Admin Menu” on page 167. The Help menu is the same as that described in “Help Menu” on page 159. The other menus are described in the following sections.

Config Menu

The Config Menu (Figure A-80) contains the following items:

“Load Traps...”

Brings up the File dialog (see Figure A-139), allowing you to load the traps from a file.

“Save Traps...”

Brings up the File dialog (see Figure A-139), allowing you to save the current traps to a file.



Figure A-80 Trap Manager
Config Menu



Figure A-81 Trap Manager
Traps Menu

Traps Menu

The Traps Menu (Figure A-81) allows you to set traps under a number of conditions. These conditions are:

- “At Source Line”
- “Entry Function”
- “Exit Function”
- “Stop Trap Default”
- “Sample Trap Default”
- “Group Trap Default”
- “Stop All Default”



Figure A-82 Trap Manager Display Menu

Display Menu

The Display Menu (Figure A-82) contains the following items:

“Delete All” Deletes all traps from the trap list.

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 A-83.



Figure A-83 Signal Panel

The Signal Panel contains an Admin menu (described in “Admin Menu” on page 167) and a Help menu (described in “Help Menu” on page 159). Each signal trigger trap in the display has a toggle associated with it. In addition, the panel has a Search field.

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 A-84.

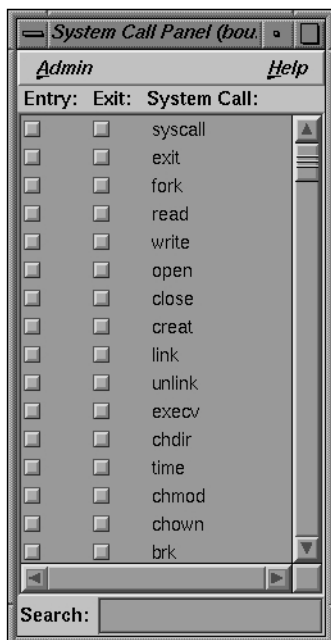


Figure A-84 Syscall Panel

The Syscall Panel contains an Admin menu (described in "Admin Menu" on page 167) and a Help menu (described in "Help Menu" on page 159). 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.

Data Examination Windows

There are several windows that are used primarily to examine your debugging data:

- “Array Browser”
- “Call Stack View”
- “Expression View”
- “File Browser”
- “Structure Browser”
- “Variable Browser”

Array Browser

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. 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. (For a tutorial example of the Array Browser, see “Examining Data” on page 34.)

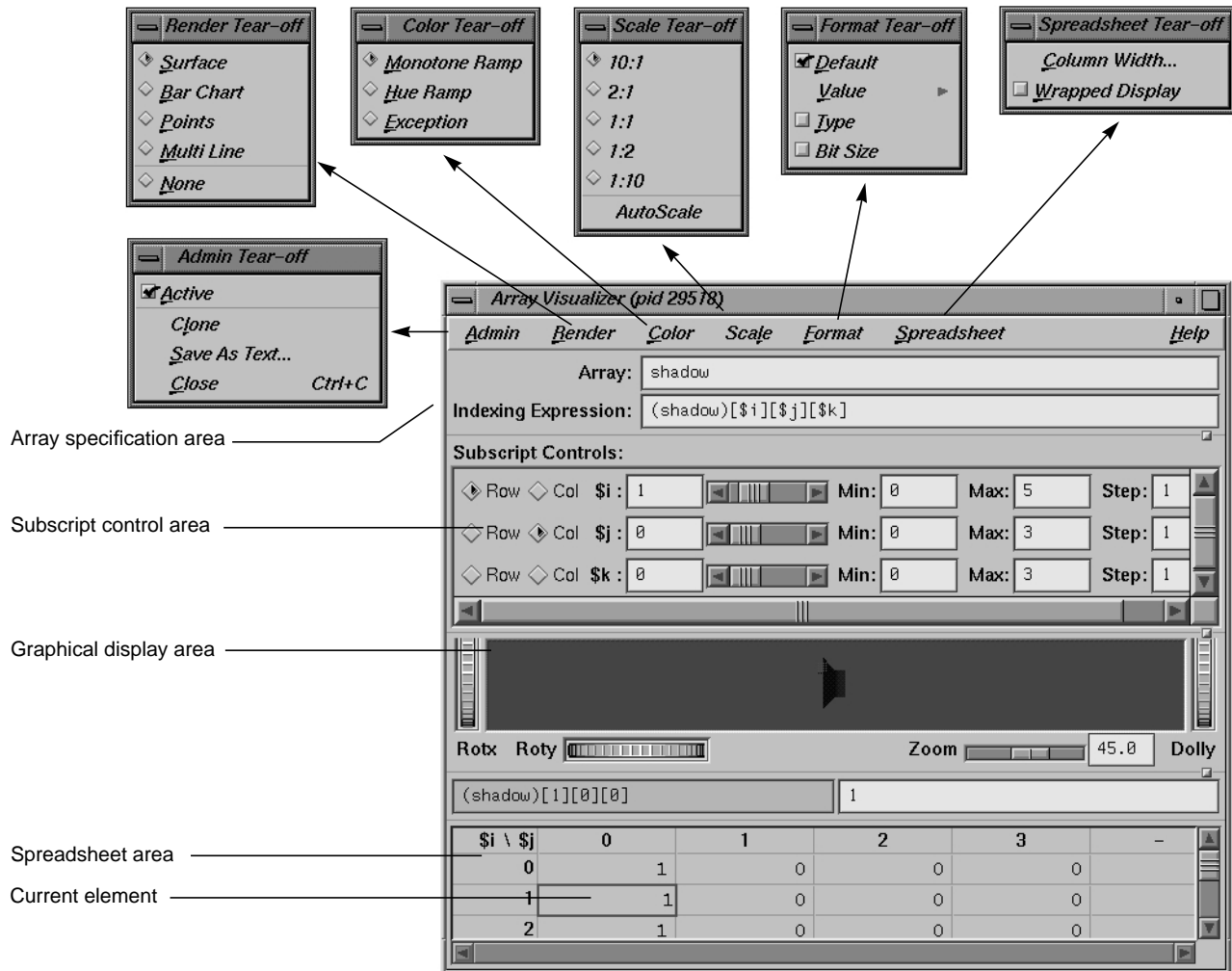


Figure A-85 Array Browser With Display Menu Options

The *array specification area* lets you specify the variable and its dimensions. It consists of these 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 control area* serves two functions: (1) it lets you control which elements in the variable are to be displayed, and (2) 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 A-86.

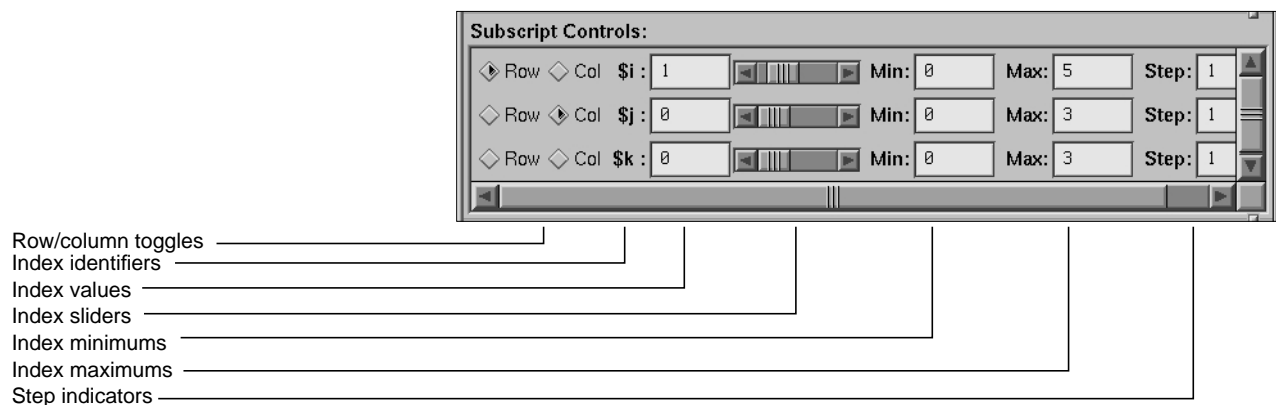


Figure A-86 Subscript Control Area in Array Browser

The subscript control area provides these 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 th 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 A-87 shows a close-up of the spreadsheet area.

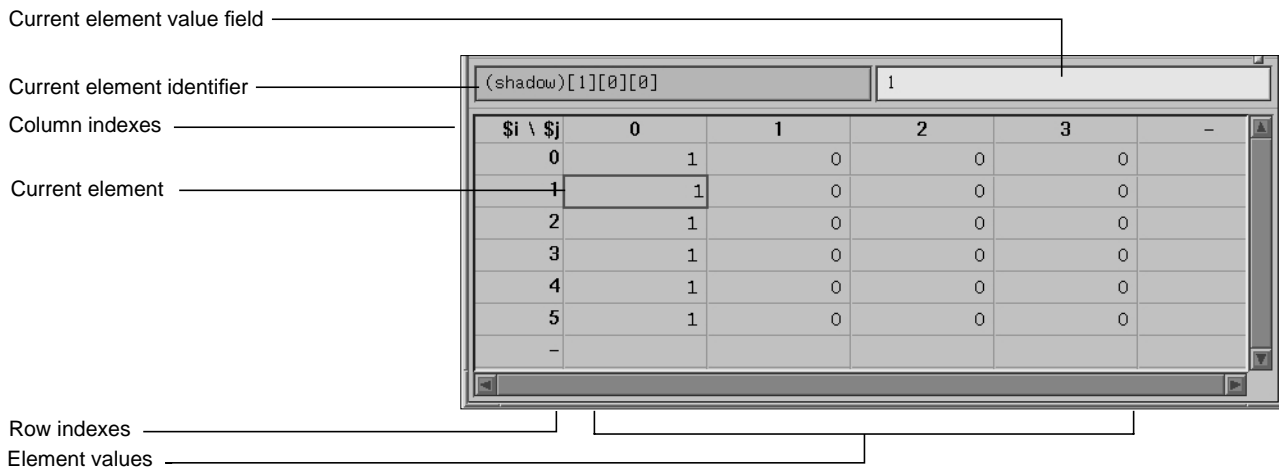


Figure A-87 Array Browser Spreadsheet Area

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.

Spreadsheet Menu

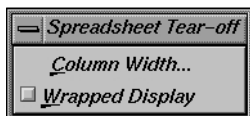


Figure A-88 Spreadsheet Menu

The Spreadsheet menu (see Figure A-88) 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 A-89 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).

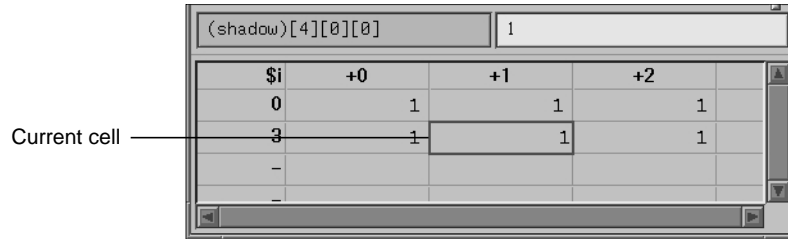


Figure A-89 Example of Wrapped Array

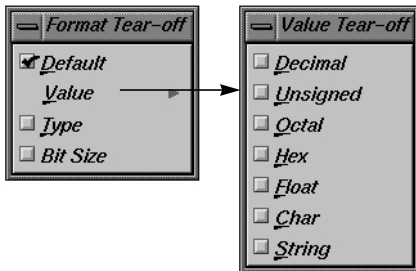


Figure A-90 Format Menu With Value Submenu

Format Menu

The Format menu (see Figure A-90) displays a separate menu that you lets you display the elements in the default format, as formatted values (decimal, unsigned, octal, and so forth) or as their data types.

The *graphical display area* presents the array data in a 3-D graph in one of the following forms:

- surface (polyhedron)
- bar chart
- points
- multiple lines (array vectors)



Figure A-91 Render Menu

Render Menu

You select the graphical display mode through the Render menu. The Render menu (see Figure A-91) has the following items:

- “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 Array Browser with no graphical display, in effect turning off graphical display mode.



Figure A-92 Color Menu

Color Menu

The Color menu (see Figure A-92) provides these 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 A-93 appear in the window.

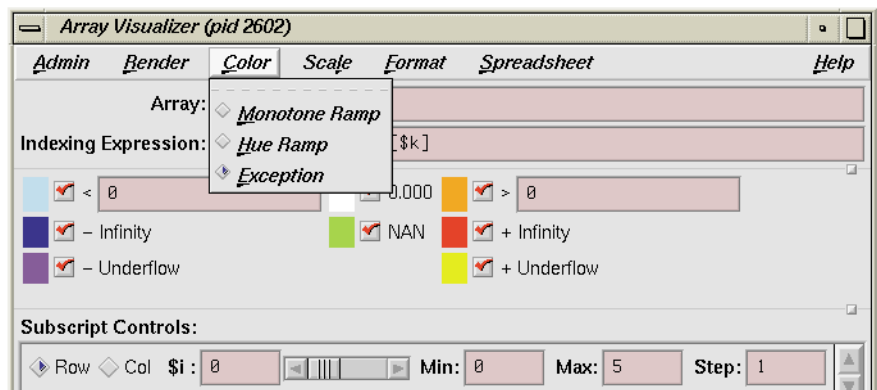


Figure A-93 Color Exception Portion of Array Browser Window

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.

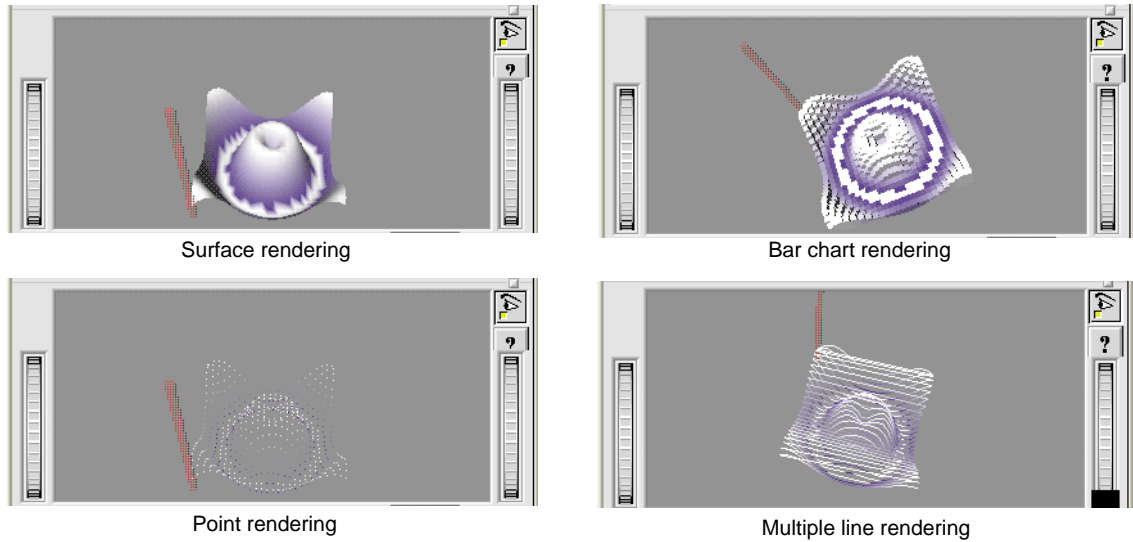


Figure A-94 Array Browser Graphic Modes

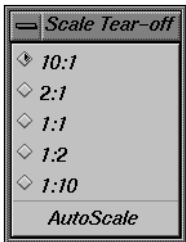


Figure A-95 Scale Menu

Scale Menu

The Scale menu (see Figure A-95) 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.

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 A-96, 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 are:

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 button 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).

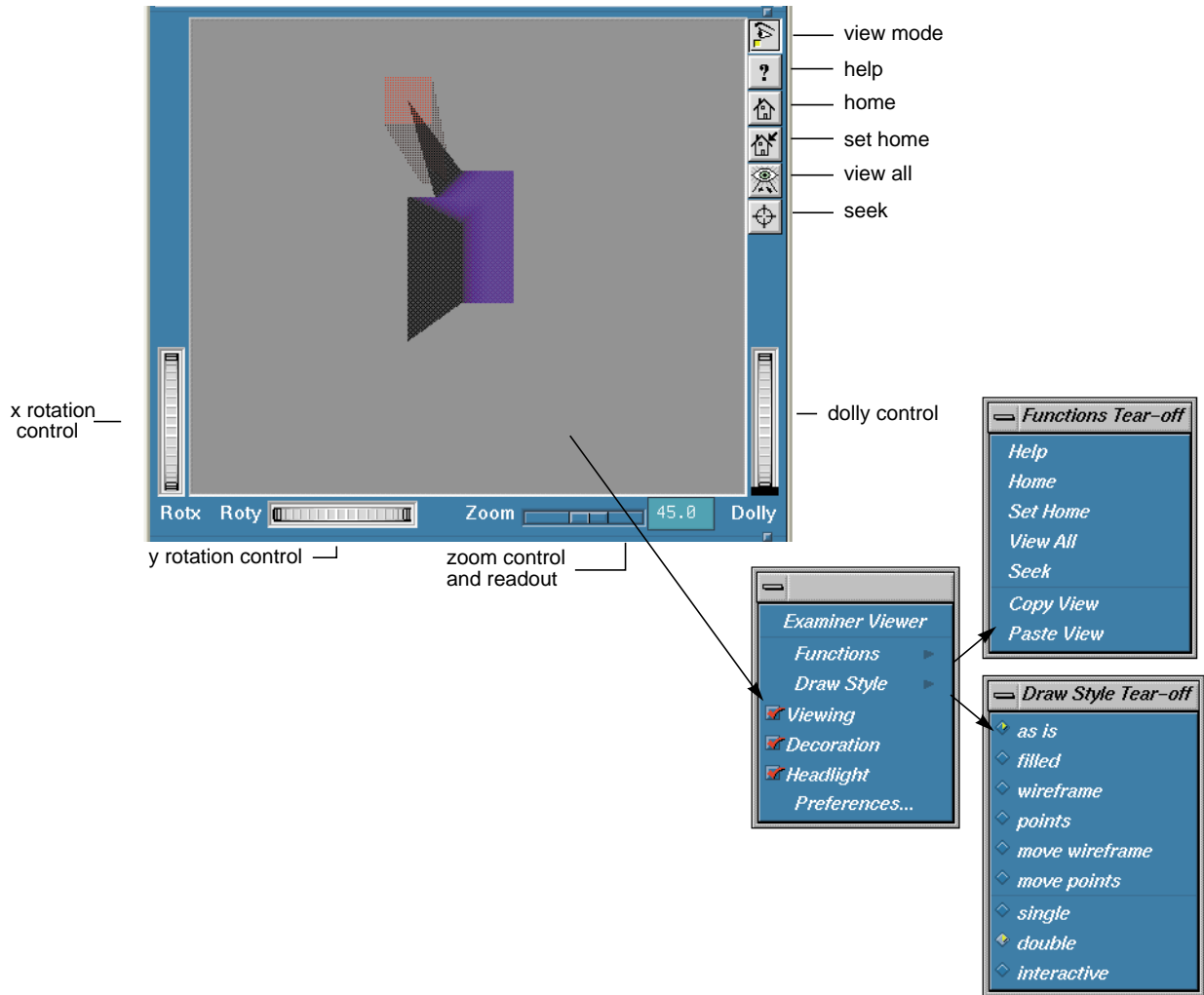


Figure A-96 Examiner Viewer With Controls and Menus

- help* Runs a special help system containing Inventor Viewer information.
- home* Repositions the graph in its original viewing position.

<i>set home</i>	Changes the home (original viewing) position for subsequent use of the <i>home</i> button.
<i>view all</i>	Repositions the display area so that the entire graph is visible.
<i>seek</i>	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.

These four controls let you move the graphic display:

<i>x rotation thumbwheel</i>	Rotates the graph around its x-axis.
<i>y rotation thumbwheel</i>	Rotates the graph around its y-axis.
<i>zoom slider and readout</i>	Changes the size of the graph by scaling it.
<i>dolly thumbwheel</i>	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.

Examiner Viewer Menu

You access the Examiner Viewer menu (see Figure A-96) by holding down the right mouse button in the graphical display area. The Examiner Viewer menu provides these selections:

“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 selections “as is,” “filled,” “wireframe,” and “points” control how the graph is displayed when it is static. These override the Render menu selections. The

selections “move wireframe” and “move points” 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 dialog box to display.

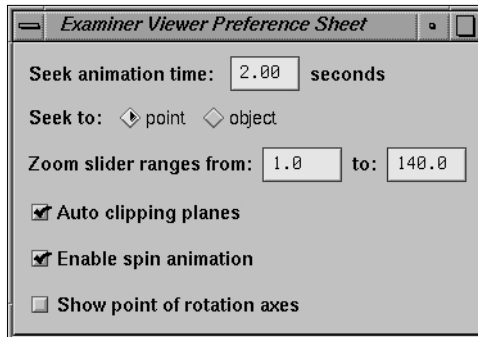


Figure A-97 Examiner Viewer Preferences Dialog Box

“Seek animation time”

Lets you specify the time it takes for the graph to be repositioned after you change the seek point. See “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.

Call Stack View

The Call Stack View (Figure A-98) displays the functions in the call stack (referred to as frames) when the process has stopped.

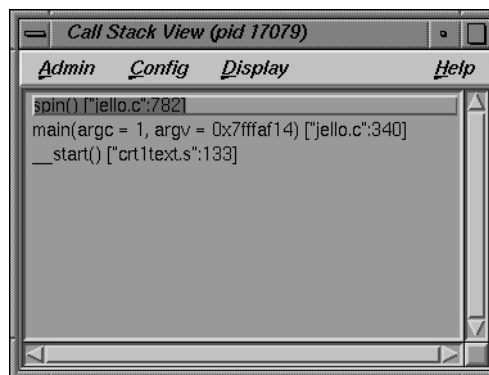


Figure A-98 Call Stack View

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.

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 “Admin Menu” on page 167. The Help menu is the same as that described in “Help Menu” on page 159. The other menus are described in the following sections.

Config Menu

The Config Menu (Figure A-99) contains the following items:

“Preferences...”

launches the preference dialog (Figure A-99), which allows you the option of setting the maximum depth of the Call Stack View.

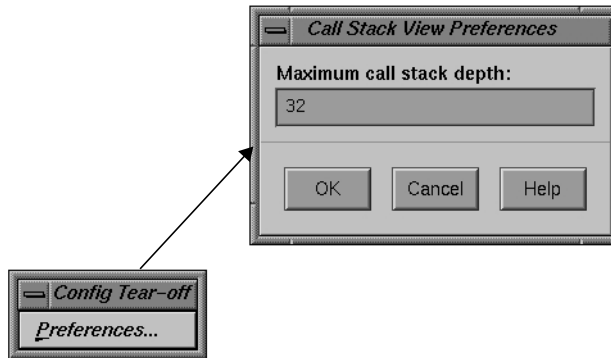


Figure A-99 Call Stack View Config Menu

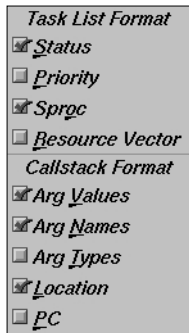


Figure A-100 Call Stack View Display Menu

Display Menu

The Display Menu (Figure A-100) contains the following toggles:

- “Arg Values” Allows you to set the argument values in the Call Stack View.
- “Arg Names” Allows you to set the argument names in the Call Stack View.
- “Arg Types” Allows you to set the argument types in the Call Stack View.
- “Location” Allows you to set the function location in the Call Stack View.
- “PC” Allows you to set the PC in the Call Stack View.

Expression View

The Expression View window is shown in Figure A-101. The expression view displays a collection of expressions that are evaluated each time the process stops or the context changes.

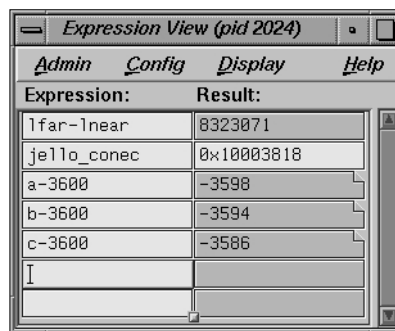


Figure A-101 Expression View

In addition to the items on the menu bar, the Expression View has two popup menus: the Language menu and the Format menu. The menu bar items and the two popup menus are discussed in the following sections. (The Admin menu is the same as that described in “Admin Menu” on page 167. The Help menu is the same as that described in “Help Menu” on page 159. The other menus are described in the following sections.)

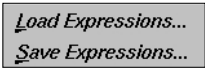


Figure A-102 Expression View
Config Menu

Config Menu

The Config Menu (see Figure A-102) contains the following items:

“Load Expressions...”

Launches the File menu (Figure A-49), allowing you to choose a source file from which to load your expressions.

“Save Expressions...”

Launches the File menu (Figure A-49), allowing you to choose a file to which you can save your expressions.



Figure A-103 Expression View
Display Menu

Display Menu

The Display Menu (see Figure A-103) contains the following items:

“Clear All” Clears all fields in the view.

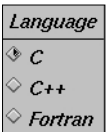


Figure A-104 Expression View
Language Popup

Language Popup

The Language popup (see Figure A-104) contains three radio buttons, allowing you to select one of three languages for evaluation: C, C++ or Fortran. The Language popup is invoked by holding down the right mouse button while the cursor is in the Expression column.

Format Popup

The Format popup (see Figure A-105) The Format menu is displayed by holding down the right mouse button in the Result column.

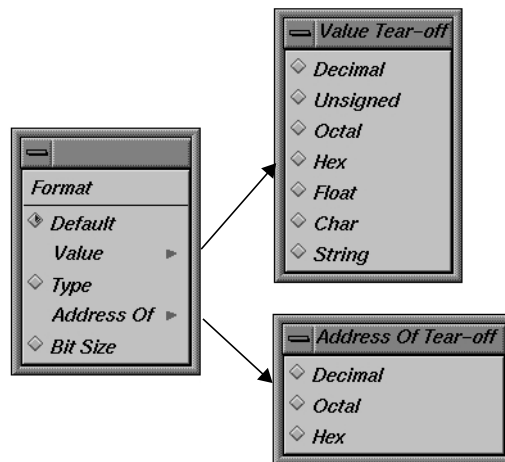


Figure A-105 Expression View Format Popup with Submenus

The Format popup contains the following items:

- “Default” A radio button that sets the format to the default values.
- “Value” Launches a submenu of radio buttons from which you can select a value of type Decimal, Unsigned, Octal, Hex, Float, Char, or String.
- “Type” Launches a submenu of radio buttons from which you can select a type of Decimal, Octal, or Hex.
- “Bit size” A radio button that sets the format to the bit size.

File Browser

The 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.

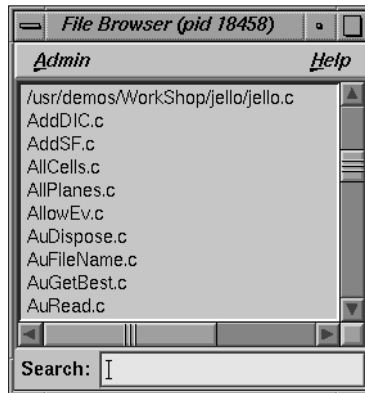


Figure A-106 File Browser

The File Browser contains an Admin menu (described in “Admin Menu” on page 167) and a Help menu (described in “Help Menu” on page 159). In addition, the browser has a Search field.

Structure Browser

The Structure Browser 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 A-107 with the Config, Display, Node, and Format menus displayed. (For a tutorial example of the Structure Browser, see “Examining Data” on page 34.)

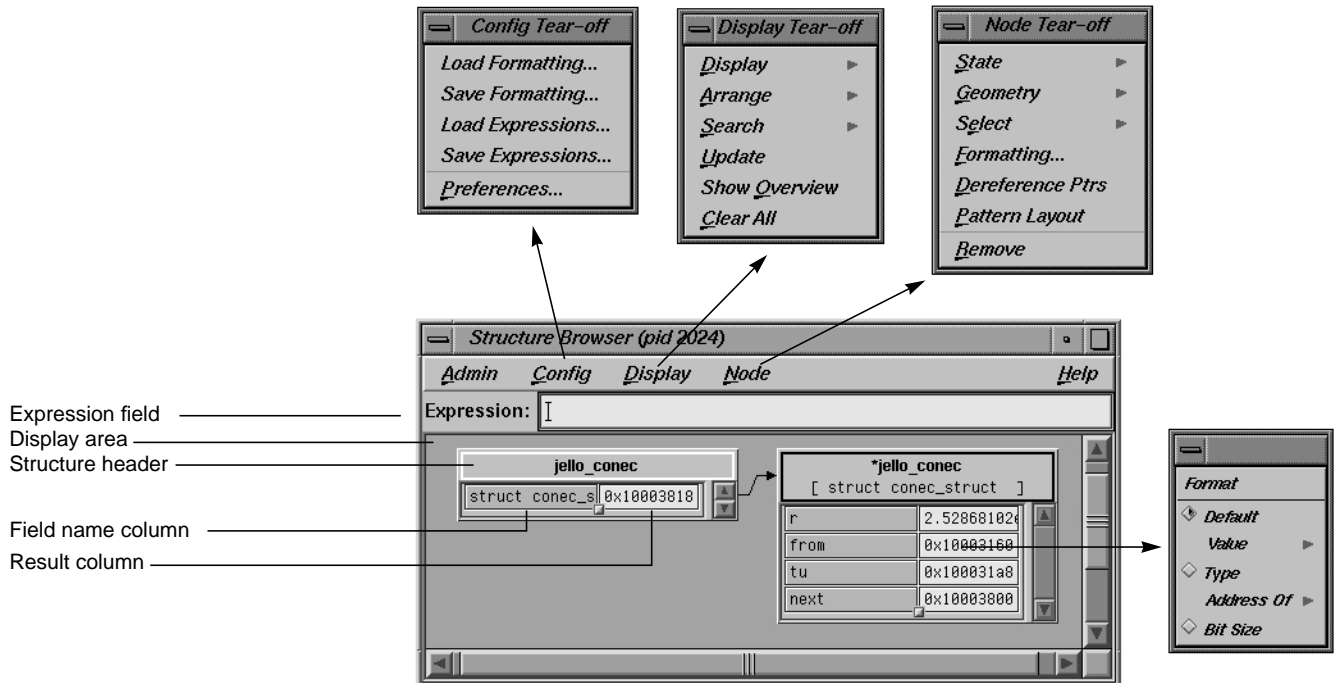


Figure A-107 Structure Browser With the Config, Display, Node, and Format Menus

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.

In addition to the Admin menu, Structure Browser provides four menus that are used to change the way the data displays. The menus are:

- | | |
|---------|---|
| Config | 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 popup menu that is accessed by holding down the right mouse button while the cursor is over the result column.

Using the Overview Window to Navigate

WorkShop provides the Overview window (from the “Show Overview” selection in the Display menu) as another way to navigate around the display area (see Figure A-108).

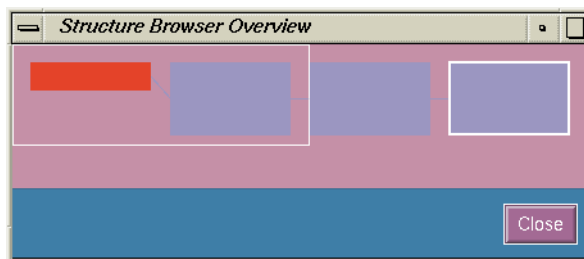


Figure A-108 Structure Browser Overview Window

The Overview 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 the Overview window repositions the currently visible area.

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 using the *Expression* field.

The *Expression* field is also used to enter strings used in searches.

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, used to change the display of a specific result (see Figure A-109). You can set a default format or request that results be displayed by value, type, address, or size in bits.

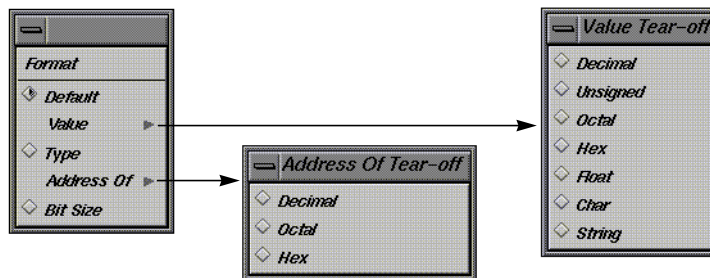


Figure A-109 Structure Browser Format Menu

Holding down the right button in the header of a Structure Browser object brings up the Node popup 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; 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 dereferences it, 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.

Structure Browser Display Menu

The Display menu controls the way structures appear in the display area of Structure Browser. The menu is shown in Figure A-110.

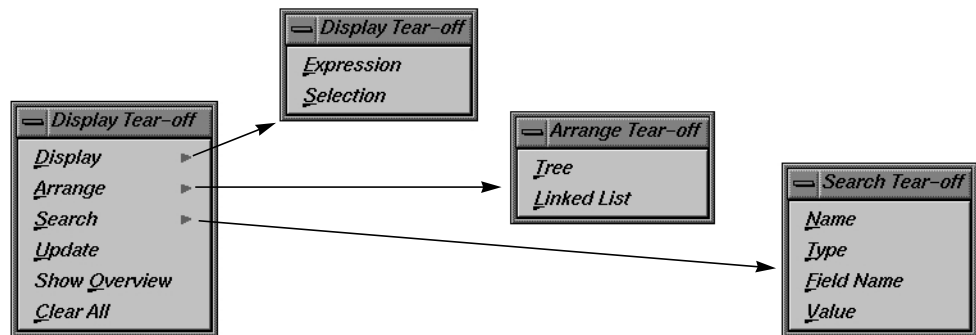


Figure A-110 Structure Browser Display Menu

The Display menu provides the following selections:

“Display”

has 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. There are two options (see Figure A-111):

- “Tree” arranges them 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 them into a linked list formation, that is, horizontally.

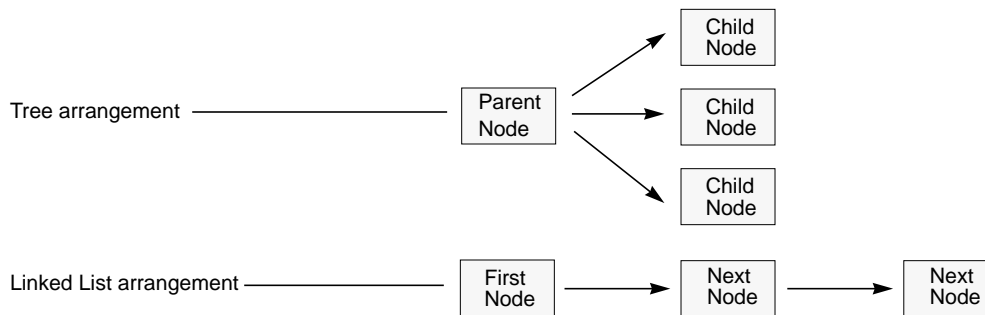


Figure A-111 Tree and Linked List Arrangements of Structures

“Search”

Lets you select structures containing the string specified in the *Expression* field. There are 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 Overview window.

“Clear All”

Clears all structures from the display area.

Node Menu

The Node menu is shown in Figure A-112.

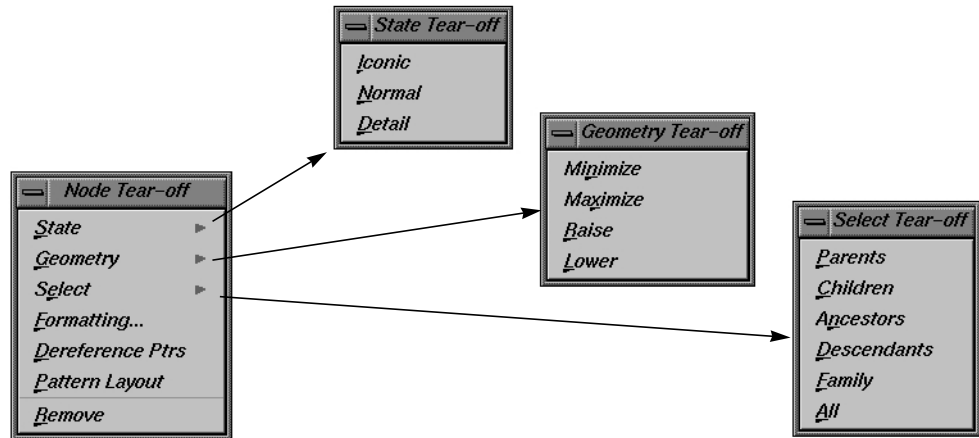


Figure A-112 Structure Browser Node Menu

The Node menu consists of the following entries and applies to the 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.
 - “Descendants” 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. See “Formatting Fields.”
- “Dereference Ptrs” Dereferences any pointers in selected objects.
- “Pattern Layout” Uses the spatial relationship between selected structures connected by pointers to position all similarly related structures in the same way.
- “Remove” Removes the selected object from the display.

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 Type Formatting dialog box. To display the

Structure Browser Preferences dialog box, select “Preferences...” from the Config menu (see Figure A-113).

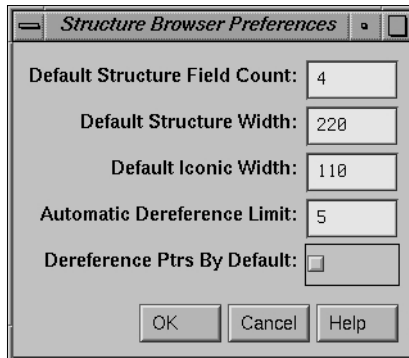


Figure A-113 Structure Browser Preferences Dialog Box

The Structure Browser Preferences dialog box 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

Sets a limit on the number of structures that are automatically dereferenced.

Dereference Ptrs By Default

Lets you toggle automatic dereferencing on and off.

To bring up the Type Formatting dialog box, select the set of structures under consideration and select “Node Formatting” from the Node menu (see Figure A-114).

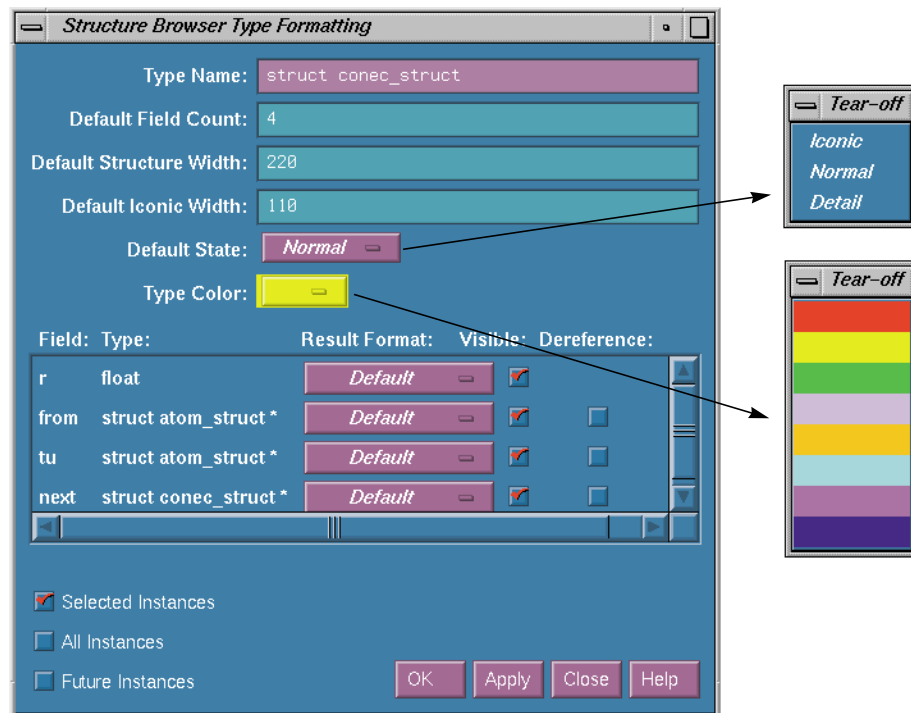


Figure A-114 Type Formatting Dialog Box

The Type Formatting dialog box has the following fields:

Type Name Displays the current data type.

Default Field Count

The number of fields to be displayed initially for objects of that type.

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.

Default State

Brings up a popup 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:

- 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

Variable Browser

The Variable Browser 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 the values, 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 A-115. (For a tutorial example of the Variable Browser see “Examining Data” on page 34.)

Typically, you inspect variable values

- at a stop trap
- 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.

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.

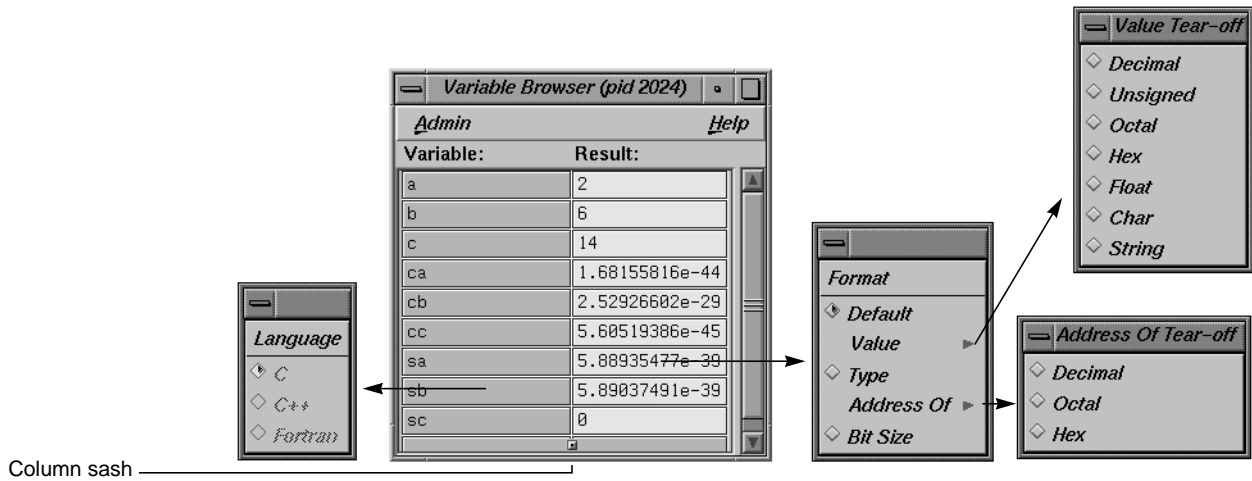


Figure A-115 Variable Browser With Language and Format Menus

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 A-115). For example, you may wish to adjust for short variable names and long result values.

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 A-116.

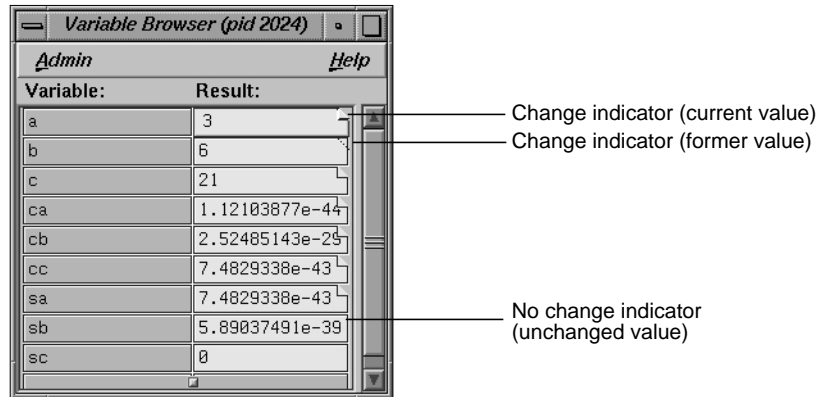


Figure A-116 Typical Variable Change Indicators

Machine-level Debugging Windows

The Debugger offers three views useful in debugging at the machine level; the Disassembly View, Register View, and Memory View.

Disassembly View

The Disassembly View of the Debugger lets you look at machine-level code rather than source-level code. A typical Disassembly View window appears in Figure A-117, with the Disassemble menu displayed.

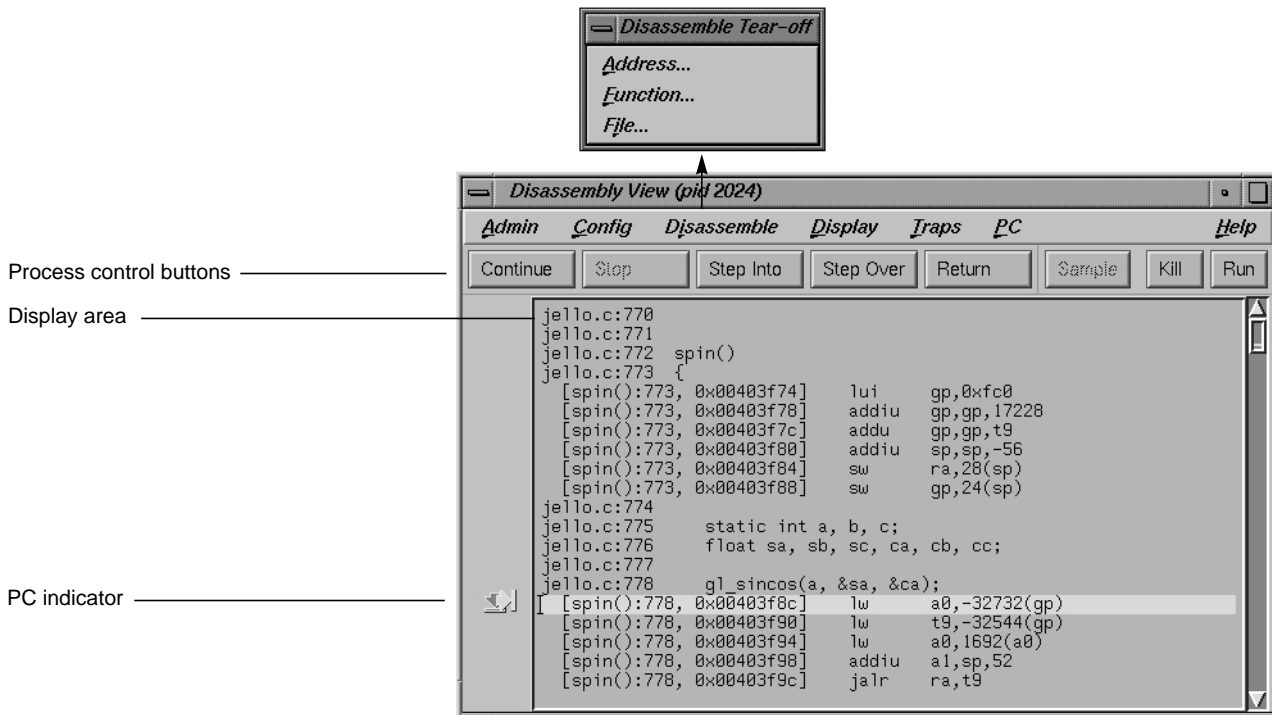


Figure A-117 Disassembly View With Disassemble Menu Displayed

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.

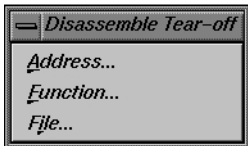


Figure A-118 Disassemble Menu

Disassemble Menu

The Disassemble menu (see Figure A-118) 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 A-119).

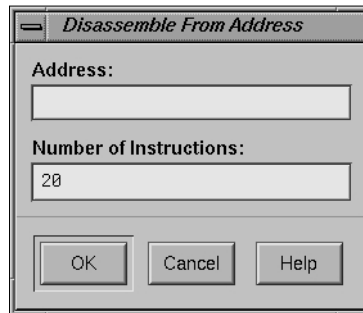


Figure A-119 Disassemble From Address Dialog Box

“Function...” allows you to disassemble a specified number of lines, starting from the beginning address of a specified function name (see Figure A-120).

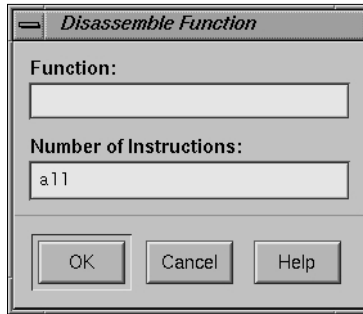


Figure A-120 Disassemble Function Dialog Box

“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 A-121). 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.



Figure A-121 Disassemble File Dialog Box

Disassembly View Preferences

Selecting “Preferences...” from the Config menu brings up the Disassembly View Preferences dialog box (shown in Figure A-122) so that you can change the global preferences.

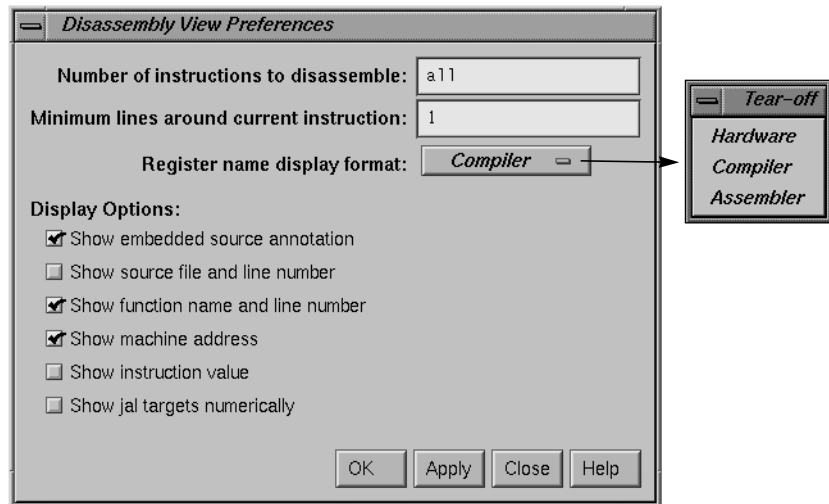


Figure A-122 Disassembly View Preferences Dialog Box with Format Popup Menu

The dialog box provides you with these 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 A-119, Figure A-120, and Figure A-121). 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.”

The *Display Options* selections control what information is shown in each disassembled line.

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.

Register View

Register View lets you examine and modify register values. You bring it up by selecting “Register View” from the Views menu in Main View. Figure A-123 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.

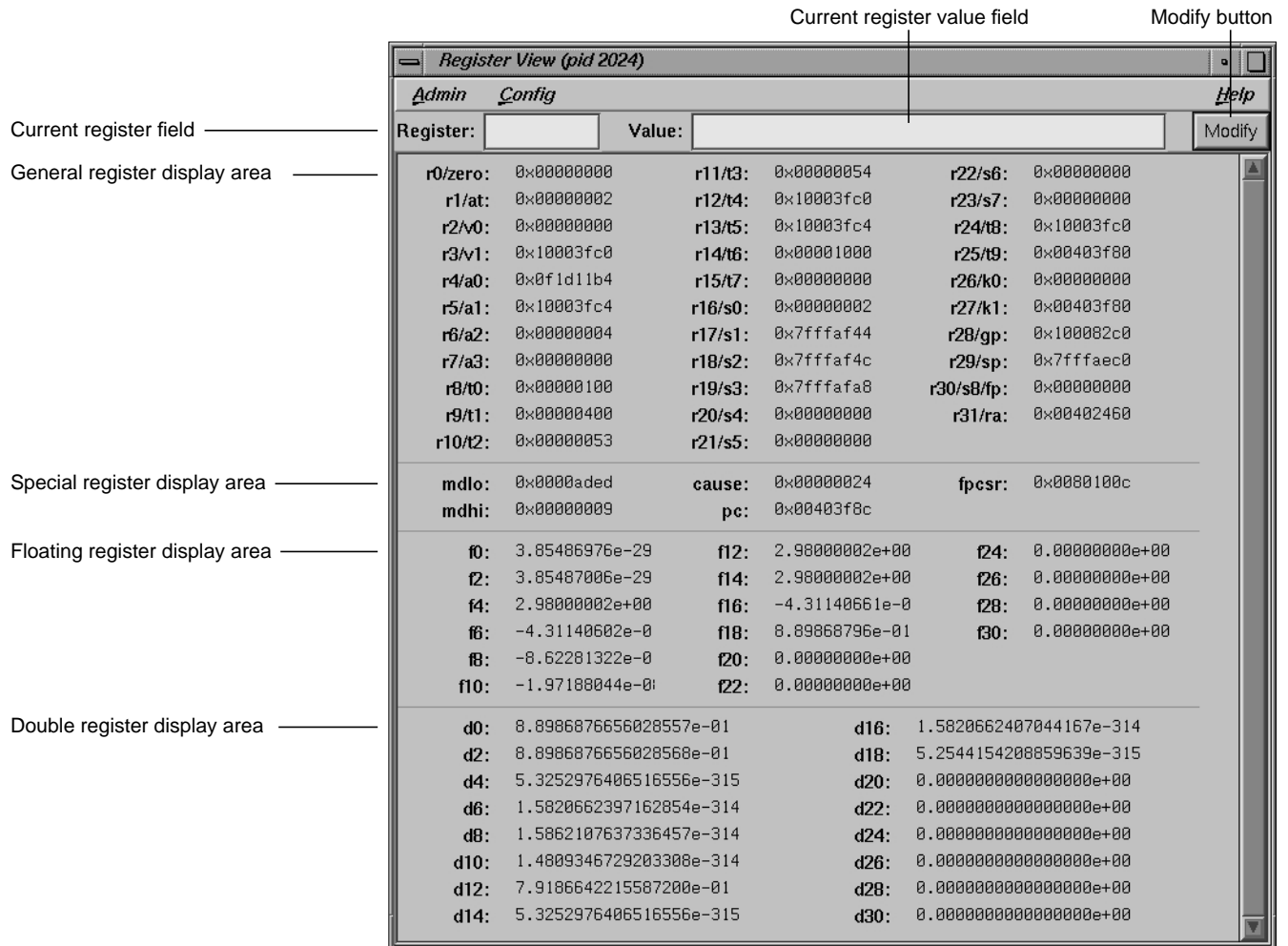


Figure A-123 Register View

Register View Window

The major features of the Register View window are:

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.

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 A-124).

The *Register Display* toggle buttons let you specify which types of registers are to be displayed by default.

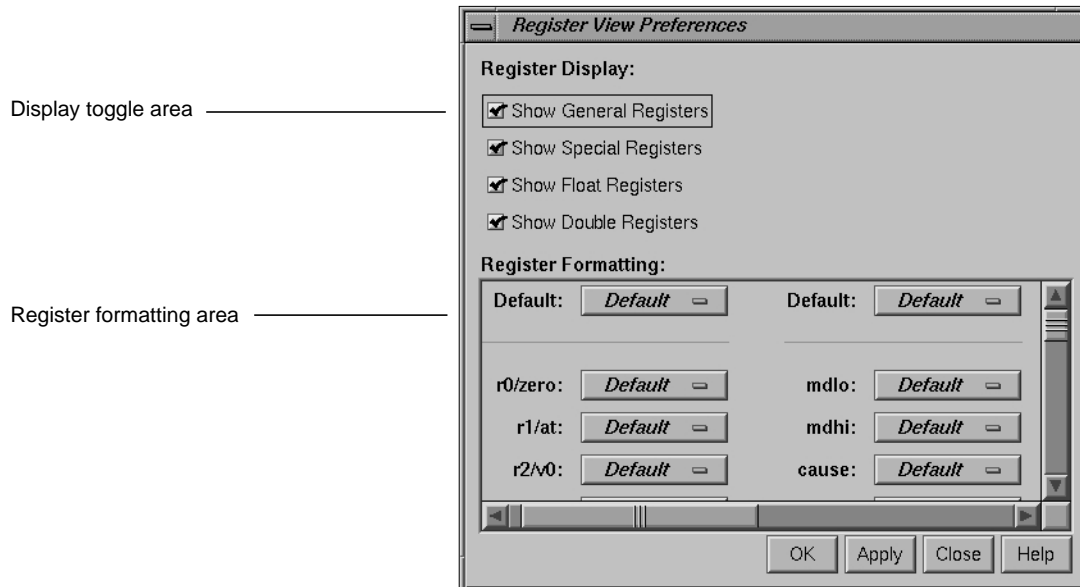


Figure A-124 Register View Preferences Dialog Box

The *Register Formatting* area lets you select formats for any of the registers. You have a choice of “default,” “decimal,” “octal,” or “hex” format.

The default fields in the top row let you change the 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.

Memory View

Memory View lets you examine and modify memory. A typical Memory View window appears in Figure A-125.

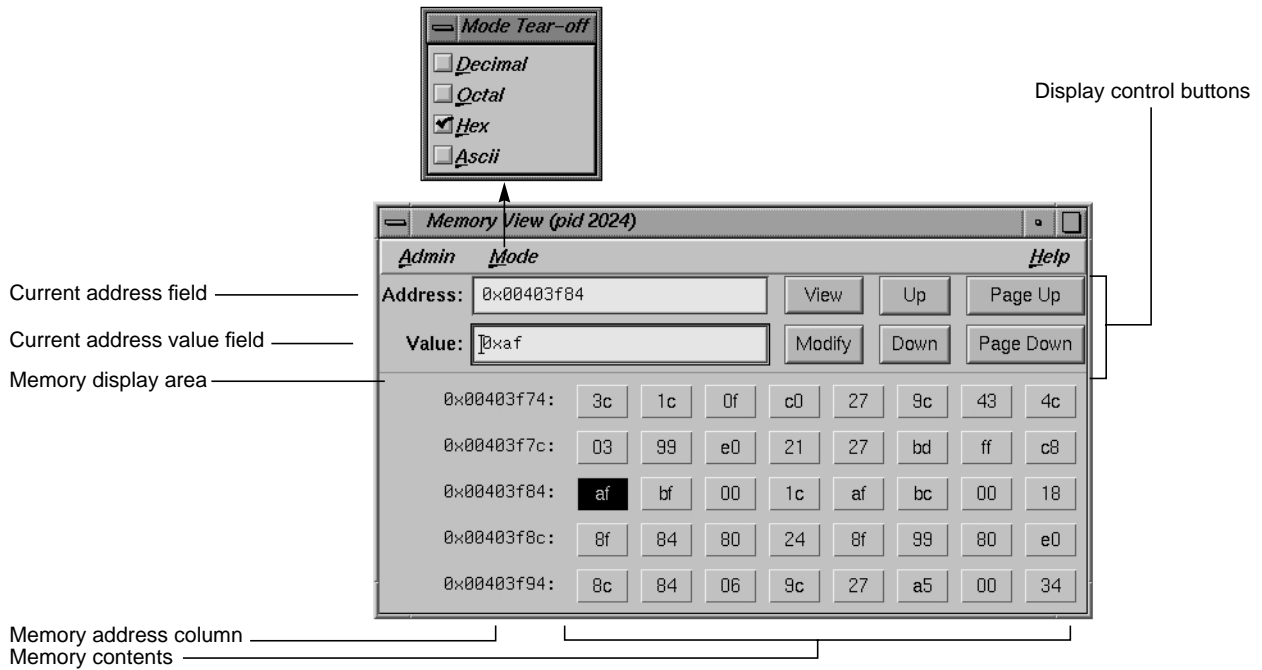


Figure A-125 Memory View With Mode Menu Displayed

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.

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.

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.

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:

- | | |
|------------------|---|
| <i>Up</i> | for moving the displayed bytes up a single row. |
| <i>Down</i> | for moving the displayed bytes down a single row. |
| <i>Page Up</i> | for moving the displayed bytes upward by as many rows as are currently displayed. |
| <i>Page Down</i> | for moving the displayed bytes downward by as many rows as are currently displayed. |

Multiple Process Debugging Windows

WorkShop supports performance analysis and debugging of multiprocess applications, including processes spawned either with *fork* or *sproc*. Multiprocess debugging is supported primarily through the Multiprocess View.

Multiprocess View

Select “Multiprocess View” from the Admin menu to bring up Multiprocess View. Main View is attached to the parent process. Figure A-126 shows a typical Multiprocess View with Config and Process menus displayed.

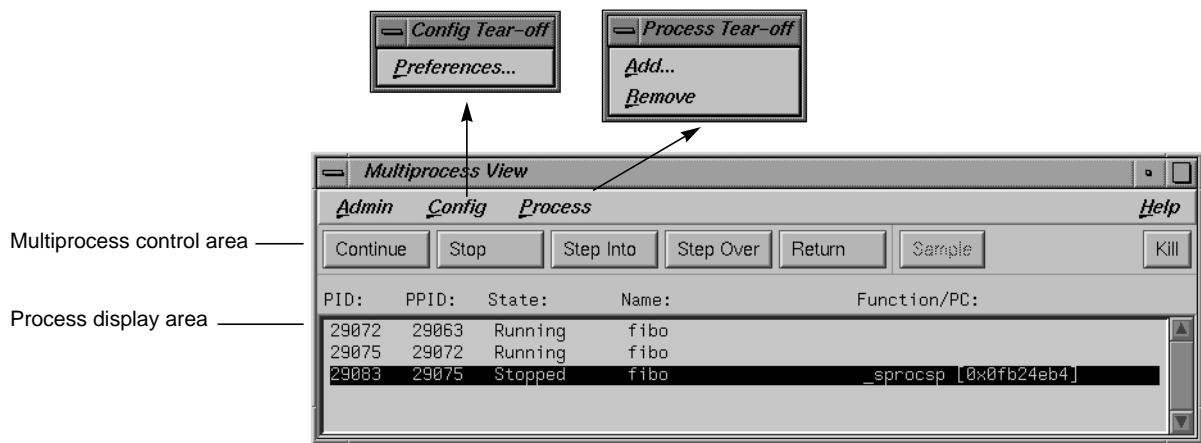


Figure A-126 Multiprocess View with Config and Process Menu Displayed

To open a Main View (or other debugging views) for another process, double-click the desired process in Multiprocess View. A separate Main View window displays the selected process, and you can select any debugging views desired. If a set of views exists for that process, the views are raised to the foreground. To reuse views already displayed, select “Switch Process...” from the Admin menu in Main View. (If a process is currently highlighted in Multiprocess View, its id is entered automatically in the *Process id:* field in the Switch Process dialog box.)

Viewing Process Status

When Multiprocess View comes up, it lists the status of all processes in the process group. This information includes:

<i>PID:</i>	shows the process identifier (id).
<i>PPID:</i>	lists the parent process ids. Notice in Figure A-126 that the first process PID#7748 is the parent process of the second.
<i>State:</i>	represents the state of the process: stopped, running, or created, which appears just prior to running. Terminated processes are not displayed.
<i>Name:</i>	identifies the process by filename.
<i>Function/PC:</i>	indicates the current function and program counter (PC) for any stopped processes.

Multiprocess Control Buttons

Multiprocess View uses the same control buttons as MainView 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 buttons are:

<i>Continue</i>	resumes program execution after a halt and continues until a stop trap or other event stops execution.
<i>Stop</i>	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.
<i>Step Into</i>	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 <i>Step Into</i> button. A popup menu displays that lets you select one of the fixed values or a specified number of steps.
<i>Step Over</i>	steps to the next source line and over function calls. To step a specific number of lines, hold down the right button over the <i>Step Over</i> button. A popup menu displays that lets you select one of the fixed values or a specified number of steps.

<i>Return</i>	executes the remaining instructions in the current function. Program execution stops upon return from that procedure.
<i>Sample</i>	collects performance data for each process (if performance data collection is enabled).
<i>Kill</i>	terminates all processes in the group.

Multiprocess Traps

As discussed in Chapter 4, “Setting Traps,” 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 the “Group Trap Default” and “Stop All Default” selections in the Traps menu in Trap Manager.

Note that the *Sample* button always samples all processes.

Adding and Removing Processes

The Process menu lets you manually add or remove a process from the process group (see Figure A-127).

To remove a process, click the process and select “Remove” from the Process menu. Note that a process in a *sproc* share group cannot be removed from the process group.

To add a process, select “Add...” The dialog box shown in Figure A-128 displays. Enter the new process id and click OK.

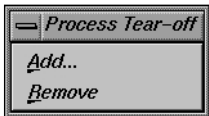


Figure A-127 Process Menu in Multiprocess View

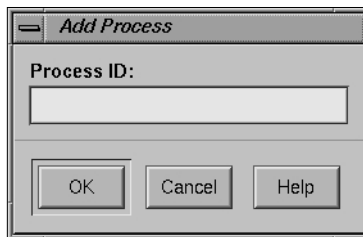


Figure A-128 Add Process Dialog Box

Multiprocess Preferences

The “Preferences...” option in the Config menu brings up the Preferences dialog box. It lets you control when processes are added to the group, and it specifies their behavior (see Figure A-129).

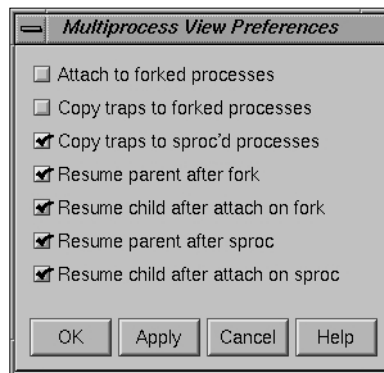


Figure A-129 Multiprocess View Preferences Dialog Box

The Multiprocess View preference options are:

Attach to forked processes

attaches new processes spawned by the *fork* command to the group automatically. (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.

Fix+Continue Windows

The Fix and Continue GUI affects several WorkShop windows and provides three more. The Debugger and Source View access the Fix and Continue utility from the menu bar. The results of running redefined code are displayed in the Debugger Execution View. Special line numbers (decimal notation) applied to redefined functions appear in several WorkShop views (refer to “Changes to Debugger Views” on page 264). Fix and Continue comes with three windows devoted entirely to Fix and Continue: Status, Message, and Build Environment. This section describes Fix and Continue menu selections and these windows.

The Fix and Continue menu is available from the Debugger Main View menu bar, as shown in Figure A-130. The menu selections operate on the selected function or on the file shown in the source view. The Fix and Continue menu is also available from Source View and from the Fix and Continue Status window.

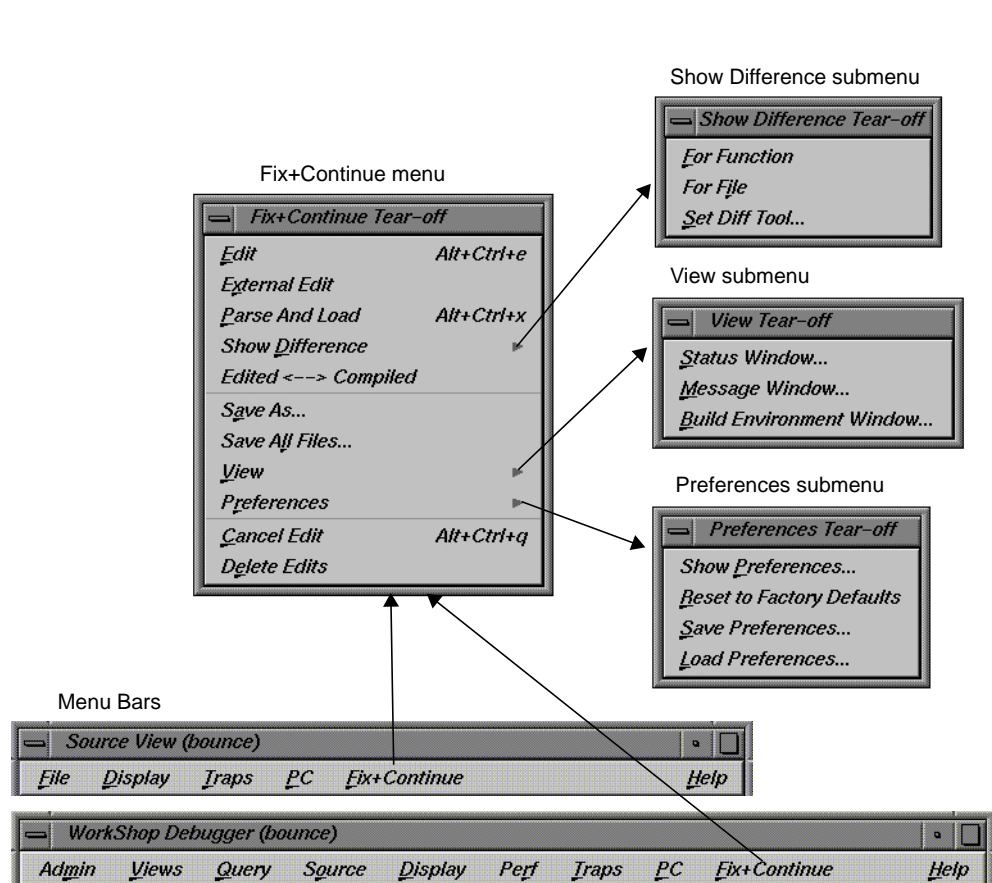


Figure A-130 Fix+Continue Menu Selections

Fix+Continue Status Window

This section describes the Fix+Continue Status window (see Figure A-131). The Status window provides you with a summary of the modifications that you have made during your session. It also allows you quick access to your modified functions, and a somewhat expanded Fix+Continue menu.

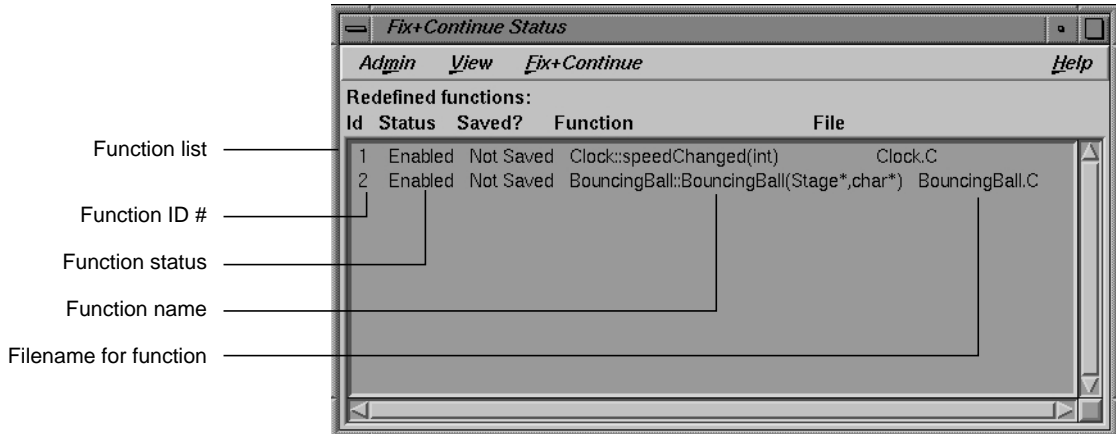


Figure A-131 Fix+Continue Status Window

The function ID number, status, name, and filename are displayed in the Status window. Double-clicking a line item in the status 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 A-132) are described below.

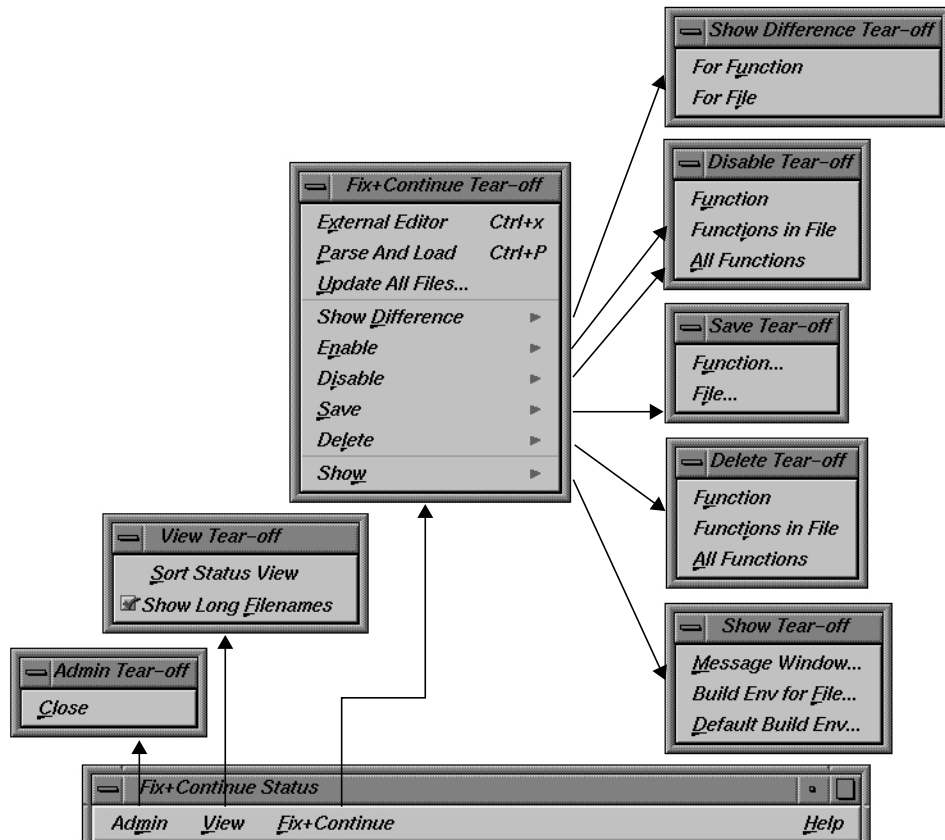


Figure A-132 Fix+Continue Status Window Menus



Figure A-133 Status Window Admin Menu

Admin Menu

The Admin menu (see Figure A-133) contains an option for closing the window.

“Close” Closes the Status window.



Figure A-134 Status Window View Menu

View Menu

The View menu (see Figure A-134) contains options for sorting the information in the window, and displaying filenames.

“Sort Status View”

Sorts the information in the status view according to the field currently selected.

“Show Long Filenames”

A toggle that allows you to show the absolute (long) pathnames, relative pathnames, or base names.

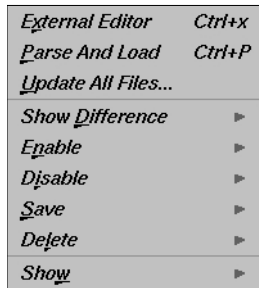


Figure A-135 Status Window Fix+Continue Menu

Fix+Continue Menu

The Fix+Continue menu (see Figure A-135) that is available from the Status view is somewhat different from that available through the Debugger main view. It contains a number of options and submenus, which are all 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.

“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 (see Figure A-25), which allows you to update the current session, saving all the modified functions to the appropriate files.

“Show Difference” submenu (see Figure A-136)

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.

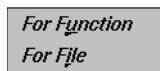


Figure A-136 Show Difference Submenu

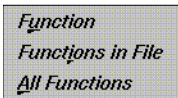


Figure A-137 Enable Submenu

- “For File” shows the differences for the entire file that contains the current function.

“Enable” submenu (see Figure A-137)

Allows you to enable the changes in your modified code in one of the three 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.

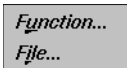


Figure A-138 Save Submenu

“Disable” submenu (see Figure A-137)

Has the same menu choices as the “Enable” submenu, but disables rather than enables.

“Save” submenu (see Figure A-138)

Allows you to save your code changes to a file. You can save the changes in one of the three following ways:

- “Function...” launches the File dialog (see Figure A-139), allowing you to save only the current function to a file.
- “File...” launches the “Save File+Fixes As...” popup window (see Figure A-25), allowing you to save the entire file that contains the current function.

“Delete” submenu (see Figure A-137)

has the same menu choices as the “Enable” submenu, but deletes rather than enables.

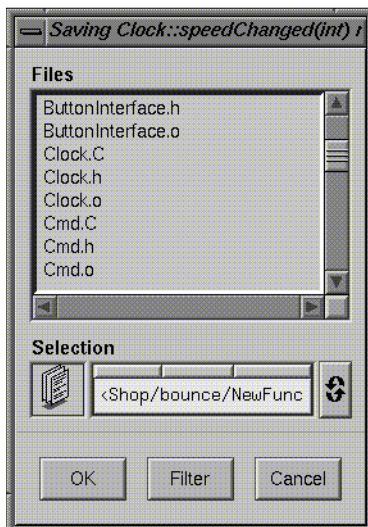


Figure A-139 File Dialog

“Show” submenu (see Figure A-140)

Allows you to launch any of the following three different Fix and Continue windows:

- “Message Window” launches a Message window for the selected function. See “Fix+Continue Message Window” on page 260 for more details.
- “Build Env for File” launches a Build Environment window for the file shown in the Source View. See “Fix+Continue Build Environment Window” on page 262 for more details on the Build Environment window.
- “Default Build Env” launches the Build Environment window to show the options that are to be used in cases where they could not be obtained from the target. See “Fix+Continue Build Environment Window” on page 262 for details on the Build Environment window.

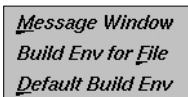


Figure A-140 Show Submenu

Fix+Continue Message Window

The Fix+Continue Message window (see Figure A-141) contains a list of any errors and other system messages that pertain to your source modifications, parses, and attempts to run your modified source.

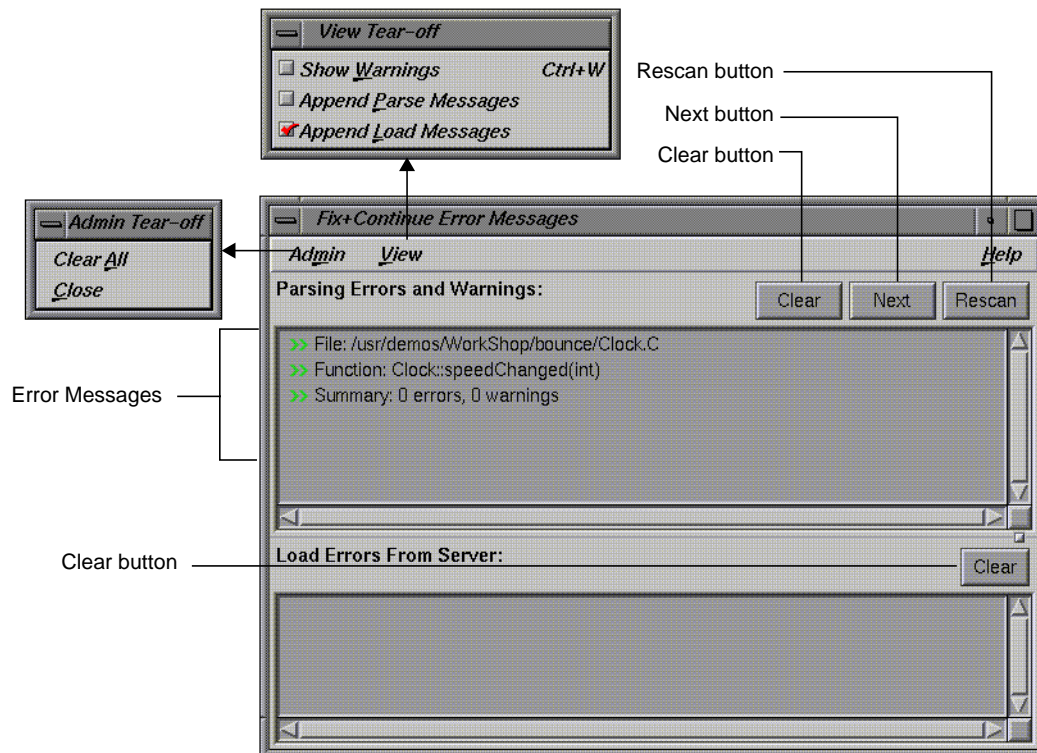


Figure A-141 Fix+Continue Message Window

You can highlight the source line where the error occurred by double-clicking the appropriate line in the Message window. The window contains the following buttons:

- | | |
|---------------|--|
| <i>Clear</i> | Clears all the parsing errors and warnings. |
| <i>Next</i> | 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. <i>Next</i> doesn't function after all the entries in the messages are ticked. |
| <i>Rescan</i> | Erases all the ticks, so that you can rescan all the error warnings from the beginning. |

The added functionality available through the Message window's Admin and View menus is described below.

Admin Menu

The Admin menu allows you to perform either of the following two operations:

- "Clear All" Clears all messages in the Message window.
- "Close" Closes the window.

View Menu

The View menu allows you to set any of the following three 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.

Fix+Continue Build Environment Window

This section describes the Fix+Continue Build Environment window (see Figure A-142). The 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.

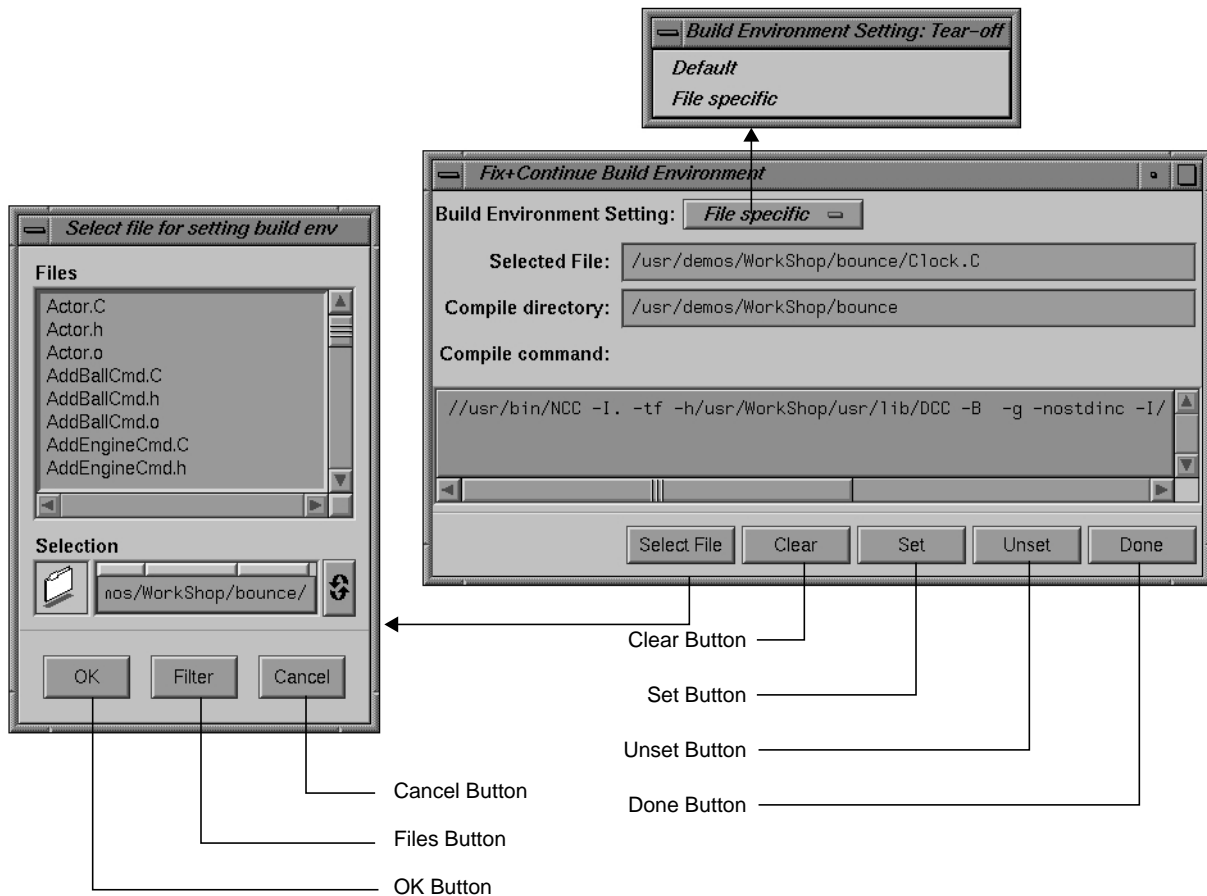


Figure A-142 Fix+Continue Build Environment Window

The compiler and associated flags that were used to compile the file are normally gathered from the target. You can use the Build Environment window to make any changes to these flags.

The Build Environment window allows you to select your build environment setting through the "Build Environment Setting" toggle, which contains the two options described below:

"Default" Sets the build environment to default that is displayed in the Build Environment 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 (see Figure A-139).

The Build Environment window also contains the following buttons:

- | | |
|--------------------|---|
| <i>Select File</i> | Launches the File dialog and allows you to select a file from which to set the build environment. |
| <i>Clear</i> | Clears the window. |
| <i>Set</i> | Sets the build environment to what is displayed in the window. |
| <i>Unset</i> | Unsets the build environment. |
| <i>Done</i> | Dismisses the window. |

Changes to Debugger Views

When you use Fix and Continue, Debugger views change to show redefined functions or stopped lines containing redefined functions.

Main View

When you open the Debugger after installing Fix and Continue, you’ll notice several changes to the environment. All Fix and Continue functions are available through the Fix+Continue menu. See Figure A-143 for details.

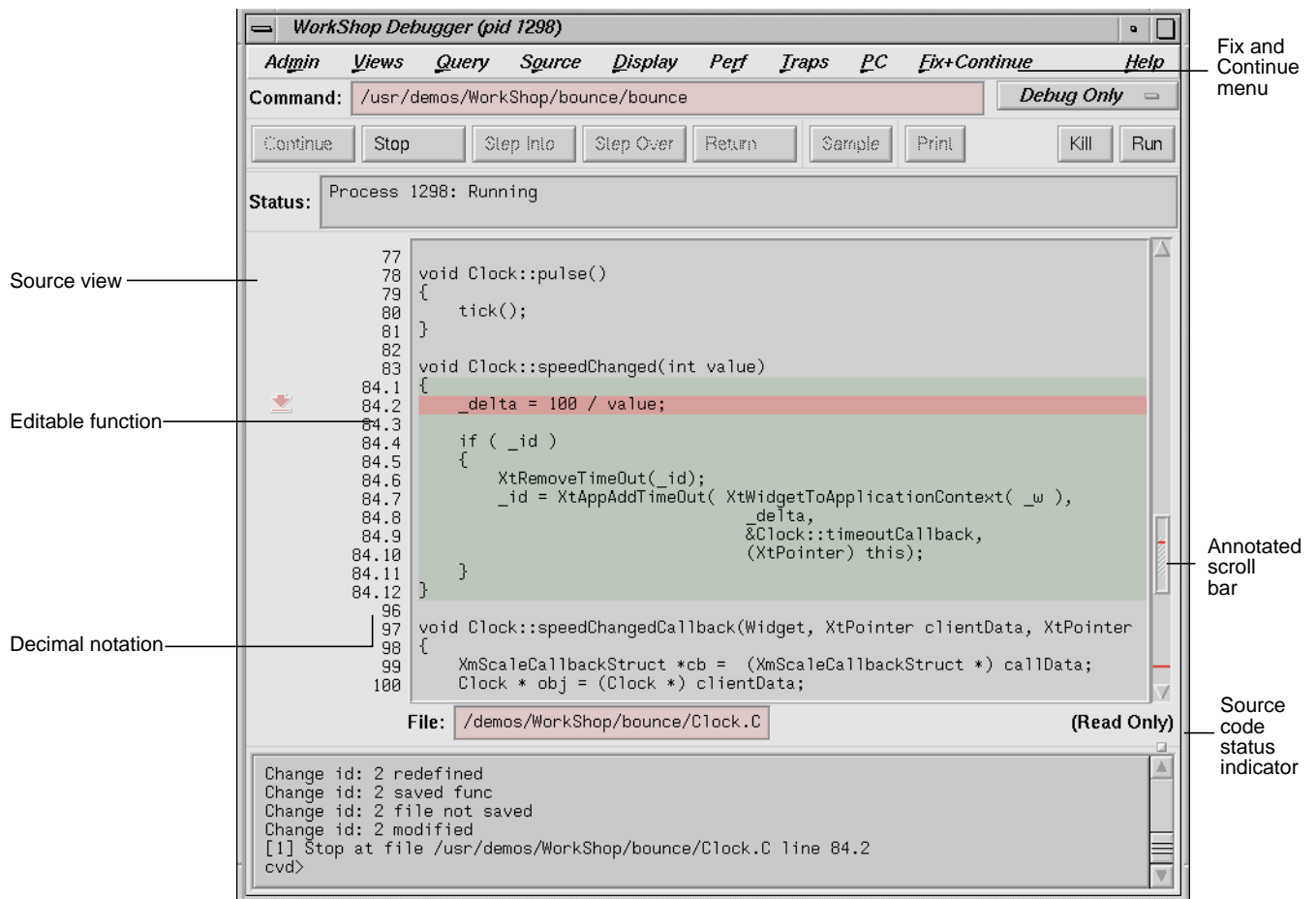


Figure A-143 Debugger Main View

You select Fix and Continue 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.

Command-Line Interface

The Debugger command-line interface accepts Fix and Continue commands and reports status involving redefined functions or files. Figure A-144 shows a function successfully redefined using the command line. Change id 1 was previously redefined and assigned the number 1.

Specify function with Change id 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 A-144 Command-Line Interface With Redefined Function

Call Stack

The Call Stack View recognizes redefined functions. It uses the decimal notation for line numbers, as shown in Figure A-145.

Decimal notation for line number

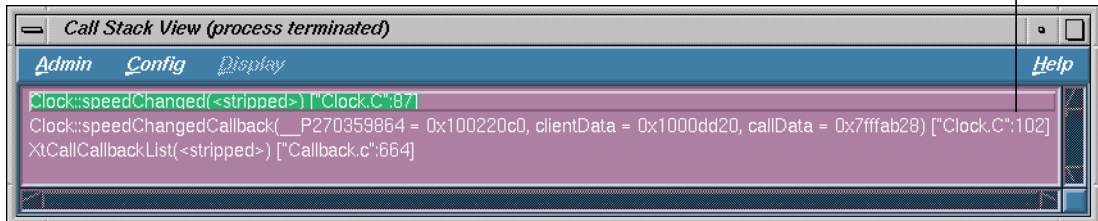


Figure A-145 Call Stack

Trap Manager

The Trap Manager recognizes redefined functions. It uses the decimal notation for line numbers, as shown in Figure A-146.

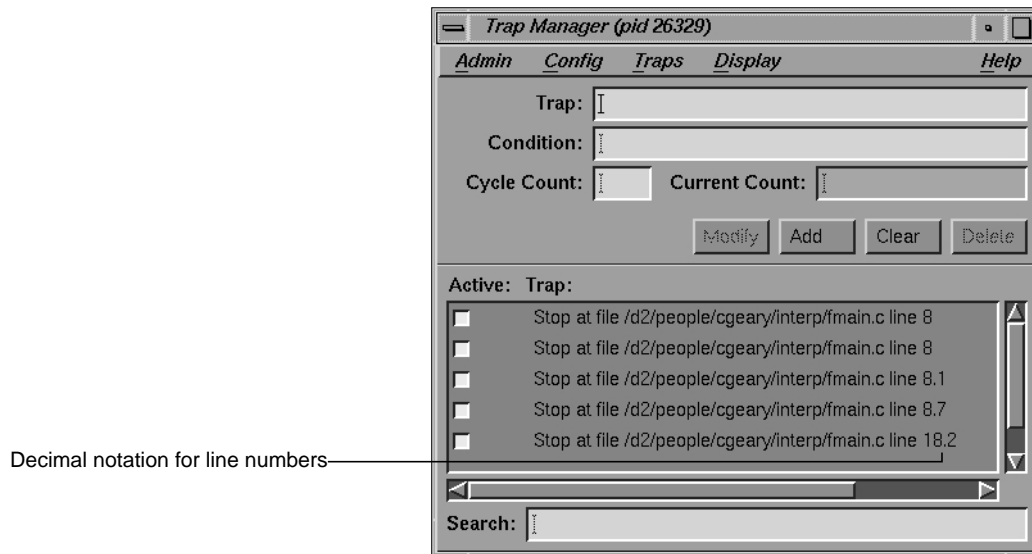


Figure A-146 Trap Manager With Redefined Function

Debugger Command Line

To use the Debugger commands, which are entered at the command line at the bottom of Main View (see Figure A-1), 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**, **replace_source**.

alias [*shortform command*]
Lists all aliases without arguments. With arguments, it assigns *command* to *shortform*.

assign *expression1=expression2*
Assigns *expression1* to *expression2*.

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 interactive 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 Continues executing a program after a breakpoint. Note that you can **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 *displaynumber* [,*displaynumber*, ...]
Deletes the specified expression from the display list.

delete all deletes all traps.

delete_changes {*func_spec* | **-all** | {-file *filename*}}
Undoes the changes corresponding to the selected functions (for example, **delete_changes** *getNumbers* **-file** *fmain.c*). Once deleted, you won't be able to use the IDs again, since the IDs associated with the selected functions are released. The default is **-all**. See also **save_changes**.

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.

delete trap_number [,*trap_number*, ...]
Deletes the specified breakpoint from the status list.

detach Detaches from the current process.

disable all Deactivates all inactive traps.

disable_changes {*func_spec* | **-all** | {-file *filename*}}
Undoes changes specified for the 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 on the display list.

down [*expression*]
Moves down the specified number of frames in the call stack. **down** moves in the direction of the called function.

dump
Prints local variable values.

enable all
Reactivates all inactive traps.

enable_changes {*func_spec* | **-all** | {-file *filename*}}
Redoes changes specified for the 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.

enable *trap_number* [,*trap_number*, ...]
Reactivates a disabled stop trap.

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 format options are:

- 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*
Goes to the specified line number.
- 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*]
Kills the specified process currently controlled by the Debugger.
- list** [[*expression* [, *expression*]] | [*function_name*]]
Lists the specified number (*expression*) of lines. The default is 10 lines. You can optionally specify a function where the list is to take place.
- 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. Figure A-147 shows the *vi* editor.

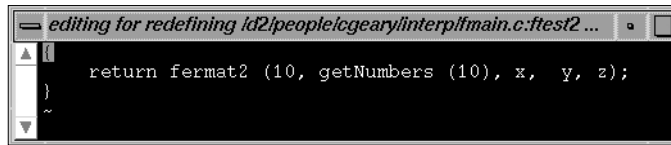


Figure A-147 Editing a Function in the vi Editor

- **-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.

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**.

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 Runs the program.

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 the checks selected will be printed when the specified function is entered next time. You can invoke this command when the process is stopped or on a running process where a break point is set at function entry.

key = [[+|-] runtime_check option *key*]

For example:

A unique identifier (*key*) is returned whenever you specify a function as an argument for `runtime_check`.

save_changes {*func_spec* | {-file *filename*}}

[-[w|a]] *filename_to_save*

Saves (enabled or disabled) function redefinitions or an entire file to a separate file (*filename_to_save*). An example of saving a function definition is the following:

```
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
```

-w replaces the *filename_to_save*. *-a* appends to the *file_to_save*. An example of adding a function to a file is the following:

```
save_changes getNumbers -a newFuncs
```

See also `delete_changes`.

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
```

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 so far.

showbuildenv "*filename*" lists any build environment specs set separately with **setbuildenv** "*filename*".

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**.

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).

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. 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** Performs a stack trace.

Using the Build Manager

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 CASEVision 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* 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).

Build View

You can access Build View from the WorkShop analysis tools, from a command line (by typing *cvmake*), or from 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, “Getting Started with the WorkShop Debugger”). 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 is shown in Figure B-1 with its Admin menu.

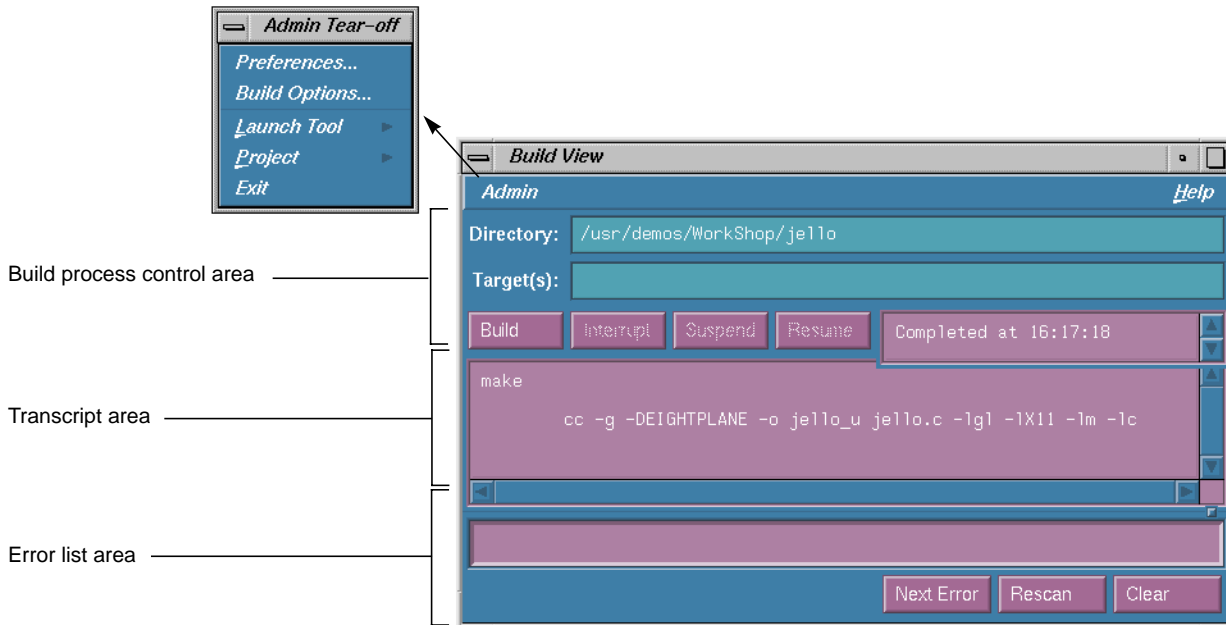


Figure B-1 Build View Window With Admin Menu Displayed

The Build View window has three major areas:

- “Build Process Control Area”
- “Transcript Area”
- “Error List Area”

Build Process Control Area

The build process control area lets you run or stop the build and view the status. See Figure B-2.

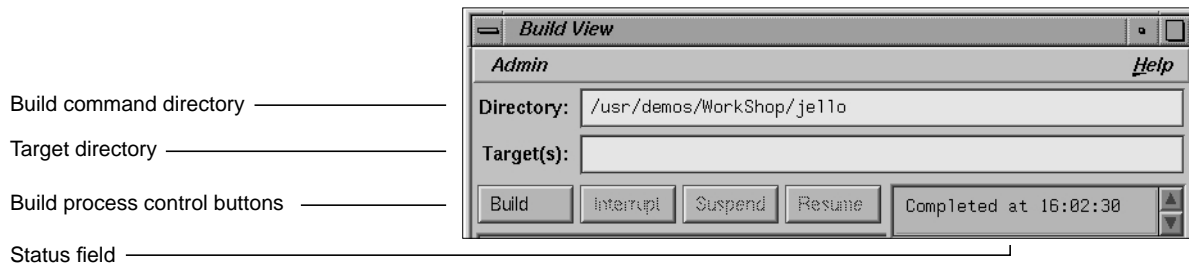


Figure B-2 Build Process Control Area in Build View Window

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 “Build View Preferences” and “Build Options”). The target to be built is specified in the Target(s) field.

The *build process control buttons* let you control the build process. The buttons are:

Build runs (or reruns) a build.

Note: If you have modified any files in Source View, you will be prompted to save the new version prior to the compile.

Interrupt terminates 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.

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 B-3.

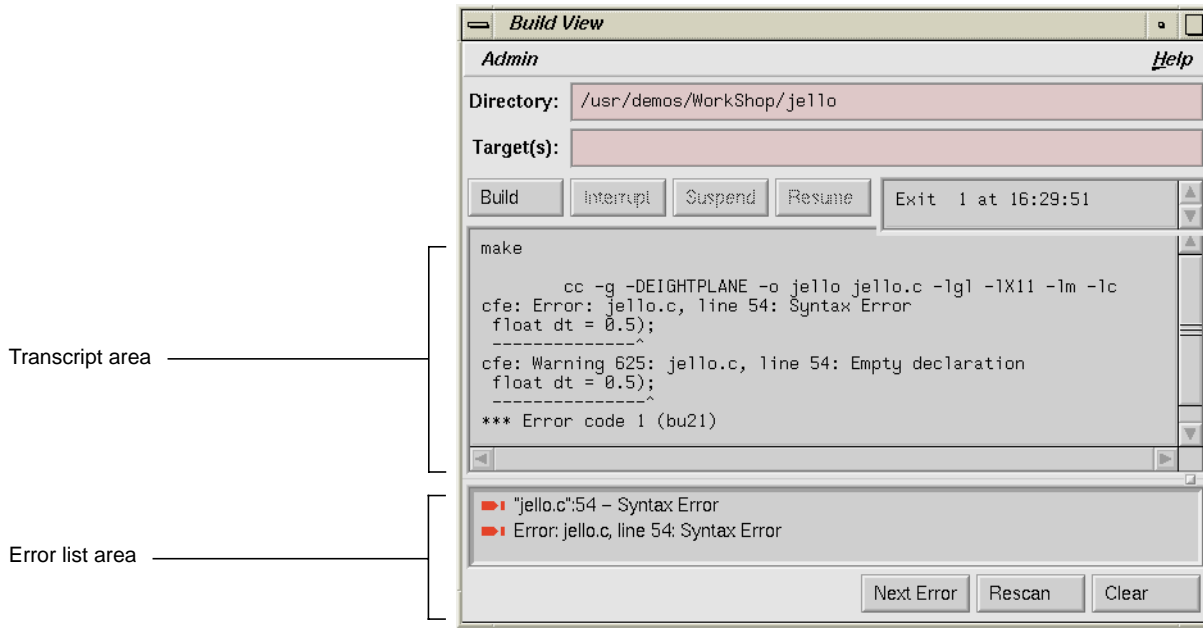


Figure B-3 Build View Window With Typical Data

Error List Area

The *error list area* consists of the error list display and three control buttons. The buttons are:

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 B-3). 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.



Figure B-4 Admin Menu in Build View Window

Build View Admin Menu

The Admin menu in Build View (see Figure B-4) has two selections in addition to the standard WorkShop entries:

- “Build View Preferences”
- “Build Options”

For more information on “Launch Tool” and “Project” menu selections, refer to the section “Admin Menu” on page 138.

Build View Preferences

The “Preferences...” selection brings up the dialog box shown in Figure B-5. 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.



Figure B-5 Build View Preferences Dialog Box

Build Options

The Build Options dialog box lets you add the options shown in Figure B-6 to your *make* command.

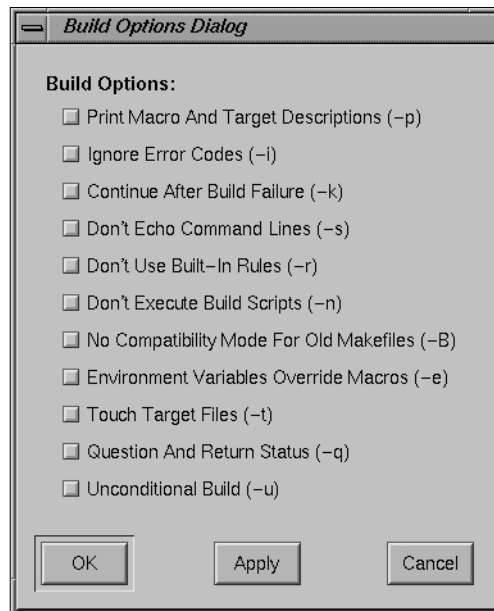


Figure B-6 Build Options Dialog Box

Using Build View

The steps in running a compile using Build View are:

1. Bring up Build View.
2. Edit the *Targets* 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.

Build Analyzer

Build Analyzer 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 B-7 with the menus displayed.

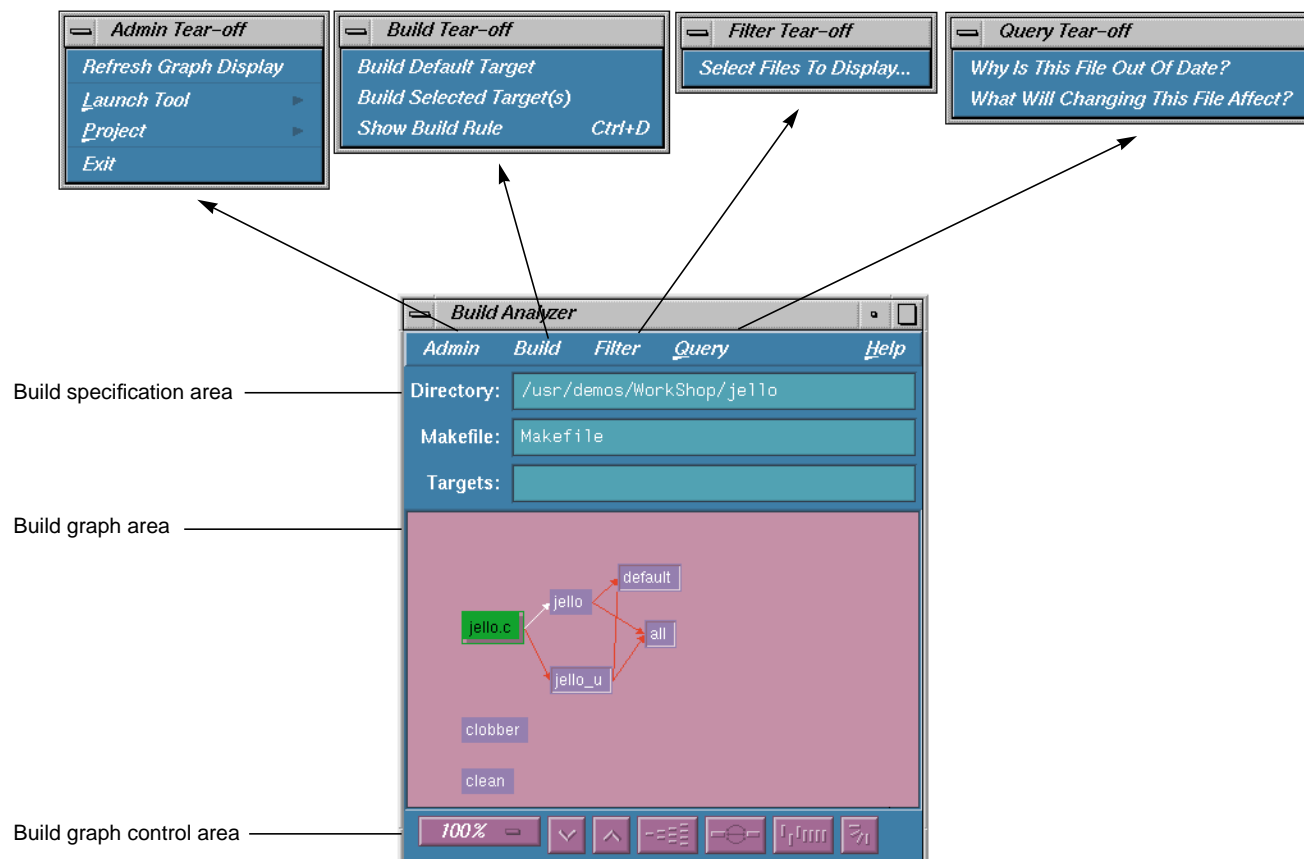


Figure B-7 Build Analyzer Window

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 *Targets* field directly. The *Targets* field also lets you specify a search string for locating a file in the build graph.

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 B-8.

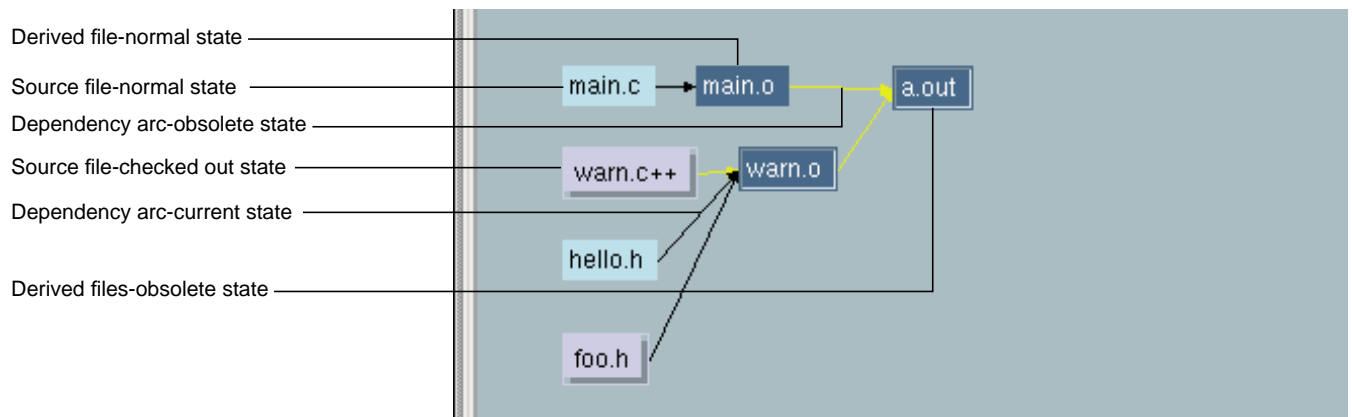


Figure B-8 Build Graph Icons

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.

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 B-9.

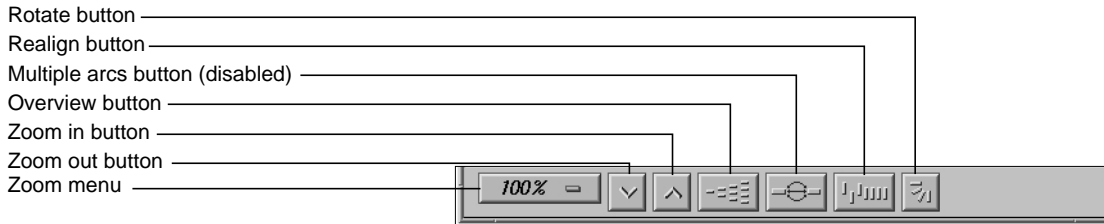


Figure B-9 Build Graph Control Area

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 B-9).

Figure B-10 shows a typical Overview window with the resulting graph in Build Analyzer. The Overview 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 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 the Overview window.

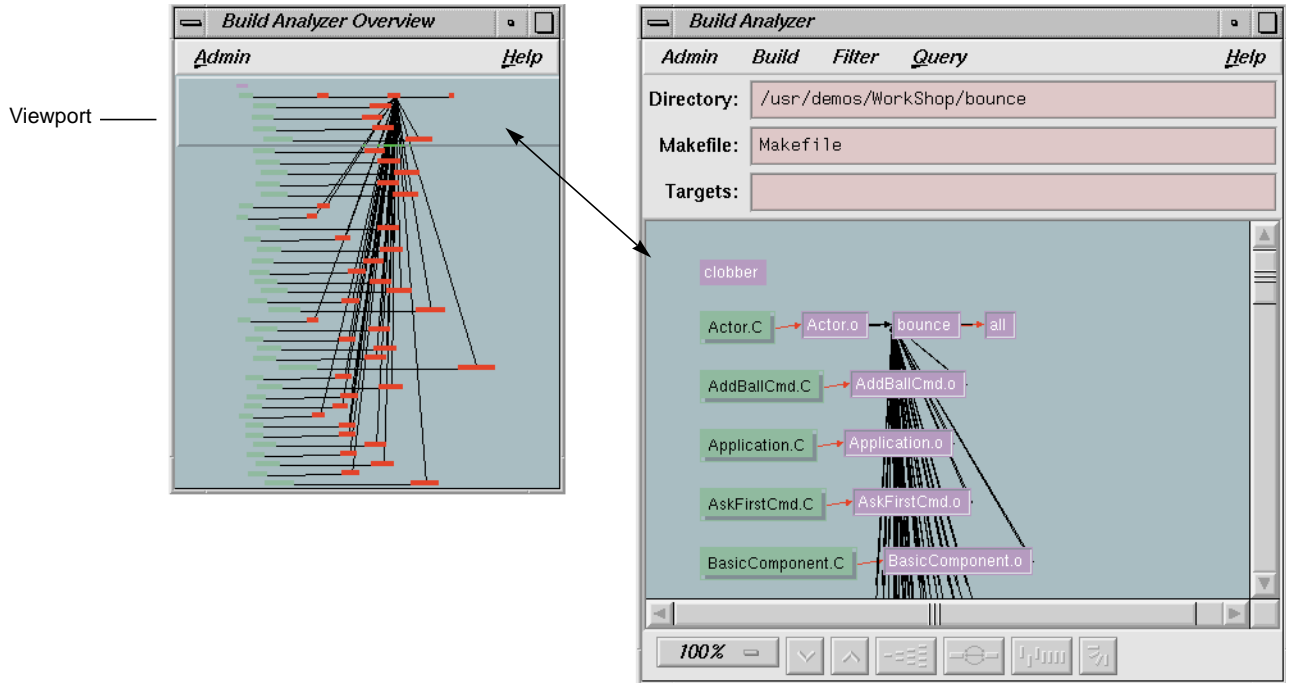


Figure B-10 Overview Window With Resulting Build Analyzer Graph

Build Analyzer Menus

Build Analyzer contains these menus:

- Admin
- Build
- Filter
- Query

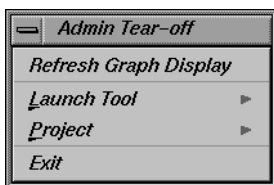


Figure B-11 Admin Menu in Build Analyzer

Build Analyzer Admin Menu

The Admin menu (Figure B-11) 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 “Admin Menu” on page 138.
- “Project” lets you control the WorkShop tools operating on the same executable as a group. For more information, see the section “Admin Menu” on page 138.

Build Menu

The selections in the Build menu (see Figure B-12) 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 *Targets* field.
- “Show Build Rule” displays a dialog box showing the makefile line for the selected node.



Figure B-12 Build Menu

Filter Menu

The Filter menu has only one selection:

- “Select files to show in graph” lets you enter a regular expression to filter the files displayed in the build graph. The File Filter dialog box appears in Figure B-13.

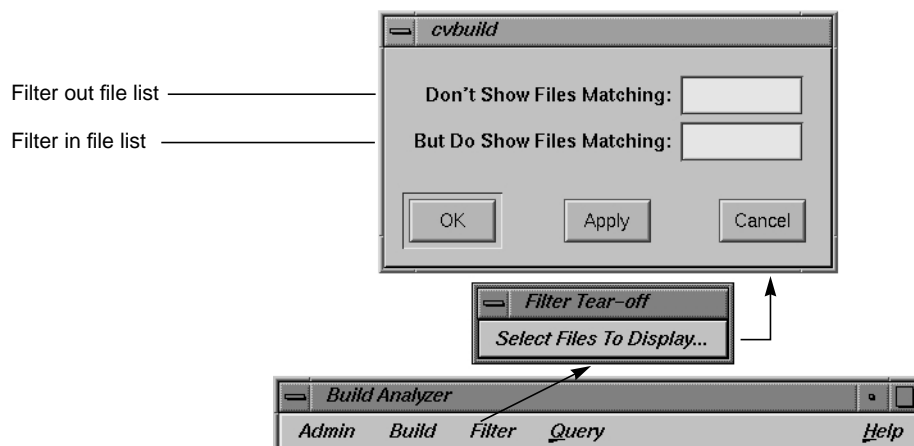


Figure B-13 Filter Dialog Box

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.

Query Menu

The Query menu lets you request information about the build graph (see Figure B-14). The selections are:

“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.



Figure B-14 Query Menu

Index

A

- accessing files, 21
- access to freed memory, 89
- Access to uninitialized memory, 89
- “Active” selection in Admin menu, 6, 35
- active toggle, 167, 172
- Add button in Trap Manager, 33
- adding a breakpoint, 123
- “Address...” selection in Disassemble menu, 241
- Admin menu, 257
 - Debugger views, 5
 - general description, 138
 - “Library Search Path...”, 138
- admin menu, 172
 - active toggle, 167, 172
 - clone, 167, 172
 - close, 167, 172
 - save as text, 167, 172
- alias debugger command, 268
- all trap Debugger command option, 49
- arguments, command line, 56
- “Arrange” selection in Structure Browser Display menu, 230
- Array Browser, 4, 40
 - general description, 209
 - subscript controls, 41
- “Array Browser” selection in Views menu, 40, 142
- Array field in Array Browser, 210
- array subscripts, 211

- array variables, 4, 209
- assign Debugger command, 268
- assigning values to variables, 67
- attach Debugger command, 268
- Automatic Dereference Limit field in Structure Browser Preferences box, 234

B

- boundary overruns, 89
- boundary underrun, 89
- breakpoint, 4
- breakpoint, adding, 123
- breakpoint results, viewing, 129
- breakpoints, setting, 81
- breakpoints, setting for a class, 124
- breakpoints examiner, 174
 - callback, 176
 - event-handler, 178
 - input-handler, 184
 - resource-change, 180
 - state-change, 185
 - timeout-procedure, 182
 - X-event, 188
- breakpoints tab, 128
- breakpoint type option button, 176
- Build Analyzer, 286-293
- Build Environment window, 262
- Build Manager, 279-293

build path, 12

Build View, 279-286

C

C++ expressions, 69

callback breakpoints examiner, 176

callback context, viewing, 125

callback examiner, 125, 196

call Debugger command, 268

Call Stack, 4

call stack, 83

“Call Stack” selection in Views menu, 34, 142

Call Stack View, 35, 61, 221

call stack view, 266

callstack view, 129

catch Debugger command, 268

C expressions, 68

C function calls, 68

change ID, 12

changes, re-enabling, 85

classes, examining widget, 124

“Clear All” selection in Structure Browser Display menu, 231

clearbuffer Debugger command, 268

Clear button in Trap Manager, 33

clearcalls Debugger command, 268

clear Debugger command, 268

“Clear Trap” selection in Traps menu, 46, 154

“Click for Help” selection in Help menu, 159

clone current window, 167, 172

“Clone” selection in Admin menu, 6

close current window, 167, 172

code, changing, 77

code, changing from command line, 79

code, comparing, 84

code, deleting changed, 79

code, switching between compiled and redefined, 85

Col button in Array Browser, 41

“Column Width...” selection in Array Browser Display menu, 213

Command field in Main View, 56

command line interface, 266

comparing function definitions, 85

Condition field in Trap Manager, 51

Config menu in Structure Browser, 227

Config menu in Trap Manager, 48

cont in Debugger command, 268

Continue button in Main View, 32, 34, 56

continue Debugger command, 268

“Continue To” selection in Disassembly View PC menu, 240

“Continue To” selection in PC menu, 59, 154

cont to Debugger command, 268

conventions, font, for manual, xxix

corefile Debugger command, 268

cvd

Execution View, 119

Main View, 118

Cycle Count field in Trap Manager, 34, 52

D

data structures, 4

dbx commands, 267

Debugger

call stack view, 266

changes to views, 264

command line interface, 266

exiting, 2

main view, 264

- process execution control, 55-59, ??-160
 - starting, 1, 27
 - trap manager, 266
 - debugger
 - Execution View, 119
 - Main View, 118
 - Debugger, exiting, 87
 - Debugger command, 268
 - Debugger command line, 3, 267-277
 - process execution commands, ??-59, ??-160
 - Debugger data, 61-??
 - Debugger views, 61, 221, 236
 - Debugger with Fix and Continue support
 - Fix and Continue
 - Debugger support with, 12
 - Default Field Count field in Structure Browser Type Formatting, 235
 - Default Iconic Width field in Structure Browser Preferences box, 234
 - Default Iconic Width field in Structure Browser Type Formatting box, 235
 - Default State field in Structure Browser Type Formatting box, 235
 - Default Structure Field Count field in Structure Browser Preferences box, 234
 - Default Structure Width field in Structure Browser Preferences box, 234
 - Default Structure Width in Structure Browser Type Formatting box, 235
 - delete all Debugger command, 269
 - delete Debugger command, 269
 - delete trap Debugger command, 269
 - Dereference Ptrs By Default field in Structure Browser Preferences box, 234
 - “Dereference Ptrs” selection in Structure Browser Node menu, 233
 - detach, 269
 - “Detach” selection in Admin menu, 140
 - “Detail” selection in Structure Browser submenu, 232
 - difference tools, 86
 - disable all Debugger command, 269
 - disable Debugger command, 269
 - disabling traps, 33
 - disassembled code, 4
 - Disassemble File dialog box, 242
 - Disassemble Function dialog box, 242
 - Disassemble menu in Disassembly View, 241
 - Disassembly View, 4
 - preferences, 242
 - “Disassembly View” selection in Views menu, 142
 - display area in Structure Browser, 229
 - display Debugger command, 269
 - Display menu
 - Main View, 147, 291
 - Display menu in Structure Browser, 227, 230
 - Display menu in Traps Manager, 48
 - “Display” selection in Structure Browser Display menu, 230
 - documentation, recommended reading, xxviii
 - double frees, 89
 - down Debugger command, 270
 - dump Debugger command, 270
- E**
- enable all Debugger command, 270
 - enable trap Debugger command, 270
 - environment variables
 - setting, 59, 160
 - erroneous frees, 89
 - Error Message window, 260
 - event examiner, 197

- event-handler breakpoints examiner, 178
 - examine menu, 173
 - selection, 173
 - widget, 173
 - widget class, 173
 - X event, 173
 - examiner
 - breakpoint, 123
 - breakpoints, 174
 - callback, 125, 196
 - callback breakpoints, 176
 - event, 197
 - event-handler breakpoints, 178
 - graphics context (GC), 198
 - input-handler breakpoints, 184
 - pixmap, 199
 - resource-change breakpoints, 180
 - state-change breakpoints, 185
 - timeout-procedure breakpoints, 182
 - trace, 191
 - tree, 121, 194
 - widget, 120, 193
 - widget class, 200
 - window, 126, 196
 - X-event breakpoints, 188
 - examiner menu
 - X graphics context, 174
 - X Pixmap, 174
 - examiners
 - overview, 14
 - selections, 15
 - examiner tabs, 174
 - examining Debugger data, 4, 61
 - Examining view data, 34
 - examining widget classes, 124
 - examining widgets, 122
 - examin menu
 - widget tree, 173
 - “Exception View” selection in Views menu, 142
 - execution control buttons, 56-58
 - Execution View, 59, 119, 160
 - “Execution View” selection in Views menu, 142
 - exiting Debugger, 87
 - exiting the Debugger, 2
 - exiting X/Motif analyzer, 130
 - “Exit” selection in Admin menu, 142
 - Expression column in Expression View, 65, 224
 - expression count Debugger command, 270
 - Expression field in Structure Browser, 38, 229
 - expressions
 - C, 68
 - C++, 69
 - Fortran, 70
 - “Expression” selection in Structure Browser Display submenu, 230
 - Expression View, 4, 37, 65, 223
 - “Expression View” selection in Views menu, 37, 143
- ## F
- File Browser, 21
 - “File Browser” selection in Views menu, 143
 - file Debugger command, 270
 - File Menu, Source View, 161
 - files
 - opening, 22
 - files, comparing source code, 86
 - files, finding, 12
 - “File...” selection in Disassemble menu, 242
 - finding files, 12
 - Fix+Continue menu, 258
 - Fix and Continue
 - basic cycle, 8
 - breakpoints, 81
 - Build Environment window, 262

build path, 12
change ID, 12
changing code, 77
changing code from command line, 79
deleting changed code, 79
editing a function, 76
environment, 11
Error Message window, 260
functionality, 8
GUI, 254
GUI command line, 12
menu operations, 154
redefining functions with, 7
restrictions, 10
sample session, 73
Session, 155, 258
Show Difference, 155
starting, 7
Status window, 84, 255
traps, 81
View, 156
WorkShop integration, 9
font conventions, for manual, xxix
"Fork Editor" selection in Source menu, 5, 145
Format menu in Expression View, 37, 65, 223, 224
Format menu in Structure Browser, 228
Format menu in Variable Browser, 237
formatting fields in Structure Browser, 233
Fortran expressions, 70
Fortran function calls, 71
Fortran variables, 70
frames, 61, 221
func Debugger command, 271
function, editing, 76
function, redefining
 Fix and Continue
 redefining functions, 75
function definitions, comparing, 85

functions, identifying, 12
"Function..." selection in Disassemble menu, 241

G

gdiff, 86
"Geometry" selection in Structure Browser Node menu, 232
givenfile Debugger command, 271
GLdebug, 139
"GLdebug" selection in Admin menu, 139
goto Debugger command, 271
Goto dialog box, 146
"Go to Line..." selection in Source menu, 146
graphics context (GC) examiner, 198
GUI command line, 12

H

heap corruption
 detection, 89-98
heap corruption problems
 defined, 89
Help menu, 159
"Hide Icons" selection in Display menu, 148
"Hide Line Numbers" selection in Display menu, 147

I

"Iconic" selection in Structure Browser submenu, 232
"Iconify" selection in Admin menu, 141
"Iconify" selection in Session menu, 202
identifying functions, 12

ignore Debugger command, 271
index identifiers in Array Browser, 212
Indexing Expression field in Array Browser, 211
index maximum specification in Array Browser, 212
index minimum specification in Array Browser, 212
"Index..." selection in Help menu, 159
index sliders in Array Browser, 212
index values in Array Browser, 212
input-handler breakpoints examiner, 184
"Insert Source..." selection in Source menu, 145
integration of WorkShop tools, 6
interface, command line, 266

J

jello program, 27
"Jump To" selection in Disassembly View PC menu, 240
"Jump To" selection in PC menu, 59, 154

K

"Keys & Shortcuts" selection in Help menu, 159
Kill button in Main View, 56
kill Debugger command, 271

L

Language menu in Expression View, 37, 65, 223, 224
Language menu in Variable Browser, 237
launching the X/Motif analyzer, 13
launching X/Motif analyzer, 119
\$LD_LIBRARY_PATH, setting, 14
Library Search Path dialog box, 138

"Linked List" selection in Structure Browser Display menu, 231
list Debugger command, 271
"Load Expressions..." selection in Expression View Config menu, 67
"Load Settings..." selection in Admin menu, 140
"Load Traps..." selection in Config menu in Trap Manager, 53

M

Main View, 118
 Command field, 56
 Continue button, 56
 control panel, 55-58
 Display menu, 147, 291
 general description, 2
 Kill button, 56
 menus, ??-159
 PC Menu, 59
 PC menu, 59
 Run button, 56
 Sample button, 58
 Status field, 56
 Step Into, 56
 Step Over button, 57
 Stop button, 56
main view, Debugger, 264
"Make Editable" in Source menu, 5
"Make Editable" selection in Source menu, 145
"Make Read Only" selection in Source menu, 145
managing source files, 21, 25
"Maximize" selection in Structure Browser Node submenu, 232
memory locations, 4
Memory View, 4, 248
Memory View Mode menu, 249
"Memory View" selection in Views menu, 143

menu operations, 154
Message window, 260
 Admin menu, 262
 buttons, 261
 View menu, 262
"Minimize" selection in Structure Browser Node submenu, 232
Minimum lines around current instruction field in Disassembly View Preferences box, 243
multiprocess traps, 48
"Multiprocess View..." selection in Admin menu, 139

N

next Debugger command, 271
nexti Debugger command, 271
Node menu in Structure Browser, 228, 232
Node popup menu in Structure Browser, 229
"Normal" selection in Structure Node submenu, 232
"N..." selection in Step Into menu, 56
"N..." selection in Step Over menu, 57
Number of instructions to disassemble field in Disassembly View Preferences box, 243

O

opening files, 22
"Open..." selection in Source menu, 145
"Overview" selection in Help menu, 159

P

path remapping, 23
"Pattern Layout" in Structure Browser Node menu, 233

PC, 154
PC menu, 59, 154
 "Continue To", 59
 "Jump To", 59
PC menu in Disassembly View, 240
PC menu in Main View, 59
performance data
 Sample button, 58
pgrp trap Debugger command option, 49
pixmap examiner, 199
pollpoint, 4
pollpoint trap Debugger command option, 50
Preference menu, 156
preparing the fileset, 117
printd expression Debugger command, 272
print expression Debugger command, 272
printo expression Debugger command, 272
printregs Debugger command, 272
printx expression Debugger command, 272
process execution control, 55-59, ??-160
 Main View control panel, 55-58
 PC menu, 59
"Process Meter" selection in Views menu, 143
"Product Information" selection in Help menu, 159
program counter, 59, 154
program output, tracking, 13
"Project" selection in Admin menu, 141
pwd Debugger command, 272

Q

quit Debugger command, 272

- R**
- “Raise” selection in Admin menu, 141
 - “Raise” selection in Session menu, 202
 - “Raise” selection in Structure Browser Node submenu, 233
 - Read-Only
 - Debugger status, 12
 - “Recompile” selection in the Source menu, 145
 - redefining functions, 7
 - Register name display format field in Disassembly View Preferences box, 243
 - registers, 4
 - Register View, 4, 244
 - Register View formatting, 247
 - Register View Preferences dialog box, 247
 - “Register View” selection in Views menu, 143
 - Register View window, 245
 - “Remap Paths...” selection in Session menu, 23
 - “Remove” selection in Structure Browser Node menu, 233
 - removing traps with mouse, 47
 - rerun Debugger command, 273
 - resource-change breakpoints examiner, 180
 - restrictions and limitations, 16
 - Result column in Expression View, 65, 224
 - Return button in Main View
 - Main View
 - Return button, 58
 - return Debugger command, 273
 - row/column toggles in Array Browser, 212
 - Run button in Main View, 56
 - run Debugger command, 273
- S**
- “Sample At Function Entry” selection in Traps submenu, 46, 153
 - “Sample At Function Exit” selection in Traps submenu, 46, 154
 - Sample button in Main View, 58
 - sample session, 117
 - Interpreter, 73
 - preparing fileset, 117
 - setting up, 117
 - sample session setup, 73
 - sample trap, 45
 - sample trap command, 48-50
 - sample traps, 3
 - “Save As...” selection in Source menu, 145
 - save as text, 167, 172
 - “Save as Text...” selection in Admin menu, 6
 - “Save As Text...” selection in Source menu, 145
 - “Save Expressions...” selection in Expression View Config menu, 67
 - “Save” selection in Source menu, 145
 - “Save Settings...” selection in Admin menu, 141
 - “Save Traps...” selection in Config menu in Trap Manager, 53
 - saving to source file, 80
 - saving view data, 5
 - Search field in Trap Manager, 53
 - “Search...” selection in Source menu, 145
 - “Search” selection in Source menu, 29
 - “Search” selection in Structure Browser Display menu, 231
 - selection, 173
 - “Selection” selection in Structure Browser Display menu, 230
 - “Select” selection in Structure Browser Node menu, 233

- Session submenu, 155, 258
- setting traps, 31, 54
- setting traps with the mouse, 47
- “Set Trap” selection in Traps menu, 46, 153
- sh Debugger command, 275
- Show Difference submenu, 155
- Show embedded source annotation field in Disassembly View Preferences box, 244
- “Show Icons” selection in Display menu, 148
- Show instruction value field in Disassembly View Preferences box, 244
- Show jal target numerically field in Disassembly View Preferences box, 244
- “Show Line Numbers” selection in Display menu, 147
- Show machine address field in Disassembly View Preferences box, 244
- “Show Overview” selection in Structure Browser Display menu, 231
- Show source file and line number field in Disassembly View Preferences box, 244
- continue, 268
- Signal Panel, 53
- “Signal Panel” selection in Views menu, 143
- signals
 - traps, 4
- signal trap Debugger command option, 50
- source annotation column
 - traps, 46
- source code display area, 2
- source code status indicator, 12, 74
- source Debugger command, 275
- source file, saving to, 80
- source files
 - managing, 21-25
- Source View
 - File Menu, 161
 - “Source View” selection in Views menu, 143
- special libraries, 14
- specifying traps, 54
- spreadsheet area in Array Browser, 212
- stack frame, 35
- stack frames, 61, 221
- starting, process execution, 56
- starting Fix and Continue, 7
- starting the Debugger, 1, 27
- starting the X/Motif analyzer, 13
- start trap command, 50
- state-change breakpoints examiner, 185
- “State” selection in Structure Browser Node menu, 232
- status, viewing, 84
- status Debugger command, 275
- Status field in Main View, 56
- status line
 - Main View, 3
- Status window, 84, 255
 - Admin menu, 257
 - Fix+Continue menu, 258
 - Preference menu, 156
 - View menu, 258
- step Debugger command, 275
- stepi Debugger command, 275
- step indicators in Array Browser, 212
- Step Into button in Disassembly View, 240
- Step Into button in Main View, 56
- Step Over button in Disassembly View, 240
- Step Over button in Main View, 57
- stop at Debugger command, 275
- “Stop At Function Entry” selection in Traps submenu, 46, 153
- “Stop At Function Exit” selection in Traps submenu, 46, 153

Stop button in Main View, 56
stop in Debugger command, 276
stopping, process execution, 56
stop trap, 45
stop trap command, 48
stop traps, 3, 31
Structure Browser, 4
 general description, 226
Structure Browser Preferences dialog box, 234
"Structure Browser" selection in Views menu, 38, 143
subscript controls In Array Browser, 41
subscripts
 array, 211
"Switch Executable..." selection in Admin menu, 140
Switch Process dialog box, 140
"Switch Process..." selection in Admin menu, 139
syscall Debugger command, 276
Syscall Panel, 53
"Syscall Panel" selection in Views menu, 143
syscall trap Debugger command option, 50
system calls
 traps, 4

T

tab overflow area, 128
tabs, 128
tabs, examiner, 174
"Task View" selection in Views menu, 143
timeout procedure breakpoints examiner, 182
trace Debugger command, 276
trace examiner, 191
tracking program output, 13
trap actions, 44

trap condition, 51
trap examples, 50
Trap Manager, 32
trap manager, 83
Trap Manager menus, 48
"Trap Manager" selection in Views menu, 143
trap manager, 266
traps, 3, 31
 disabling, 33
 general description, 43-54
 one-time, 59
 removing with mouse, 47
 setting conditions, 51
 setting cycle count, 52
 setting with mouse, 47
 Signal Panel, 53
 Syscall Panel, 53
 triggering, 44
traps, setting, 81
Traps menu in Main View, 45
Traps menu in Trap Manager, 48, 52
traps multiprocess, 48
trap terminology, 44
tree examiner, 121, 194
"Tree" selection in Structure Browser Display submenu, 230
triggering traps, 44
Type Color field in Structure Browser Type Formatting box, 236
Type Formatting dialog box, 235
Type Name field in Structure Browser Type Formatting box, 235

U

unalias Debugger command, 276
undisplay Debugger command, 276

“Update” selection in Structure Browser Display menu, 231
updating view data, 5
up Debugger command, 276
use Debugger command, 276
using
 Interpreter, 73
using the X/Motif analyzer, 14

V

Variable Browser, 4, 35
 general description, 237
“Variable Browser” selection in Views menu, 35, 143
variables
 assignment, 67
view, call stack, 266
view changes in Debugger, 264
view data, 5
viewing status, 84
View menu, 258
views
 Debugger, 236
Views menu in Main View, 142
View submenu, 156

W

watch command, 33
watchpoint, 33
watchpoints, 4
watch trap Debugger command option, 49
whatis Debugger command, 277
when at Debugger command, 277
when in Debugger command, 277

where Debugger command, 277
which Debugger command, 277
widget classes, examining, 124
widget class examiner, 200
widget class menu item, 173
widget examiner, 193
widget hierarchy, 121
widget item, 173
widgets, examining, 122
widget structure, navigating, 120
widget tree menu item, 173
window attributes, viewing, 126
window examiner, 126, 196
window menu item
 examine menu
 window, 173
WorkShop integration, 9
“Wrapped Display” selection in Array Browser Display menu, 213

X

X-event breakpoints examiner, 188
X event menu item, 173
X graphics context menu item, 174
X/Motif analyzer
 launching, 13
X/Motif analyzer
 additional features, 126
 default view, 120
 exiting, 130
 launching, 119
 navigating widget structure, 120
 restrictions and limitations, 16
 sample session, 117
 starting, 13
 using, 14

“X/Motif Analyzer” selection in Views menu, 144

X Pixmap menu item, 174

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 007-2879-003.

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