Debugging a Program

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-7105-10
Revision A, August 1994

© 1994 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, Sun Microsystems Computer Corporation, the Sun Microsystems Computer Corporation logo, SunSoft, the SunSoft logo, ProWorks, ProWorks/TeamWare, ProCompiler, Sun-4, SunOS, Solaris, ONC, ONC+, NFS, OpenWindows, DeskSet, ToolTalk, SunView, XView, X11/NeWS, AnswerBook, and Magnify Help are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark of Novell, Inc., in the United States and other countries; X/Open Company, Ltd., is the exclusive licensor of such trademark. OPEN LOOK® is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun® Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox® Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a trademark and product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Please
Recycle

Contents

Preface xvii

Part 1 — Overview of the Debugger

1. Introduction to the Debugger 1-3

1.1 Solaris 2.x and Solaris 1.x 1-3

1.1.1 Summary of Version Differences 1-4

1.2 The Debugger and dbx 1-4

1.3 How the Debugger Works 1-5

1.3.1 Loading a Program or Attaching to a Process . . . 1-6

1.3.2 Main Features of the Debugger 1-6

1.3.3 Interaction of dbx with the Dynamic Linker 1-8

1.3.4 Debugging Optimized Code 1-11

1.3.5 History Facility 1-11

1.4 Graphical Overview 1-11

1.4.1 Menu Items 1-13

1.4.2 Information Fields 1-13

1.4.3 Source Display 1-13
1.4.4 Button Commands 1-14
1.4.5 Command Pane 1-14
1.4.6 Program Input/Output Window (PIO) 1-16
1.4.7 Debugger Properties Window Panes 1-18
1.4.8 The Debugger Initialization Files 1-18
1.5 Example Application Program 1-19
Part 2 — Basic Debugging
2. Preparing Programs 2-23
2.1 Compiling with the -g Option 2-23
2.1.1 Limited Support for Code Compiled Without -g Option 2-23
2.1.2 Shared Libraries Need -g for Full Debugger Support 2-24
2.1.3 C++ Support and the -g Option 2-24
2.1.4 Completely Stripped Programs on Solaris 2.x . . . 2-24
2.2 Using the Auto-Read Facility 2-24
2.2.1 Auto-Read is the Default for the Debugger 2-25
2.2.2 Disabling Auto-Read with the -xs Compiler Option 2-25
2.2.3 Listing Modules That Have Been Read In 2-26
2.3 Debugging Optimized Code 2-26
3. Starting the Debugger 3-29
3.1 Starting a Debugging Session 3-29
3.1.1 Starting the Debugger from the Manager 3-30
3.1.2 Starting the Debugger from a Command Tool . . 3-31

3.1.3 If a Core File Exists 3-32
3.1.4 Quitting a Debugging Session 3-32
3.1.5 Killing a Program Without Terminating the Session 3-32
3.1.6 Attaching the Debugger to a Process 3-33
3.1.7 Detaching a Process from the Debugger 3-33
3.1.8 Debugger Start-up Sequence 3-33
3.1.9 Running dbx Alone from a Shell 3-34
3.2 Setting the Source/Object File Search Path 3-34
3.2.1 Setting the Search Path in the Initialization File . 3-36
3.2.2 Setting the Search Path with `pathmap` 3-37
3.3 Listing and Reading-in Program Modules 3-37
3.4 Options to the Debugger Start-up Command 3-38
4. Viewing and Visiting Code 4-43
4.1 Knowing the State of the Program 4-44
4.2 Reading the Information Fields 4-44
4.2.1 Directory 4-44
4.2.2 Stopped in File…Function…Line 4-46
4.2.3 Currently in File…Function…Lines 4-46
4.3 Visiting Code 4-47
4.3.1 Visiting a Function 4-47
4.3.2 Selecting from a List of C++ Ambiguous Function Names 4-48
4.3.3 Redisplaying the Line Where the Program is Stopped 4-49
4.3.4 Using `list` to Display a Function 4-50

4.3.5 Visiting a File 4-50
4.3.6 Walking the Call Stack to Visit Code 4-51
4.4 Visiting Functions from the Command Pane 4-51
4.4.1 Qualifying Symbols with Scope Resolution Operators 4-52
4.4.2 Linker Names 4-53
4.4.3 Ambiguous or Overloaded Function Names in a Command Pane 4-54
4.4.4 Scope Resolution Search Path 4-54
4.5 Locating Symbols: the `whereis` and `which` Commands 4-55
4.5.1 `whereis`: Printing a List of Occurrences of a Symbol 4-55
4.5.2 Seeing a Preview of Which Symbol the Debugger Will Use 4-56
4.6 Looking Up Variables, Members, Types, and Classes . 4-57
4.6.1 Displaying Declarations of Variables and Members 4-57
4.6.2 Looking Up Definitions of Types and Classes. . . 4-59
4.6.3 Using `whatis` to See Inherited Members 4-60
5. Setting Breakpoints and Traces 5-63
5.1 Setting Breakpoints 5-64
5.1.1 Setting a Breakpoint at a Line of Source Code . . 5-66
5.1.2 Setting a Breakpoint in a Function 5-67
5.1.3 Setting a Breakpoint in a Dynamically Linked Library. 5-67
5.2 Setting Multiple Breaks in C++ Programs. 5-68

5.2.1 Set Breakpoints in Member Functions of Different Classes 5-68
5.2.2 Setting Breakpoints in Member Functions of Same Class 5-69
5.2.3 Setting Multiple Breakpoints in Nonmember Functions 5-69
5.2.4 Setting a `when` Type Breakpoint at a Line 5-69
5.3 Tracing Code 5-70
5.3.1 Setting `trace` Commands 5-70
5.3.2 Controlling the Speed of a `trace` 5-71
5.4 Listing and Clearing Event Management Commands. 5-72
5.4.1 Listing Breakpoints and Traces 5-72
5.4.2 Clearing a Breakpoint at a Line 5-72
5.4.3 Clearing a Breakpoint from the Current Focus Line 5-73
5.4.4 Deleting Specific Breakpoints Using Status ID Numbers 5-73
5.4.5 Watchpoints 5-73
5.5 Setting Event Filters 5-75
5.6 Event Efficiency 5-76
6. Event Management 6-79
6.1 Event Management Commands 6-80
6.2 Commands to Manipulate the Event Handlers 6-83
6.3 Event Counters 6-84
6.4 Event Specifications 6-84
6.5 Event Specification Modifiers 6-91
6.6 A Note on Parsing and Ambiguity 6-93

6.7 Additional Process Control Commands & Enhancements 6-94
6.8 Predefined Variables 6-95
6.8.1 Event-Specific Variables 6-96
6.9 Examples 6-98
6.9.1 Set Watchpoint for store to Array Member 6-98
6.9.2 Simple Trace. 6-98
6.9.3 Enable Handler While Within the Given Function (`in` func) 6-98
6.9.4 Determine the Number of Lines Executed in a Program 6-99
6.9.5 Determine the Number of Instructions Executed by a Source Line 6-99
6.9.6 Enable Breakpoint after Event Occurs 6-99
6.9.7 Set Automatic Breakpoints for `dlopen` Objects . 6-100
6.9.8 Reset Application Files for `replay` 6-100
6.9.9 Check Program Status 6-100
6.9.10 Catch Floating Point Exceptions 6-101
7. Running, Stepping, Continuing. 7-103
7.1 Running a Program in the Debugger. 7-103
7.1.1 Running a Program Without Arguments. 7-104
7.1.2 Adding Arguments to the `Run` Command 7-104
7.1.3 Redirecting Input and/or Output. 7-104
7.1.4 Rerunning a Program 7-105
7.1.5 Attaching the Debugger to a Running Process . . 7-105
7.1.6 Detaching a Process from the Debugger 7-106

7.2 Single-step Execution: `next`, `step` and `call` 7-107
7.2.1 Issuing Next and Step Commands 7-107
7.2.2 Calling a Function 7-108
7.3 Continuing a Program 7-109
7.3.1 Skipping Lines of Code with `cont at` *line* 7-110
7.3.2 Using `cont at <line>` with the `assign` Command 7-111
7.4 Using `Cntl-C` to Stop a Process 7-111
7.4.1 Generic Case 7-111
7.4.2 If the Process was Attached While It was Running 7-112
7.4.3 Continuing Again, After Stopping Using `^c` ... 7-112
8. Saving and Restoring a Debugging Run 8-113
8.1 Saving a Debugging Run 8-113
8.2 Restoring a Saved Run 8-115
8.3 Saving and Restoring with the `replay` Command ... 8-116
9. Examining the Call Stack 9-119
9.1 Examining the Stack 9-119
9.1.1 Walking the Stack and Returning Home 9-120
9.2 The Stack Inspector 9-122
9.2.1 The Stack Inspector Show Menu 9-123
9.2.2 The Stack Inspector Hide Menu 9-124
9.2.3 The Hide/Unhide Menu 9-125
9.3 Debugging a Core File 9-125
10. Evaluating and Displaying Data 10-127

10.1 Evaluating and Dereferencing Variables and Expressions 10-128
10.1.1 Printing the Value of a Variable or an Expression 10-129
10.1.2 Dereferencing Pointers 10-131
10.2 Monitoring Expressions 10-132
10.2.1 Turning Off Display (Undisplay) 10-133
10.3 Assigning a Value to an Expression 10-134
10.4 Evaluating Arrays 10-134
10.4.1 Array Slicing for FORTRAN Arrays 10-135
10.4.2 Syntax for FORTRAN Array Slicing and Striding 10-135
11. Visual Data Inspector 11-139
11.1 Starting the Visual Data Inspector (VDI) 11-139
11.2 The VDI Window 11-140
11.2.1 The View Menu 11-142
11.2.2 The Node Menu 11-143
11.2.3 The VDI Props Menu 11-143
11.2.4 The Inspect Menu 11-145
11.3 Examining Variables and Complex Structures 11-145
11.3.1 Starting an Inspection Session 11-145
11.3.2 The VDI Node 11-147
11.3.3 The VDI Display Pane 11-147
11.3.4 Array Inspection 11-148
12. Process/Thread Inspector 12-151
12.1 Introduction to the Multithreaded Debugger 12-151

12.2 Displaying the Process Inspector 12-152
12.3 The Process/Thread Inspector Window 12-152
12.3.1 Thread Information. 12-153
12.4 Viewing the Context of Another Thread 12-155
12.4.1 Resuming Execution. 12-158
12.5 LWP Information 12-158
13. Customizing the Debugger 13-161
13.1 Changing the Debugger Environment Attributes 13-163
13.2 Debugger Events 13-167
13.3 Changing the Debugger Window Configurations 13-168
13.4 Changing Settings for Run Time Check. 13-171
13.5 Setting the Source/Object File Search Path. 13-171
13.6 Setting Miscellaneous Properties 13-173
13.7 Using the Debugger Initialization File. 13-176
13.7.1 A Sample Initialization File 13-176
13.7.2 Setting the Graphical User Interface (GUI) Resources 13-177
13.8 Adding Buttons 13-178
13.8.1 The Button Command 13-178
14. Editing a Program 14-181
14.1 Starting an Editor in a Separate Command Tool 14-181
14.2 Enabling Editing in the Source Display. 14-182
14.2.1 Checking Out Files Under `SCCS` 14-182
14.2.2 Saving Changes, Disabling Editing 14-182

14.2.3 Discarding Changes, Disabling Source Display Editing 14-183

Part 3 — Advanced Debugging

15. Debugging at the Machine-Instruction Level 15-187

15.1 Examining the Contents of Memory 15-187

15.1.1 Using the `examine` Command 15-188

15.1.2 Using the `dis` Command. 15-190

15.1.3 Using the `listi` Command 15-191

15.2 Stepping and Tracing at Machine-instruction Level . . 15-192

15.2.1 Single-Stepping the Machine-Instruction Level . 15-192

15.2.2 Tracing at the Machine-Instruction Level 15-193

15.3 Setting Breakpoints at Machine-instruction Level 15-194

15.3.1 Setting a Breakpoint at an Address. 15-195

15.4 Sun-4 Register Information 15-196

15.5 x86 `Register Information` 15-197

16. Debugging Child Processes 16-199

16.1 Simple Attach 16-199

16.2 Follow `exec` (Solaris 2.x) 16-200

16.3 Follow `fork` Under dbx (tty Mode) (Solaris 2.x) 16-200

16.4 Follow `fork` Under the Debugger (GUI Mode) (Solaris 2.x) 16-201

16.5 Interaction with Events and Other Features 16-201

17. Working with System Signals. 17-203

17.1 Sequence of Events Involving Signals 17-203

17.2 Catching System Signals 17-204

17.2.1 Default `catch` and `ignore` Lists 17-204
17.2.2 Changing the Default `catch` and `ignore` Signal Lists 17-205
17.2.3 Trapping the FPE Signal under Solaris 2.x. 17-205
17.3 Sending a System Signal in a Program 17-206
18. Collecting Performance Tuning Data 18-207
18.1 Starting the Collector from the Debugger 18-208
18.1.1 Experiment. 18-209
18.1.2 Sample 18-209
18.2 Taking a Sample for Use with the Analyzer 18-210
18.2.1 Manual Sampling 18-210
18.2.2 Storing the Collected Data 18-211
19. Runtime Checking 19-213
19.1 Getting Started 19-215
19.2 Summary of Operation 19-215
19.2.1 Operation from dbx 19-216
19.2.2 Operation from the Debugger 19-217
19.2.3 Batch Mode Operation (`bcheck`) 19-218
19.3 Error Reporting. 19-219
19.4 Memory Access Error Checking 19-219
19.5 Memory Access Errors 19-220
19.5.1 Memory Access Error Reporting. 19-221
19.5.2 Read from Uninitialized Memory (`rui`) 19-222
19.5.3 Read from Unallocated Memory (`rua`) 19-222

19.5.4 Write to Unallocated Memory (wua) 19-222
19.5.5 Write to Read-Only Memory (wro) 19-223
19.5.6 Misaligned Read (mar) 19-223
19.5.7 Misaligned Write (maw) 19-223
19.5.8 Duplicate Free (duf) 19-224
19.5.9 Bad Free (baf) 19-224
19.5.10 Misaligned Free (maf) 19-225
19.5.11 Out of Memory (oom) 19-225
19.6 Using Memory Access Checking 19-225
19.7 Memory Leaks Checking 19-227
19.8 Memory Leak Errors 19-229
19.8.1 Memory Leak Error Reporting 19-229
19.8.2 Memory Leak (mel) 19-231
19.8.3 Address in Register (air) 19-231
19.8.4 Address in Block (aib) 19-232
19.9 Error Suppression 19-232
19.9.1 Error Suppression from dbx 19-233
19.9.2 Error Suppression from the Debugger 19-233
19.10 Command Syntax 19-237
19.10.1 Command check|uncheck 19-237
19.10.2 Command showleaks 19-238
19.10.3 Command suppress|unsuppress 19-239
19.10.4 dbxenv Variables 19-241
19.11 Running Commands from a Script 19-242

19.12 Functions Intercepted by Runtime Checking 19-243
19.13 Troubleshooting Tips 19-243
20. `fix` and `continue` 20-245
20.1 Basic Concepts 20-245
20.2 How `fix` and `continue` Operates 20-246
20.3 Source Modification Using `fix` and `continue` 20-246
20.4 Precautions 20-247
20.5 Example 20-248
20.6 Command Summary 20-250
21. Handling C++ Exceptions 21-253
21.1 Commands for Handling Exceptions 21-254
21.2 Exception Handling in the Debugger 21-255
22. Debugging with C++ Templates 22-259
22.1 Template Example 22-259
22.2 Debugger Commands on Templates 22-261
23. Korn Shell 23-267
23.1 Features of ksh-88 not Implemented 23-268
23.2 Extensions to ksh-88 23-268
23.3 Renamed Commands 23-269
23.4 The Debugger Startup Mode 23-269
Part 4 — Appendixes
A. Debugger Commands A-273
A.1 On-Line Help in the Command Pane A-273
A.2 Debugger Commands By Functional Groups A-273

B. Operators Recognized By the Debugger. B-277
B.1 Operators Recognized by the Debugger B-278
B.2 Precedence and Associativity B-279
C. A FORTRAN Quick Start C-281
C.1 Overview for FORTRAN C-281
C.2 Sample Program C-282
C.3 A Sample FORTRAN Session C-283
C.4 Calling a Function C-285
C.5 Structures and Pointers C-286
C.6 Parameters. C-288
C.7 `BLOCK DATA` Subprograms C-288
C.8 Passing Arguments to a Main Program. C-289
C.9 Miscellaneous Tips C-291
Index I-293

Preface

This manual explains how to use the Debugger, a multi-language source and machine level debugging tool.

Operating Environment

The SPARCworks™ and ProWorks™ toolsets run under the Solaris™ 2.x operating environments. SPARCworks also runs under Solaris™ 1.x. From a usage point of view, almost all of the aspects of the SPARCworks (under Solaris 2.x and 1.x) and ProWorks toolsets are the same. This includes functionality, behavior, usage, and features. For the few details that are different, the documentation calls out those differences.

For SPARCworks, Solaris 1.x implies:

- Solaris 1.2 (or later) operating environment
- SunOS™ 4.1.x operating system
- A SPARC® computer (either a server or a workstation)
- The OpenWindows™ 3.x application development platform

For SPARCworks, Solaris 2.x implies:

- Solaris 2.2 (or later) operating environment
- SunOS 5.2 (or later) operating system
- A SPARC® computer (either a server or a workstation)
- The OpenWindows 3.x application development platform

For ProWorks, Solaris 2.x for the x86 environment implies:

- Solaris 2.1 (or later) operating environment
- SunOS 5.1 operating system
- An 80x86-compatible computer
- The OpenWindows 3.x application development platform

Note – In this document, text that pertains to a specific operating environment is indicated by one of these names: Solaris 1.x, Solaris 2.x (includes SPARC and x86), SPARC, or x86.

The SunOS 5.x operating environment is based on the System V Release 4 (SVR4) UNIX[1] operating system, and the ONC™ family of published networking protocols and distributed services. The SunOS 4.1.x operating environment is based on the UCB BSD 4.3 operating system.

The Debugger is an integrated component of the toolset, which also includes:

(Solaris 2.x)

- Analyzer
- dbx
- FileMerge
- Maketool
- Manager
- SourceBrowser

Before You Begin

This manual is written for programmers who want to use the Debugger on workstations and computer systems running ANSI C, C++, FORTRAN, and Pascal.

This manual assumes you are familiar with

- Sun™ operating system commands and concepts
- The OPEN LOOK® interface and the OpenWindows™ environment, particularly the use of the mouse to activate a window, select text, and click on buttons

---

1. UNIX is a registered trademark of UNIX System Laboratories, Inc.

If you are not familiar with the OPEN LOOK interface, see *Managing the Toolset*. For more information on the OPEN LOOK GUI, see the *OPEN LOOK Application Style Guidelines*.

For more information on the OpenWindows environment, see the *OpenWindows Developer's Guide: User's Guide*.

How this Book is Organized

This manual is organized as follows:

Part 1— Overview of the Debugger

- **Chapter 1, "Introduction to the Debugger,"** is an overview of the user-interface and the capabilities of the Debugger.

Part 2—Basic Debugging

- **Chapter 2, "Preparing Programs,"** explains how to compile programs for use with the Debugger.
- **Chapter 3, "Starting the Debugger,"** describes how start and quit the Debugger, including options to the start-up command.
- **Chapter 4, "Viewing and Visiting Code,"** describes the information fields by which users navigate while debugging. It also explains how to visit code away from the current stop location and how to qualify the scope of the debugging command when working from the command pane; also, how to query for the location of symbols, and to display declarations and definitions of identifiers, types, and C++ classes.
- **Chapter 5, "Setting Breakpoints and Traces,"** describes how to use event management commands to stop the program execution at a specified location or when a specified condition arises, and how to trace program execution.
- **Chapter 6, "Event Management,"** describes the general capability of the Debugger to perform certain actions when certain events take place in the program being debugged.
- **Chapter 7, "Running, Stepping, Continuing,"** describes how to run a program with or without arguments, continue after stopping, and single-step through lines of code.

- **Chapter 8, "Saving and Restoring a Debugging Run,"** describes how to save a debugging run, and how to replay it later.
- **Chapter 9, "Examining the Call Stack,"** describes how to display functions currently on the stack.
- **Chapter 10, "Evaluating and Displaying Data,"** describes how to spot-check the values of expressions and variables and how to use the Data Display window to monitor expressions and variables as the program accesses them.

(Solaris 2.x)
- **Chapter 11, "Visual Data Inspector,"** describes how to examine program variables including complex structures, and how to monitor values during program execution.

(Solaris 2.x)
- **Chapter 12, "Process/Thread Inspector,"** describes how to find information about threads by using the Process/Thread Inspector.
- **Chapter 13, "Customizing the Debugger,"** describes how to set controls for customizing attributes of the debugging environment: including how to use the initialization file, `.dbxrc`.
- **Chapter 14, "Editing a Program,"** describes how to edit source files without leaving the Debugger. You can start up an edit session in a new shell or convert the source display to an editable text pane.

Part 3 —Advanced Debugging

- **Chapter 15, "Debugging at the Machine-Instruction Level,"** explains the use of the machine-instruction variants of several event management commands and how to display the contents of memory.
- **Chapter 16, "Debugging Child Processes,"** shows how to debug processes from a simple attach to processes under Solaris 2.x that create children via fork(2) and exec(2).
- **Chapter 17, "Working with System Signals,"** describes how to control the signals the Debugger will catch.

(Solaris 2.x)
- **Chapter 18, "Collecting Performance Tuning Data,"** shows you how to take a manual sample of program data for performance analysis.
- **Chapter 19, "Runtime Checking,"** describes how to automatically detect run time errors in an application during the development phase.

- **Chapter 20, "fix and continue,"** describes how to fix (change) a file, recompile it with the same options, install the new code in the program, and continue.
- **Chapter 21, "Handling C++ Exceptions,"** describes how to set up try blocks to catch exceptions that have been raised by throw expressions defined earlier in the code.
- **Chapter , "Debugging with C++ Templates,"** describes how to load programs containing class and function templates into the Debugger and invoke any of the debugging commands on a template that you would use on a class or function.
- **Chapter 23, "Korn Shell,"** describes the Debugger command language, which is based on the Korn shell and includes I/O redirection, loops, built-in arithmetic, history, and command-line editing.

Part 4 —Appendixes

- **Appendix A, "Debugger Commands,"** is a reference list of Debugger text commands that you may use in the command pane.
- **Appendix B, "Operators Recognized By the Debugger,"** contains two tables that describe operators, their precedence and associativity.
- **Appendix C, "A FORTRAN Quick Start,"** introduces some features that are useful in FORTRAN.

What Typographic Changes and Symbols Mean

The following table describes the notational conventions and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
♦	A single-step procedure	♦ **Click on the Apply button.**

Code samples are included in boxes and may display the following:

`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

How to Get Help

Help tools include the following on-line help facilities:

- **AnswerBook® system** displays all tools manuals. You can read this manual on line and take advantage of dynamically linked headings and cross-references.

  To start the AnswerBook system, type: `answerbook &`

- **Magnify Help**™ messages are a standard feature of the OpenWindows software environment. If you have a question, place the pointer on the window, menu, or menu button and press the Help key.
- **Notices** are a standard feature of OPEN LOOK. Some notices inquire about whether or not you want to continue with an action. Others provide information about the end result of an action and appear only when the end result of the action is irreversible.
- **Manual Pages** (man pages) provide information about the command-line utilities of the SunOS operating system. Each tool has at least one man page.

  The Debugger manual pages include:

  ```
  debugger(1)
  dbx(1)
  dbxrc(4)
  bcheck(1)
  ```

♦ **To access the man pages, type:** `man` *utility_name*

- **Command Pane Help** lists all of the commands you can use in the command pane (or in a shell or tty terminal) when you type `help` at the prompt in the Debugger command pane. Type `help` *command_name* to display a brief help message for each Debugger command. Type `help -k` *word* to search for the use of *word* in the help information.

Related Documentation

This manual is part of a complete document set. Other manuals in this set include:

- *Installing SunSoft Developer Products Software on Solaris*
- *SPARCworks/ProWorks Tutorial*
- *Browsing Source Code*
- *Building Programs with MakeTool*
- *Managing the Toolset*
- *Merging Source Files*
- *Performance Tuning an Application*

You can find these and other related documents in the on-line AnswerBook system. To access the AnswerBook system, refer to *3.0.1 Installation AnswerBook*.

Part 1 — Overview of the Debugger

Introduction to the Debugger 1

The Debugger is an interactive, window-based, source code and machine-instruction level debugging tool. It provides facilities to run a program in a controlled fashion and to inspect the state of a stopped program. The Debugger gives you complete control of the dynamic execution of a program, including the collection of performance data (Solaris 2.x).

This overview chapter is organized into the following sections:

Solaris 2.x and Solaris 1.x	*page 1-3*
The Debugger and dbx	*page 1-4*
How the Debugger Works	*page 1-5*
Graphical Overview	*page 1-11*
Example Application Program	*page 1-19*

1.1 Solaris 2.x and Solaris 1.x

The Debugger runs under Solaris 2.x—which uses the SunOS 5.x operating system and Solaris 1.x—which uses the SunOS 4.x operating system.

Note – If the Debugger is started on a machine running Solaris 2.x, and the DISPLAY environment variable is set for a machine running Solaris 1.x, you must install patch 100626-05 on Solaris 1.x (SunOS 4.x) to allow the Debugger

to work properly. To kill the *ttsession* on the Solaris 1.x (SunOs 4.x) machine before starting the Debugger is an alternative that may not be feasible if an application requires *ttsession*. The application may hang or die.

Also set the `OPENWINHOME` environment variable to `/usr/openwin/bin` and place the path at the beginning of your `PATH` variable.

1.1.1 Summary of Version Differences

Here is a list of the Debugger differences between the Solaris 2.x environment and the Solaris 1.x environment.

- On-demand automatic reading in of debugging information: On Solaris 2.x, the Auto-Read facility can read in individual programs and library modules as needed; Solaris 1.x, can read in groups of interdependent modules. See Section 2.2, "Using the Auto-Read Facility," on page 2-24 for more details.
- The catch and ignore signal lists differ for each version.
- Features supported only on Solaris 2.x:
  - Child process follow-fork
  - Process/Thread Inspector
  - Visual Data Inspector
  - Collector performance tuning tool
  - The `debugger/dbx` - <*pid*> (dash, space, *process_id* option, which allows you to start the Debugger or dbx with a *process_id* only.
  - Modify (watchpoint) events: `fault`, `modify`, and `sysin/sysout` events.

1.2 The Debugger and dbx

The Debugger is a sophisticated window-based tool that interfaces with dbx. dbx is an interactive, line-oriented, source-level, symbolic debugger. It lets you determine where a program crashed, view the values of variables and expressions, set breakpoints in the code, and run and trace a program. In addition, machine-level and other commands are available to help you debug code.

During program execution, dbx obtains detailed information about program behavior and supplies the Debugger with this information via a communications protocol.

The Debugger is a window-based interface to dbx. Debugging is easier because you can use the mouse to enter most commands from redefinable buttons on the graphical user interface. You can use any of the standard dbx commands in the command window.

The Debugger also offers a program editing facility to minimize the need to change tools.

1.3 How the Debugger Works

The Debugger relies on debugging information a compiler generates using the compiler option `-g` to inspect the state of the process. By default on Solaris 2.x, debugging information for each program module is stored in the module's `.o` file. On Solaris 2.x, the Debugger reads in the information for each module as it is needed. This facility is called the Auto-Read facility.

For Auto-Read to work on Solaris 2.x, you must preserve a module's `.o` files. You can disable Auto-Read by compiling with the `-xs` option. See Section 2.2, "Using the Auto-Read Facility," on page 2-24 for details.

With Solaris 1.x, debugging information is kept in the `a.out` file and is pre-scanned to determine which sections belong to each module. The Auto-Read facility then only processes the sections needed as modules are entered or requested.

Quick Start

While reading this and the remaining parts of the overview chapter, you may want to start the Debugger and explore the tool as you read.

To start the Debugger:

- **Type `debugger` at a command shell prompt or double-click the Debugger icon in the SPARCworks Manager.**
  (See Chapter 3, "Starting the Debugger," for more information.)

The user interface combines the advantages and conveniences of both a window-based, select-operate interface and a text command-based interface. dbx, the familiar Sun debugger, provides the underlying debugging engine; dbx commands remain available as *Debugger commands* entered in the

integrated *command pane*. The Debugger command language is based on the Korn Shell[1] (ksh-88). The GUI aspect of the interface is a redesigned and extended OPEN LOOK version of dbxtool.

Note – In OpenWindows, you can specify the mouse pointer focus as "follow pointer" or "click to focus". You make the selection in the OpenWindows properties sheet, Miscellaneous Category, and Set Active Window field. Selections are "Move Pointer" and "Click Mouse". Choose Click Mouse to avoid the possibility of typing into a new pop-up window by mistake. If you choose Move Pointer, wait for all pop-up windows to display before typing.

1.3.1 Loading a Program or Attaching to a Process

The Debugger allows you to place a program under its control by:

- Loading a program into the Debugger
- Attaching the Debugger to a running program (after debugging the process, you detach the Debugger from it without quitting)
- Debugging the core file of a crashed program ("Postmortem debugging")

1.3.2 Main Features of the Debugger

The Debugger provides an extensive range of event management, process control, and data inspection features. You can:

- *Set breakpoints* at lines or in functions; *trace* program execution line by line across a whole program or within a function; *set watchpoints* to stop or trace a program if a specified value or expression changes or meets some other condition.
- *Set multiple breakpoints or tracepoints in C++* code: in all member functions of the same name across a class; in all members of a specified class; in all regular (that is, nonmember) functions of the same name (overloaded functions).

---

1. The Korn Shell Command and Programming Language, Morris I. Bolsky and David G. Korn, Prentice Hall, 1989

- *Single-step* through program code one line at time at either the source or machine-language level; step “over” or “into” function calls; step “up” and “out” of a function call arriving at the line of the calling function line (but after the call).
- *Run, stop, and continue* execution (at the next line or at some other line); *save and then replay* all or part of a debugging run.

(Solaris 2.x)
- Use the *performance-tuning data collection facility* for collecting data for use with the Analyzer tool.
- Use *Run Time Checking* (RTC) to automatically detect run time errors in an application. RTC detects memory access errors and memory leaks.
- *Look up declarations* of identifiers and definitions of types, classes and templates.
- *Spot check the value of variables or expressions* whenever the program is stopped; *monitor variables or expressions for chang*es over time; examine the call stack; move up and down the call stack, and call functions in the program.

(Solaris 2.x)
- Use *Data Inspector* to graphically examine program variables including complex structures and arrays, and monitor values during program execution.

(Solaris 2.x)
- Use *Process/Thread Inspecto*r to display lists of threads.
- Use *Fix and Continue* to fix (change) a file, recompile it with the same options, install the new code in the program, and continue.
- Support *Virtual Functions* for C++.
- Support *C++ exceptions*.
- Debug with C++ Templates.
- Use *Function Overloading* (C++) for argument resolution.
- Use *Default arguments* (C++).
- Use an embedded Korn shell for *Programmability*.
- Use the *Program Input/Output Window* to provide an I/O command interface for applications which does not interfere with the dbx interaction in the command window.

(Solaris 2.x)
- *Follow program*s as they `fork` (2) and `exec` (2).

1.3.3 Interaction of dbx with the Dynamic Linker

On Solaris 2.x, the Debugger provides full debugging support for programs that use dynamically-linked, shared libraries, provided that these libraries were compiled using the `-g` option.

Definitions

The dynamic linker, also known as "rtld", "RunTime ld", or "ld.so", arranges to bring shared objects (load objects) into an executing application. There are two primary areas where rtld is active:

1. **Program startup**
   At program startup, rtld runs first and dynamically loads all shared objects specified at link time. (You can use `ldd(1)` to find out what shared objects a program will load.) These are "preloaded" shared objects and commonly include `libc.so`, `libC.so`, `libX.so`, and so on.

2. **Application Requests**
   The application uses function calls `dlopen(3)` and `dlclose(3)` to dynamically load and unload shared objects, or plain executables. dbx uses the term "load objects" to refer to a shared object (`.so`) or plain executable (`a.out`).

The dynamic linker maintains a list of all loaded objects in a list called a "link map". The link map is maintained in user memory, and is indirectly accessed via the symbol name `_DYNAMIC`.

dbx traverses the link map to see:

- Which objects were loaded
- What their corresponding binaries are
- At what base address they were loaded

Corruption of these data structures can at times confuse dbx.

Debugging Support for Shared Objects

dbx can debug shared objects, both preloaded and those opened with `dlopen()`. Some restrictions and limitations are described in the following sections.

Startup Sequence

To put breakpoints in preloaded shared objects, the address of the routines has to be known to the Debugger. For the Debugger to know the address of the routines, it must know the shared object base address. Doing something as simple as

```
stop in printf
run
```

requires special consideration by the Debugger. Whenever you load a new program (either by starting dbx from the shell, the Debugger from a menu, or by using the `debug` command), dbx automatically executes the program up to the point where rtld has completed construction of the link map. dbx then reads the link map and stores the base addresses. After that, the process is killed and you see the prompt. These dbx tasks are completed silently.

At this point, the symbol table for `libc.so` is available as well as its base load address. Therefore, the address of `printf` is also known.

The activity of dbx *waiting* for rtld to construct the link map and accessing the head of the link map is known as the "rtld handshake". The event (see Chapter 6, "Event Management") `syncrtld` occurs when rtld is done with the link map and the Debugger has read all of the symbol tables.

With this scheme, the Debugger depends on the fact that when the program is Run, the shared libraries are loaded at the same base address. The assumption that shared libraries are loaded at the same base address is seldom violated; usually only if you change `LD_LIBRARY_PATH` between loading of the program and running it. In such cases, dbx takes note of the new address and prints a message. However, breakpoints in the moved shared object may be incorrect.

Startup Sequence and `.init` *Sections*

A `.init` section is a piece of code belonging to a shared object that is executed when the shared object is loaded. For example, the `.init` section is used by the C++ runtime system to call all static initializers.

The dynamic linker first maps in all the shared objects, putting them on the link map. Then, the dynamic linker traverses the link map and executes the `.init` section for each shared object.

Under Solaris 2.2 (and prior versions), the `syncrtld` event occurs after the `.init` sections have been run. You should note the following precautions:

- When you start debugging a new program, even though you have not invoked Run, the hidden handshake causes the `.init` sections to get executed.
- Because a shared objects symbol table is not available until after `syncrtld`, you cannot put breakpoints in the `.init` section.

  But, if a fatal error occurs in the `.init` section before the rtld handshake is complete, the process is not killed and you can still get a stack trace. If the program crashed in an `.init` section while running outside of the Debugger and generates a coredump, you can successfully do a postmortem analysis.

(After Solaris 2.2, the `syncrtld` point is between the link map construction and the execution of the `.init` sections; the precautions are not necessary.)

`dlopen()` *and* `dlclose()`

dbx automatically detects that a `dlopen` or a `dlclose` has occurred and loads the symbol table of the loaded object. You can put breakpoints in and debug the loaded object like any part of your program.

When a shared object is unloaded, the symbol table is discarded and the breakpoints are marked as "`(defunct)`" when you request `status`. Unfortunately, there is no way to automatically re-enable the breakpoints if the object is opened again on a consecutive run.

Two events, `dlopen` and `dlclose` (see Chapter 6, "Event Management", Section 6.4, "Event Specifications," on page 6-84), can be used with the `when` command, and some shell programming, to help ease the burden of managing breakpoints in `dlopen` type shared objects.

`fix` *and* `continue`

Using `fix` and `continue` (see Chapter 20, "fix and continue") with shared objects requires a change in how they are opened in order for fix and continue to work correctly. Use mode '`RTLD_NOW|RTLD_GLOBAL`' or '`RTLD_LAZY|RTLD_GLOBAL`'. This procedure works only for Solaris 2.3 or newer.

Procedure Linkage Tables (PLT)

PLTs are structures used by the rtld to facilitate calls across shared object boundaries. For instance, the call to `printf` goes via this indirect table. The details of how this is done can be found in the generic and processor specific SVR4 ABI reference manuals.

For the Debugger to handle `step` and `next` commands across PLTs, it has to keep track of the PLT table of each load object. The table information is acquired at the same time as the rtld handshake.

1.3.4 Debugging Optimized Code

You can debug optimized code in the Debugger; that is, code compiled with both the `-O` option and the `-g` option. However, a number of restrictions apply. See Section 2.3, "Debugging Optimized Code," on page 2-26.

1.3.5 History Facility

The Debugger records each command you enter. You can display the history of a session by typing `history` in the command pane. You can also issue a command from the list by number, using substitution commands that follow the csh (for backward compatibility) `history` conventions (`!!`, `!n`, `!chars`). The history is also used by the `replay` functionality.

1.4 Graphical Overview

The Debugger base window consists of a source display, a set of menu buttons, three rows of information fields, two rows of default button commands, a command pane, and a message/status area.

1

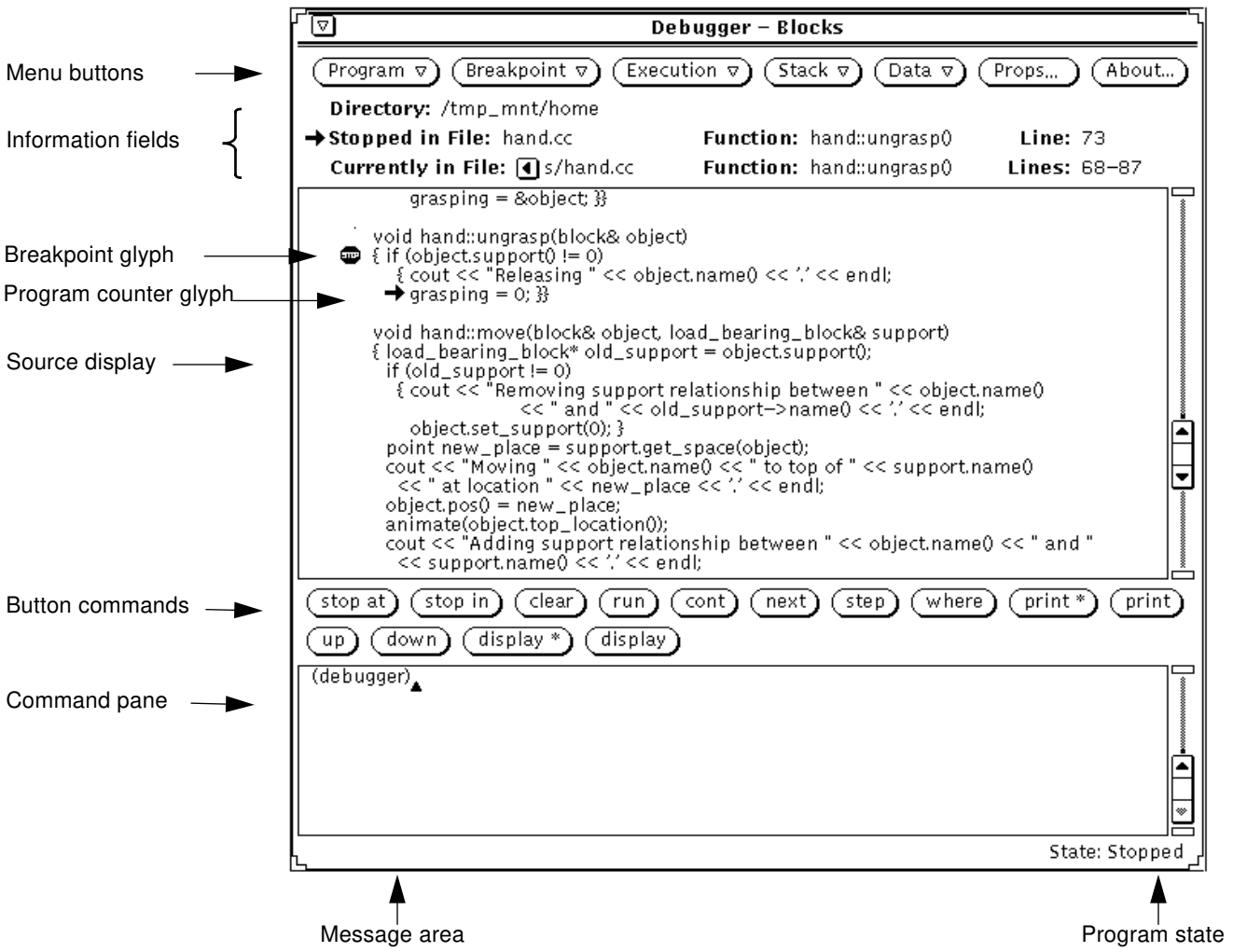
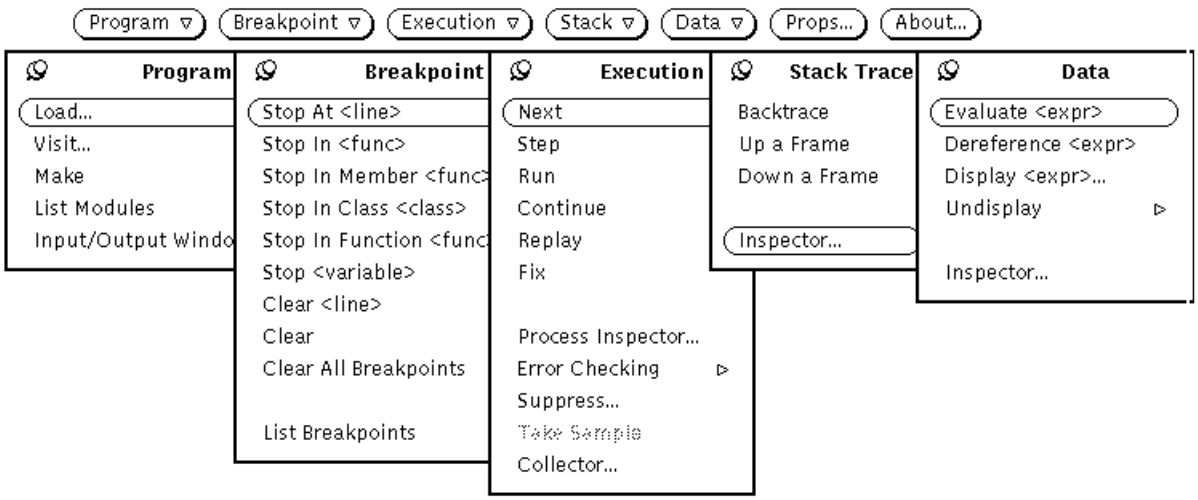


Figure 1-1 Debugger Base Window

1.4.1 Menu Items

The menu buttons beneath the Debugger title bar contain the most important and commonly-used debugging commands, as well as an assortment of debugging utility commands and Property window controls. The Props button is described in Section 1.4.7, “Debugger Properties Window Panes,” on page 1-18. The About button displays a window that briefly describes the Debugger tool. The circled item in each menu is the default item; you do not have to open the menu to choose the default item, just click on the menu button:



1.4.2 Information Fields

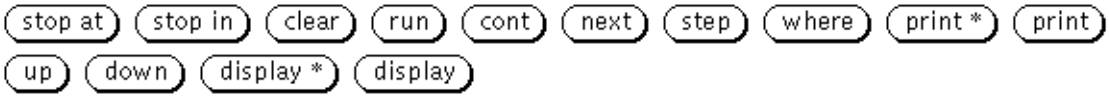
The information fields tell you where the program is stopped, what code is displayed in the source display, and the name of the current function. (See Chapter 4, “Viewing and Visiting Code,” for a detailed description of how to read the information presented in the base window.)

1.4.3 Source Display

The source display displays either the code where the program is stopped or code you are visiting away from the stop location. By default, the source display is a read-only pane. You can convert the source display to a text editor pane by choosing Enable Edit from the source display floating menu.

1.4.4 Button Commands

Button commands appear between the source display and the command pane. The Debugger opens with these preset button commands.



The preset button commands have the same function as their menu item or text command counterparts of the same name.

Using the Debugger `button` command and its modifiers, you can change any of these button commands, deleting, adding or rearranging them as you like. If your set of buttons cannot fit on these two rows, the Debugger creates a new row beneath the second one.

1.4.5 Command Pane

The command pane is at the bottom of the Debugger main window. The command pane serves as a subwindow for entering Debugger text commands and as an output device for displaying the results of various debugging commands. For example, if you choose Evaluate from the Data menu, the Debugger prints the value of the selected variable in the command pane.

Note that the Debugger supports a number of more sophisticated debugging commands or variants of simpler commands as *text commands only—commands which you enter in the command pane*. The Debugger command pane does not support the full ksh command line editing functionality. See Appendix A, "Debugger Commands," for a reference to the command pane commands.

In most cases, working with Debugger text commands in the command pane is the same as using dbx in a Command shell. However, there are a few differences. These differences reflect how the Debugger takes advantage of the source display for displaying code, rather than printing lines of code in the command pane. For example, when you issue a Debugger line-command like `list` *function_name*, the Debugger displays the code in the source display, not in the command pane.

The *Currently in Function* information field keeps track of the *current function*, that is the function that holds the Debugger focus for the targeting variables and visiting functions using Debugger line commands. See Section 4.4, "Visiting Functions from the Command Pane," on page 4-51 for details.

Shell Commands for the Command Pane

The Debugger supports a number of primitive commands for debugging. If `set -o path` is used, path searching is enabled and common UNIX commands are available. With or without `set -o path`, certain common commands not used for debugging are also built-in. These commands might have slightly different semantics.

`cd`	Change working directory.
`pwd`	Print working directory.
`use`	Set path for the Debugger to search for source files (debugging specific).
`search`	Search for a pattern matching *strings* in the current file (debugging specific).
`bsearch`	Search backwards for a pattern matching *strings* in the current file (debugging specific).
`dalias`	Make an alias for a debugging command.
`history`	Print a list of the most recent debugging commands (default 15), type `history` *n* to change default to *n.*
`kill`	Kill the process loaded in the Debugger, but not the Debugger.
`quit`	Terminate the session; detach the process if it was attached.
`shell_cmd`	Issue a ksh command.
`!!`	Execute previous command.
`!`*num*	Execute command number *num.*

`!-`*num*	Execute the command *num* commands before the current command.
`!`*str*	Execute most recent command starting with *str.*
`:s/l/r/`	Substitute modifier: substitute `r` for `l`.
`:p`	Print modifier: print (but not execute) the command.
`^l^r^`	Quick substitution: substitute `r` for `l`.

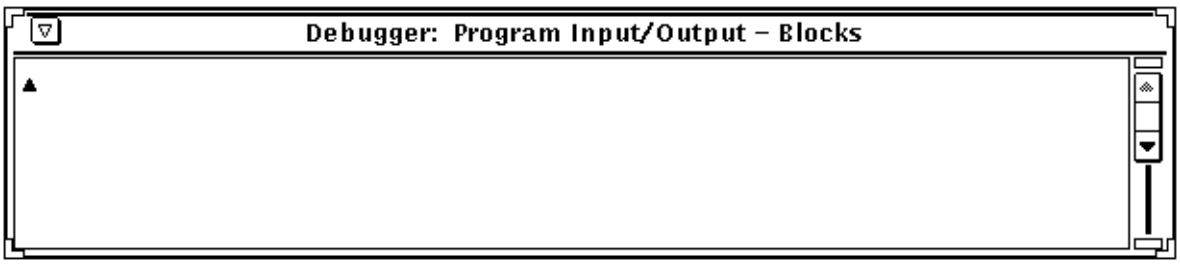
Note – If a program is running in the Debugger, Cntl-C (`^c`) interrupts the program, returning control to the Debugger.

The Debugger ignores a Cntl-D (`^d`) in the command pane. Use `quit` to end the Debugging session.

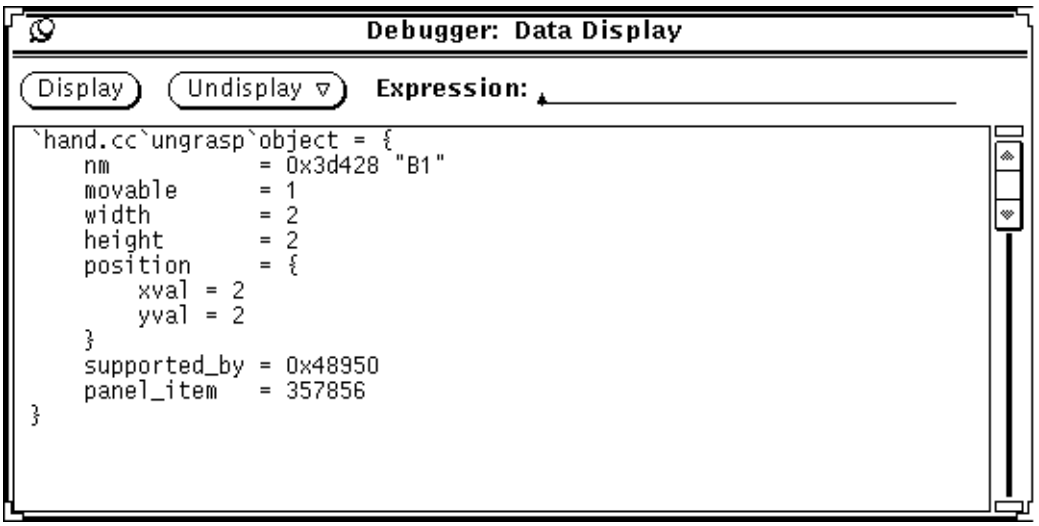
1.4.6 Program Input/Output Window (PIO)

The PIO window is a separate window that pops up initially at the bottom of the Debugger main window. The PIO serves as a display window for all input/output activities of the program being debugged. This window allows for separation of the usual debugging I/O from the program I/O.

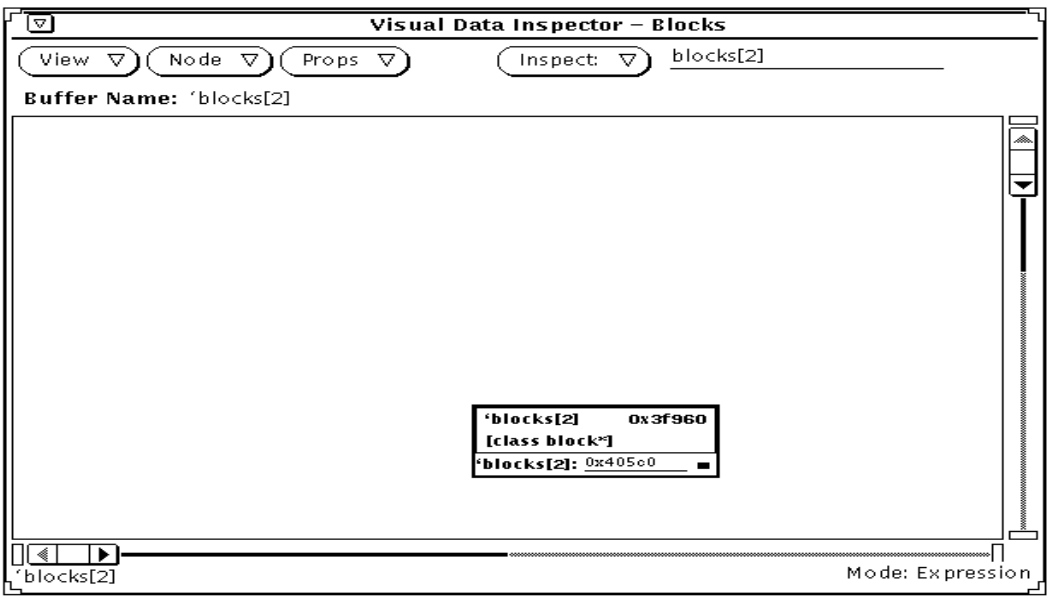
To move the PIO window to the front of the workspace, click on the Input/Output Window item on the Program menu.



In addition to the base window, the Debugger uses windows to monitor expressions and variables: the Data Display window and the Visual Data Inspector window (Solaris 2.x). Here is the Data Display window:

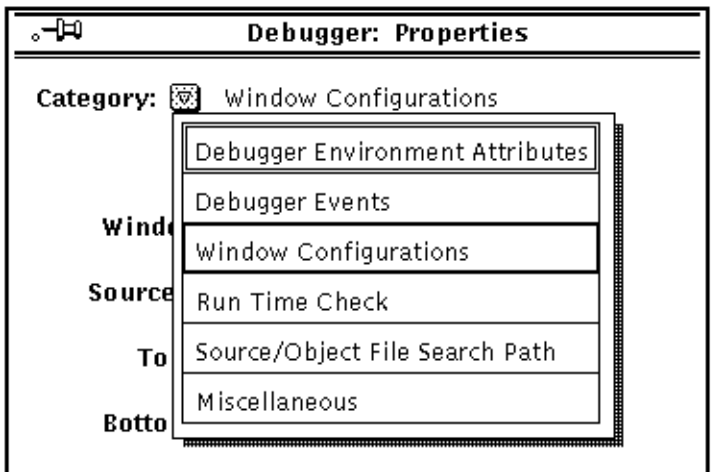


Here is the Visual Data Inspector window (Solaris 2.x).



1.4.7 Debugger Properties Window Panes

The Debugger Props button displays the Properties window. See Chapter 13, “Customizing the Debugger“, for details. Use the Category menu to move from one property sheet to another:

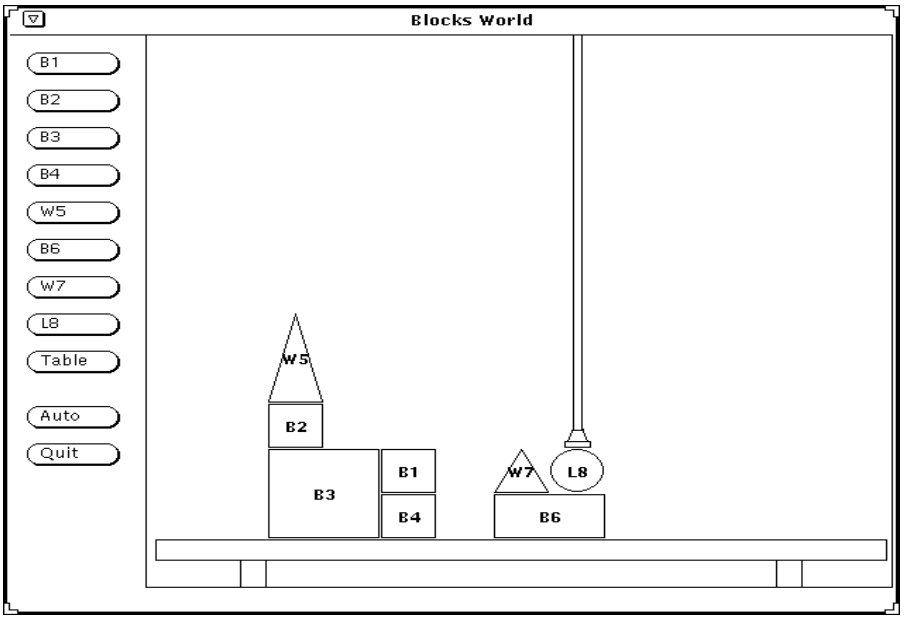


1.4.8 The Debugger Initialization Files

The Debugger supports an initialization file, `.dbxrc` with new syntax (Korn shell, see Section 23.4, “The Debugger Startup Mode” for more details) or `.dbxinit` with the old syntax (for backwards compatibility). In `.dbxrc` you can store debugging environment commands, commands to create customized button commands, source and object file search path settings, user-defined aliases for debugging commands, and user-defined Korn Shell functions for debugging. Also, `.debugger-init` controls the GUI resources. See Chapter 13, “Customizing the Debugger“, for details on how to use this file.

1.5 Example Application Program

This manual uses the Blocks[1] application in examples. Blocks is a C++ implementation of a Lisp application called Blocks World. The Blocks application defines several types of blocks (bricks, wedges, a ball, and a table) as subclasses of the basic class block. The Blocks application is used to move blocks on top of other blocks. The Blocks application installs automatically when you install the Debugger.



To use the Blocks application in the examples of this manual, you need to compile the application as specified in Section 2.1, “Compiling with the -g Option,” on page 2-23

1. The Blocks application used in this manual is derived from “Blocks World CLOS demo” from Chapter 21 of “Lisp”, third edition, by Patrick Henry Winston and Berthold K. P. Horn, Copyright (c) 1989, 1984, 1981 by Addison-Wesley Publishing Company, Inc. and San Marco Associates.

Part 2 — Basic Debugging

Preparing Programs 2

To use the Debugger effectively, a program must have been compiled with the `-g` or `-g0` option. The `-g` option instructs the compiler to generate Debugger information during compilation. The `-g0` (zero) option is for C++ support.

This chapter is organized into the following sections.

Compiling with the -g Option	*page 2-23*
Using the Auto-Read Facility	*page 2-24*
Debugging Optimized Code	*page 2-26*

2.1 Compiling with the `-g` Option

To compile a program for full Debugger support of the resulting code

♦ **Compile the source code with the `-g` option.**

2.1.1 Limited Support for Code Compiled Without `-g` Option

While most debugging support requires that a program be compiled with `-g`, the Debugger still provides the following level of support for code compiled without `-g`:

- Backtrace (Stack menu or the Debugger `where` command)
- Calling a function (but without parameter checking)

- Checking global variables

Note, however, that the Debugger cannot display source code in the source display unless the code was compiled with the `-g` option. This also applies to code that has had `strip -x` applied to it.

2.1.2 Shared Libraries Need `-g` for Full Debugger Support

For full debugger support, a shared library must also be compiled with the `-g` option. If you build a program with some shared library modules that were not compiled with `-g`, you can still debug the program. However, full debugging support is not possible because the information was not generated for those library modules.

2.1.3 C++ Support and the `-g` Option

In C++, `-g` turns on debugging and turns off inlining of functions. The `-g0` (zero) option turns on debugging and does not affect inlining of functions. You cannot debug inline functions with this option. The `-g0` option can significantly decrease link time and Debugger start-up time (depending on the use of inlined functions by the program).

2.1.4 Completely Stripped Programs on Solaris 2.x

The Debugger cannot debug programs that have been completely stripped on Solaris 2.x. These programs contain absolutely no information that can be used to debug your program.

2.2 Using the Auto-Read Facility

In general, you should compile the entire program you want to debug by using the `-g` option. Depending on how the program was compiled, the debugging information generated for each program and shared library module is stored in either the object code file (`.o` file) for each program and shared library module, and/or the program executable file.

2.2.1 Auto-Read is the Default for the Debugger

On Solaris 2.x, when you compile with the `-g` compiler option, debugging information for each module remains stored in its `.o` file. The Debugger then reads in debugging information for each module automatically, as it is needed, during a session. This read-on-demand facility is called *Auto-Read*.

On Solaris 1.x, all debug information is stored in the executable.

On Solaris 2.x, Auto-Read saves considerable time when loading a large program into the Debugger. Auto-Read depends on the continued presence of the program `.o` files in a location known to the Debugger.

Note – If you archive `.o` files into `.a` files, and then link using the archive libraries, you can then remove the associated `.o` files, but you must keep the `.a` files.

By default, the Debugger looks for files in the directory where they were when the program was compiled and the `.o` files in the location from which they were linked. If the files are not there, the Debugger uses the *Source/Object File Search Path* (the Debugger `use` command) to look for it. If you move source files or `.o` files, or use different mount points for the same files, be sure to set the Search Path accordingly. See Section 3.2, "Setting the Source/Object File Search Path," on page 3-34, for more information. See also: the `use` command and the `pathmap` command.

If No Object File Produced, Debugging Info Stored in Executable

For a compilation that does not produce `.o` files, the compiler stores all the debugging information in the executable and the Debugger does not use the Auto-read facility.

2.2.2 Disabling Auto-Read with the `-xs` Compiler Option

On Solaris 2.x, programs compiled with `-g` store debugging information for each module in the module's `.o` file. Auto-Read requires the continued presence of the program and shared library `.o` files.

In circumstances where it is not feasible to keep program `.o` files or shared library `.o` files for modules that you want to debug, compile the program using the compiler `-xs` option (use in addition to `-g`). (You can have some modules compiled with `-xs` and some without.) The `-xs` option instructs the compiler to have the linker place all of the debugging information in the program executable. Then when you load the program executable into the Debugger, all of the information loads at once. For a large program compiled with `-xs`, reading in the debugging information requires additional time; however, debugging is no longer dependent on the availability of the `.o` files.

2.2.3 Listing Modules That Have Been Read In

The `modules` command lists the modules for which debugging information is already read-in to the Debugger (optionally, it lists the `modules` not yet read-in.) For programs using the Auto-Read facility, use the `module` *module_name* command to specify modules to read in immediately. You can also have the Debugger read in the information for all of the modules, by using the `module -a` command. See Section 3.3, "Listing and Reading-in Program Modules," on page 3-37 for more information.

2.3 Debugging Optimized Code

The Debugger provides partial debugging support for optimized code. When analyzing optimized code, you can

- Stop execution at the start of any function (`stop in` *function* command)
- Display global variables and arguments
- Evaluate, display, or modify global or static variables

However, with optimized code, the Debugger cannot

- Single-step from one line to another (`next` or `step` command)
- Evaluate or display (monitor) local variables
- Assign values to local variables

To compile optimized code for use with the Debugger:

♦ **Compile the source code with both the `-O` (upper-case letter O) and the `-g` options.**
For example, to compile using C++:

```
% CC -O -g example_source.cc
```

See Chapter 20, “fix and continue,” for information on how to recompile selected modules with `-g` and how to obtain improved debugging ability using the `fix -g` option.

Starting the Debugger 3

This chapter describes how to start a debugging session. You can start the Debugger from the *Manager* or a *Command* or *Shell* tool window. This chapter describes start-up options and how to quit a session.

This chapter is organized into the following sections:

Starting a Debugging Session	*page 3-29*
Setting the Source/Object File Search Path	*page 3-34*
Listing and Reading-in Program Modules	*page 3-37*
Options to the Debugger Start-up Command	*page 3-38*

Note – These start-up instructions assume that the two component Debugger programs, `dbx` and `debugger`, are correctly installed on your system. See *Installing SunSoft Developer Products Software on Solaris* for details.

3.1 Starting a Debugging Session

Start the Debugger from

- The SPARCworks Manager (double-click on the Debugger tool icon)
- Command or Shell Tool window (type `debugger`)

3.1.1 Starting the Debugger from the Manager

To start the Manager,

♦ **Type** `sparcworks` **at the prompt in a Command or Shell tool.**

To start the Debugger from the Manager**:**

♦ **Double-click on the Debugger icon or drag and drop it onto the workspace.**

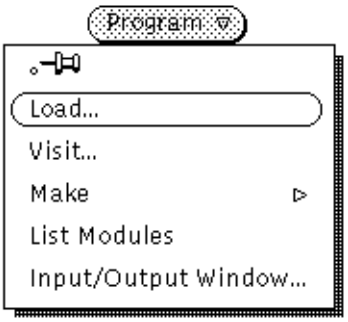
Program Loader

When starting the Debugger from the Manager, the Debugger displays the base window, which is empty because no program is yet loaded.

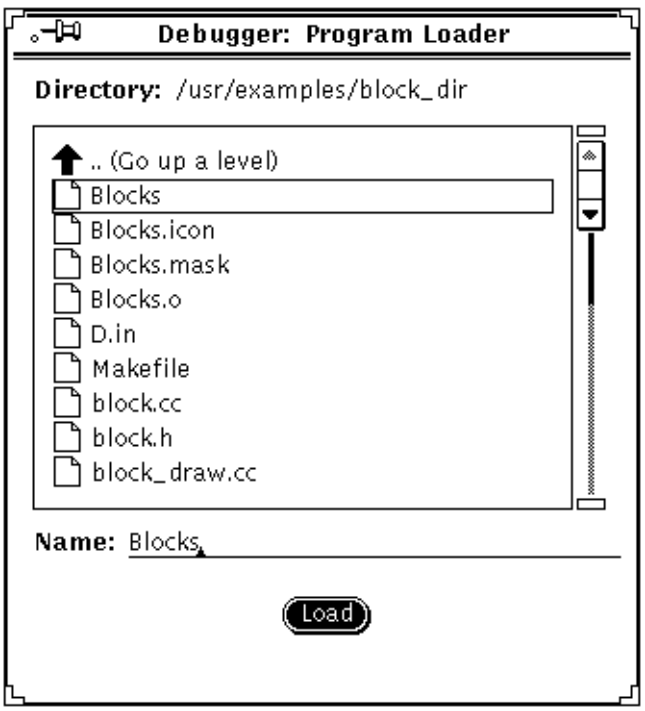
Use the Program Loader as a file chooser to select a program executable to load into the Debugger. Also, use the Program Loader to load a new program into the Debugger (replacing one already loaded).

To load a program for debugging:

1. **Choose Load from the Program menu to display the Program Loader, if it is not already displayed.**



2. **Click on a filename to select a program, then click Load (or double-click on the filename).**
   You can also type the name of the program in the Name field and then press Load.



Or, you can use the `debug` command in the command pane:

`(debugger)` **`debug`** [*program_name* [*corefile* | *process_id*]]

3.1.2 Starting the Debugger from a Command Tool

To start the Debugger from a Command tool (or Shell tool) window,

♦ **At the prompt, type**

`%` **`debugger`** [*–options*] [*program_name* [*corefile* | *process_id*]]

You can use the program Loader to choose a program executable file to debug.

For a list and description of the options to the start-up command, see Section 3.4, "Options to the Debugger Start-up Command," on page 3-38.

3.1.3 If a Core File Exists

If a file named `core` exists in the directory where you start the Debugger, it is not read in by default, you must explicitly request the core file with the `debug` command or on the Debugger command line. Use the Stack (or the Debugger `where`) command to see where the program was executing when it dumped core.

Note – To prevent you from overwriting a `core` file from a previous dump stored in the default location (the directory where you started the Debugger), the Debugger saves any core file generated by the Debugger in `/tmp`. If dbx or Debugger suffers an internal failure and dies, any existing core in `/tmp` is renamed.

3.1.4 Quitting a Debugging Session

A Debugger session runs from the time you start the Debugger until you quit the Debugger; you can debug any number of programs in succession during a Debugger session.

To quit a debugging session:

- **Choose `Quit` from the base window title bar menu or type `quit` in the command pane.**

If, when you start the Debugger, you attach it to a running process by using the *process_id* option, when you quit the Debugging session, the process survives and continues. That is, the Debugger performs an implicit `detach` before quitting the session. See Section 3.1.7, "Detaching a Process from the Debugger," on page 3-33, for more on `detach`.

3.1.5 Killing a Program Without Terminating the Session

The Debugger `kill` command terminates debugging of the current process as well as killing the process. However, `kill` preserves the Debugger session itself and the Debugger is ready to debug another program.

Killing a program is a good way of eliminating the remains of a program you were debugging without exiting the Debugger.

To kill a program that is executing in the Debugger:

♦ **Type, in the command pane,**

```
(debugger) kill
```

3.1.6 Attaching the Debugger to a Process

You can attach a running process to the Debugger by using the process *process_id* (*pid*) as an argument to the `debugger` command (or, if the Debugger is already in session, the `debug` command.)

Chapter 16, "Debugging Child Processes", describes how to attach the Debugger to a child process.

(Solaris 2.x) You can also attach to a process using its process ID number without knowing its name:

```
% debugger - pid                    // dash space process_id
```

Because the name remains unknown to the Debugger, you cannot pass arguments to the process in a `run` type command.

3.1.7 Detaching a Process from the Debugger

If you have attached a process to the Debugger, you can detach the process from the Debugger without killing it or the Debugging session.

To detach a process from the Debugger without killing the process:

♦ **Type in the command pane,**

```
(debugger) detach
```

3.1.8 Debugger Start-up Sequence

As the Debugger loads information, it prints a message in the command pane: `Reading symbolic information...` If the Debugger encounters problems loading the program, it prints messages reporting the problems.

Once the program is finished loading, the Debugger is in a ready state, visiting the "main" block of the program (For C or C++: `main()`; for FORTRAN `MAIN()`). Typically, you want to set a breakpoint and then issue a `run` command.

3.1.9 Running dbx Alone from a Shell

You can also start `dbx`—the command-line version of the Debugger—in a shell, and use it without the window interface component. Note that to run dbx outside of Openwindows, dbx still requires access to two shared libraries that are located in the Openwindows directory hierarchy:

```
installation_specific_location.../usr/openwin/lib/libtt.so
installation_specific_location.../usr/openwin/lib/libX11.so
```

If Openwindows is already installed in the default location on your system, then dbx should already have access to these two libraries. If not, put the directory containing the two libraries shown above in your `LD_LIBRARY_PATH`.

To start dbx:

♦ **In a Command or Shell Tool, or at a tty terminal, type**

```
% dbx [-options] [program_name [corefile | process_id]]
```

(Solaris 2.x) You can also start the Debugger with a process ID number:

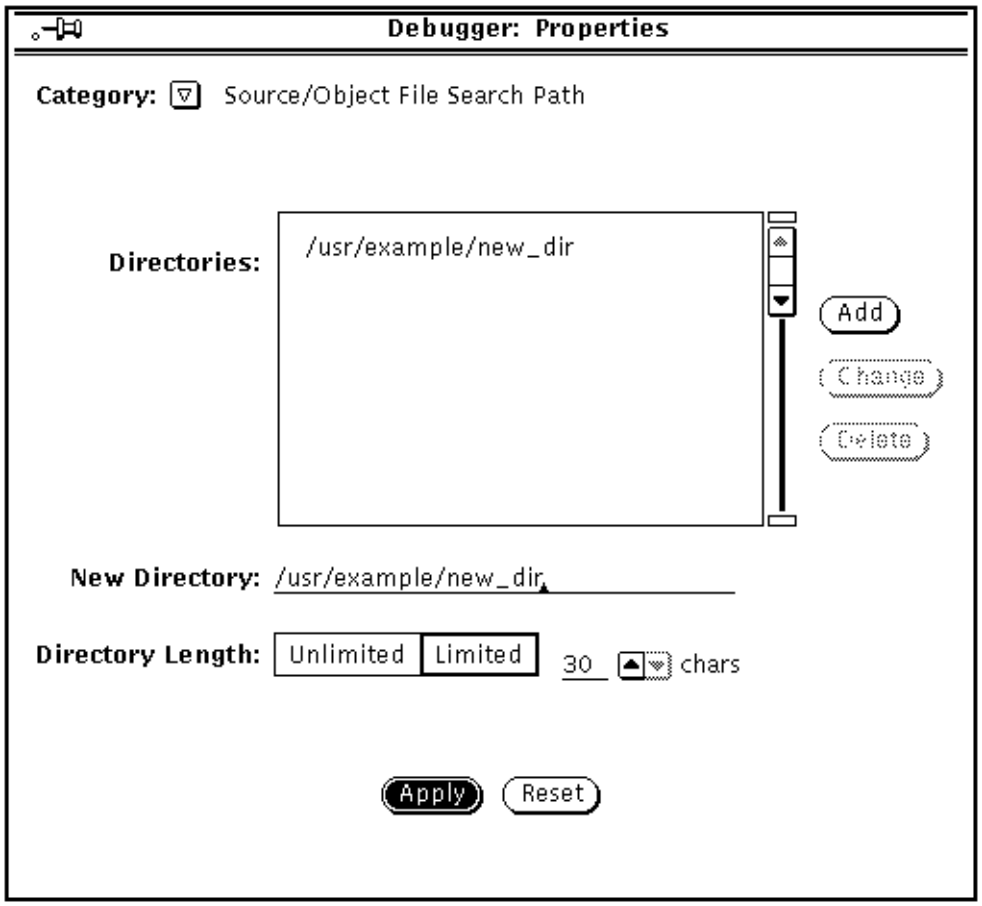
```
% dbx - pid                    // dash space process_id
```

3.2 Setting the Source/Object File Search Path

By default, the Debugger looks for the source files associated with the program being debugged in the directory in which they were when the program was compiled. If you move source/object files you must inform the Debugger of their new location. You can check the path, add to the list, edit a pathname, or delete a pathname:

To check the current File Search Path:

♦ **Display the Properties Window and choose Source/Object File Search Path from the Category menu.**



You must click the Apply button to execute the changes you make to the search path. Until you click Apply, the search path does not change. You can click Reset *before* clicking Apply to discard your most recent edits.

To Add a directory to the Source/Object File Search Path:

1. **Type in the name of the directory you want to add in the New Directory text field; click Add.**
   The directory name is displayed in the Directories display box.
2. **Click Apply to execute the change.**

To Change a path name already on the list:

1. **Select the path you want to edit by clicking on it in the Directories display box.**
   The selected entry is boxed.
2. **Click the Change button.**
   The command writes the pathname on the New Directory text field.
3. **Edit the pathname on the New Directory text field; then click Apply.**

To delete a path from the search path list:

- **Select a pathname in the Directories box; click Delete; then click Apply.**

Adjusting the Display of Pathnames in the Directories Box

By default, the Directories box truncates pathnames on the left if they are larger than 40 characters. You can adjust this setting up or down to see more or less of the pathnames.

Also, if you set Directories Length to Unlimited, the box expands to accommodate the longest pathname, up to the maximum number of characters before truncating the name.

3.2.1 Setting the Search Path in the Initialization File

If you want the Debugger to use a special source and object file search path each time you start a new session, add a line to your `.dbxrc` initialization file using the `use` command and pathname.

3.2.2 Setting the Search Path with `pathmap`

The `pathmap` command creates a mapping from one pathname to another. The mapping is applied to source paths, object file paths, and the current working directory (if "`-c`" is used).

The pathmap command is useful for dealing with automounted and explicit NFS mounted file systems with different base paths on differing hosts. Use "`-c`" when you try to correct problems due to the automounter because current working directories are inaccurate on automounted file systems.

The mapping of `/tmp_mnt` to `/` exists by default.

To establish a new mapping from <`from`> to <`to`>, where <`from`> and <`to`> are file path prefixes, type:

```
(debugger) pathmap [ -c ] <from> <to>
```

If "`-c`" is used, the mapping is applied to the current working directory as well.

To list all existing path mappings:

```
(debugger) pathmap
```

To delete the given mapping:

```
(debugger) pathmap -d <from>
```

3.3 Listing and Reading-in Program Modules

By default on Solaris 2.x, the compilers leave debugging information for each module compiled using the `-g` option, dispersed in each module's `.o` file. The Debugger reads in the debugging information as it is needed. (For more on this topic, see "Using the Auto-Read Facility" on page 2-24.)

Listing Modules

The Debugger `modules` command and its options help you to keep track of program modules during the course of a debugging session. The List Modules item on the Program menu is equivalent to the first of the Debugger `modules` commands listed here: `modules -read`.

To list the names of modules containing debugging information that have already been read into the Debugger:

```
(debugger) modules -read
```

To list names of all program modules (with or without debugging info):

```
(debugger) modules
```

To list all program modules with debugging info:

```
(debugger) modules -debug
```

Reading in Debugging Information

Use the `module` command to read in debugging information for one or all modules.

To print the name of the current module:

```
(debugger) module
```

To read in debugging information for a module *name*:

```
(debugger) module [-f] [-q]  <name>
```

To read in debugging information for all modules:

```
(debugger) module  [-f] [-q] -a
```

where,

`-f` forces reading of debugging information, even if the file is newer than the executable (use with caution).

`-q` is quiet mode.

3.4 Options to the Debugger Start-up Command

This section describes the options to `debugger`, the Debugger start-up command. Enter options after the command and before the name of a program, corefile, or process_ID you may want to load at start-up.

```
% debugger [-options] [program_name [corefile | process_ID]]
```

The options are organized in the table into the following categories:

- Help option
- Remote display option
- Source display buffer option
- Window applications option
- dbx options

Note – These start-up options are different from the Debugger environment attributes and Debugger window configuration options. You set environment and window controls from a Property window or by using `dbxenv` or `dbx setenv` commands. See Chapter 13, “Customizing the Debugger“, for more information.

Table 3-1 Options to `debugger`, the Start-up Command

Help Option	Description
`-help`	Display a brief summary of these debugging options.

Remote Display Option	Description
`-display` *hostname*`:0.0`	Display the Debugger on a remote display; *hostname* is the name of the remote machine.

Source Display Buffer Option	Description
`-M` *bytes*	Set a limit on the size of the file buffer associated with code displayed in the source display. The default is no limit, which means that the Debugger uses as much disk space as it needs to display in the source display on a cumulative basis (like a Command tool). For a system with a small amount of disk space, this no limit default could use all available disk space. For long debugging sessions on systems with a small amount of disk space, set *bytes* to 20000.

Window Applications Option	Description
`-Wfsdb`	Allows debugging of an OpenWindows program on the same server as the one on which it displays. Active grabs are disabled when you press a key, move the pointer, or perform a server action. Passive grabs are disabled when you press a mouse button. Note: you must set `-Wfsdb` as an argument to the Run item (or debugging `run` command) for OpenWindows applications. Without this option, the program may hang.

dbx Options	Description
`-c` *"dbx_cmd; ...;"*	Execute *dbx_commands* after initialization. Use quotation marks around debugging commands; for example, `% debugger -c "stop in main" demo`
`-e`	Echo all input commands on standard out.
`-I` *dir*	Add *dir* to the list of directories searched when looking for a source file. Normally, the Debugger looks for source files in the directory where the source file was compiled. If it cannot find the source files in that directory, it complains. If the source files are not in the compile directory, you must tell `dbx` where to find them, using either this option or setting the directory search path with the `use` command.
`-kbd`	Debug a program that sets the keyboard into up-down translation mode. This flag is necessary if a program uses up-down decoding.
`-r`	Load and then run the executable file immediately (instead of loading and waiting for a `run` command). Arguments to the program being debugged follow the object filename (redirection is handled properly). If the program terminates successfully, the Debugger exits. Otherwise, the Debugger reports the reason for termination and waits for your response. When `-r` is specified and standard input is not a terminal, the Debugger reads from `/dev/tty`.

	When using `-r`, any handlers for events dealing with the `exit` (2) system call might interfere with proper operation of the feature.
`-q`	Silence the echoing of the "`Reading symbol table for …`" message and the "`Attached to…`" message.
`-s` *start-up*	Read initialization commands from the file named *start-up*, not from the `.dbxrc` file.
`-sr` *start-up*	Read and delete "`start-up`".
`-wskip` *n*	Cause the `where` command to skip the first *n* stack frames.
`-C`	Enable Run Time Checking.

Viewing and Visiting Code 4

This chapter describes how the Debugger reports what is happening during a debugging session. It also covers how to use commands to visit code or look up declarations for identifiers, types, and classes.

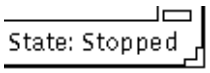
In general, each time the program stops, the code displayed in the source display is the code associated with the *stop location*. Also, each time the program stops, the Debugger resets the value of the *current function* to the function in which the program is stopped. When the program is stopped, you can "visit" functions and files elsewhere in the program.

This chapter is organized into the following sections:

Knowing the State of the Program	*page 4-44*
Reading the Information Fields	*page 4-44*
Visiting Code	*page 4-47*
Visiting Functions from the Command Pane	*page 4-51*
Locating Symbols: the whereis and which Commands	*page 4-55*
Looking Up Variables, Members, Types, and Classes	*page 4-57*

4

4.1 Knowing the State of the Program



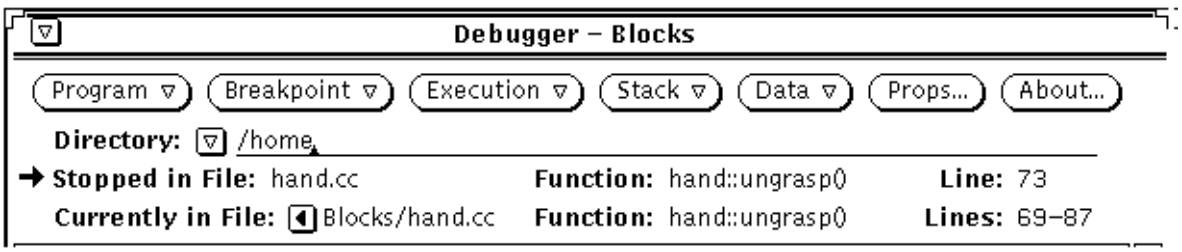
The Debugger reports the state of the program in the lower right message area of the base window.

The program may be in one of the following states:

- exiting
- killed
- initial
- ready
- resumed
- running
- stopped

4.2 Reading the Information Fields

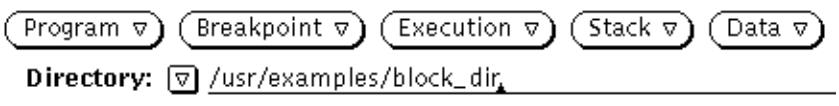
The Debugger displays three information fields beneath the menu buttons: Directory, Stopped in (File, Function, Line), and Currently in (File, Function, Lines).



4.2.1 Directory

The Directory information field lists the current directory of the Debugger. Initially, Directory is the Debugger start-up directory name.

The Directory field is initially read-only, but if you click on any character in the directory name, the field converts to an editable text field:



When you edit the directory name and then press Return, The Debugger changes to the new Directory.

Returning to a Directory

When you change the directory, the Directory menu records the history of the directories to which you have changed during the current session.

To change the location of the Debugger to a previous working directory,

♦ **Choose the directory item from the Directory menu.**

Note – The Debugger must know where the source and object code files associated with a program are located. The default directory for the object files is the one they were in when the program was last linked. The default directory for the source files is the one they were in when last compiled. If you move the source or object files, or copy them to a new location, you must either relink the program or enter the new location in the Source/Object File Search Path before debugging. (See Section 3.2, "Setting the Source/Object File Search Path," on page 3-34, and "Scope Resolution Search Path" on page 4-54.)

4.2.2 Stopped in File…Function…Line

The Stopped in information fields tells you where the program is stopped by filename, function name, and the line number. Also, when a program is stopped, a *solid black arrow* is displayed to the left of the Stopped in information fields; the same type of arrow is displayed in the source display to the left of the line where the program stopped.

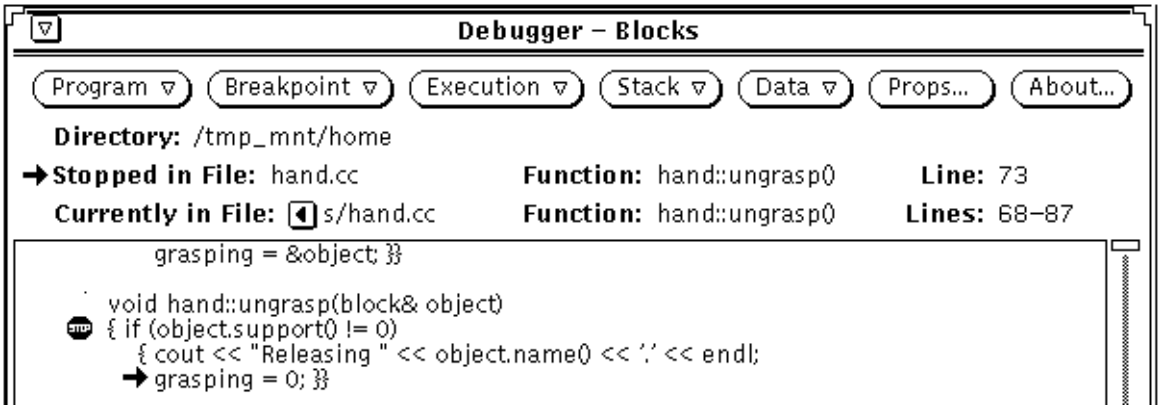


Figure 4-1 Stopped in File-Function-Line Information Field

4.2.3 Currently in File…Function…Lines

The Currently in fields report which code is shown in the source display. Currently in Function also reports the name of the *current function* (as shown in the next illustration).

Notice that in Figure 4-1, the values of Stopped in Function and Currently in Function are the same: every time the program stops, the Debugger changes the current function (the value of Currently in Function, and the value of the Debugger `func` command) to match Stopped in Function. However, when you *visit* a function, the two values diverge:

The *current function* is the function that holds the Debugger's "focus" or "attention" for the purposes of targeting debugging line commands.

Specifically, the current function determines the *scope resolution search order* that applies when you give a variable name to a Debugger command in the command pane. Knowing the current function is mainly important for issuing `Print`, `Print*`, `Stop in`, and `Display` line commands in the command pane. See "Visiting Functions from the Command Pane" on page 4-51, for details.

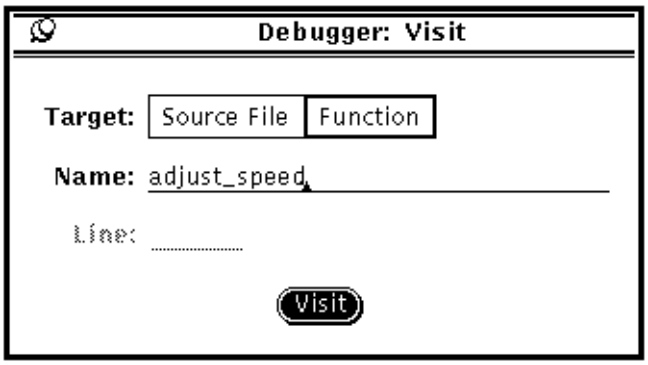
4.3 Visiting Code

When a program is stopped, you can visit code elsewhere in the program. The Currently in fields show what code you are visiting. You can visit any function or file that is part of the program.

4.3.1 Visiting a Function

To visit a function:

- **Choose Visit from the Program menu to display the Visit pop-up window; set Target to Function.**
  When visiting among a number of functions (or files), consider pinning open the Visit pop-up window.



- **Type in the name of the function and press Visit.**

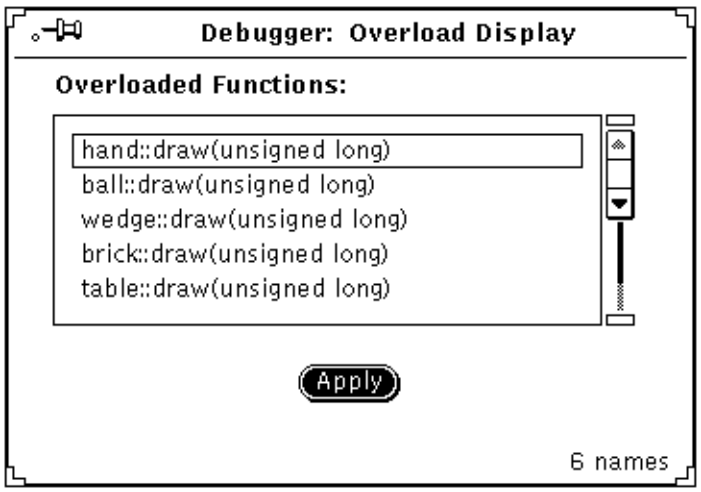
  You can also use a dbx command to visit a function:

  Type the command `func` followed by the function name. For example:

  ```
  (debugger) func adjust_speed
  ```

4.3.2 Selecting from a List of C++ Ambiguous Function Names

If you try to visit a C++ member function with an ambiguous name or an overloaded function name, the Overload Display pop-up window is displayed, showing a list of all functions with the overloaded name.



To specify a function from the list of ambiguous or overloaded function names:

♦ **Click the name of the function you want to visit and then click Apply.**

After visiting a function, Currently in Function reports the new current function. If the function is in a different file, Currently in File changes too. Lines reports which line numbers are visible in the source display. Note that no arrow is displayed in the source display next to a function you are visiting (unless it is part of the active call stack).

You can also type in the command pane instead of using the menu:

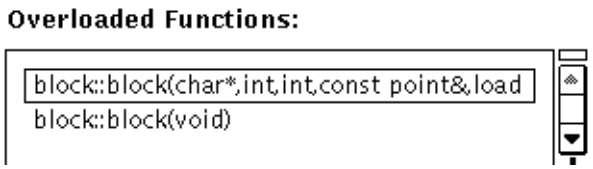
```
(debugger) func block::block
```

If the Search for a Symbol Name Finds Multiple Occurrences

If, at any place along the search path the search finds multiple occurrences of *symbol* at the same scope level, The Debugger prints a message in the command pane reporting the ambiguity:

```
(debugger) func main
(debugger) which block::block
Class block has more than one function member named
block.
```

At the same time, the Debugger pops up the Overload Display window listing the ambiguous symbols names:



In the context of the `which` command, choosing from the list of occurrences does not affect the state of the Debugger or the program. Whichever occurrence you choose, the Debugger merely echoes the name in the command pane.

Remember that the `which` command gives you a preview of what happens if you make *symbol* (in this example, `block`) the target of a command that must operate on *symbol* (for example, `a print` command). In the case of ambiguous names, the Overload Display window indicates that the Debugger does not yet know which occurrence of two or more names it would use. The Debugger lists the possibilities and waits for you to choose one.

4.3.3 *Redisplaying the Line Where the Program is Stopped*

When a program is stopped, the solid black arrow is displayed to the left of the Stopped in fields. The arrow remains there when you visit a function or file.

To redisplay the stop location:

♦ **Click on the solid black arrow to the left of the Stopped in fields.**

Note – Redisplaying the stop location by clicking on the arrow does *not* change the current function. The Debugger changes the current function to match the stop location only after the program executes lines of code and stops again.

4.3.4 Using `list` to Display a Function

As noted already, visiting a function changes the current function. To display a function *without* changing the current function, use the Debugger line command, `list`:

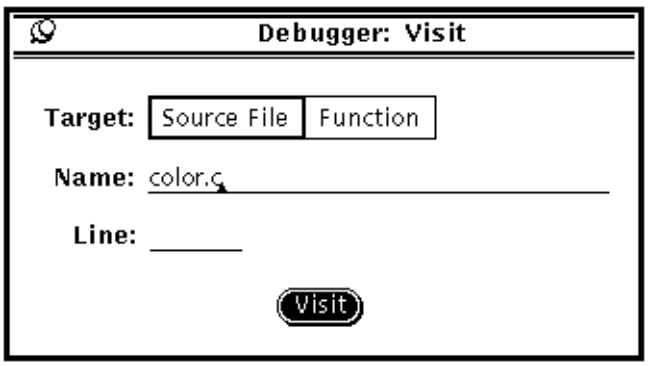
```
(debugger) list function_name
list function_name (args)
```

4.3.5 Visiting a File

You can visit any file the Debugger recognizes as part of the program (even if a module or file was not compiled with the `-g` option.) Visiting a File does *not* change the current function. So, if you visit a file, the Currently in File field reports the name of the file currently displayed in the source display, but the Currently in Function field still reports the function that holds the Debugger focus.

To visit a file:

- **Choose Visit from the Program menu to display the Visit pop-up window; set Target to File; type in the name of the file and click Visit.**



You can specify a line number in the source file for the Debugger to display. Otherwise, the Debugger displays the file from its first line.

4.3.6 Walking the Call Stack to Visit Code

Another way to visit code is to "walk the call stack," that is, to use the Stack commands or the Up and Down command buttons to view some function currently on the stack. (See Section 9.2, "The Stack Inspector," on page 9-122, for details on how to use the Stack Inspector to visit code.)

Walking the stack causes the current function to change each time you display a stack function. (The stop location is considered to be at the "bottom" of the stack; so to move away from it, you must use the Up command, that is, move toward the `main` or `begin` function.)

When you move away from the stop-location using Stack commands, a hollow arrow is displayed to the left of the *Currently in* fields. This hollow arrow reminds you that the function displayed is *on* the stack but *not at the top* of the stack:

See Chapter 9, "Examining the Call Stack", for more information.

4.4 Visiting Functions from the Command Pane

Using the `func` command in the command pane is equivalent to visiting a function using Program:Visit-Function. Also, the Debugger `file` command is equivalent to using Program:Visit-Source File. However, when using `func` or `file`, you may need to use *scope resolution operators* to qualify the names of the functions or files that you give as targets for these line commands.

For instance, in a C++ program, you may want to qualify an overloaded function name. If you do not qualify it, the Debugger pops up the Overload Display window for you to choose which function you mean to visit. (See "Selecting from a List of C++ Ambiguous Function Names" on page 4-48.) If you know the function class name, you can use it with the double colon scope resolution operator to qualify the name and avoid popping up the Overload Display window.

`(debugger)` **`func`** *class*`::`*function_name (args)*

Also, a program may use the same function name in two different files (or compilation modules). In this case, you must also qualify the function name to the Debugger so that it knows which function you mean to visit. To qualify a function name with respect to its filename, use the general purpose backquote (`) scope resolution operator:

```
(debugger) func `file_name`function_name
```

4.4.1 Qualifying Symbols with Scope Resolution Operators

The Debugger provides three scope resolution operators with which to qualify symbols: the backquote operator (`), the C++ double colon operator (::), and the block local operator (:<*lineno*>). You use them separately, or in some cases, together.

Note – In addition to qualifying file and function names when visiting code, symbol name qualifying is also necessary for printing and displaying out-of-scope variables and expressions, and for displaying type and class declarations (`whatis` command). The symbol qualifying rules are the same in all cases; this section covers the rules for all types of symbol name qualifying.

Backquote Scope Resolution Operator

The syntax for the backquote character(`) are:

`*global* // search for variable of global scope

`*source_file_name*`*function_name*`*variable_name*

`*source_lib_name*`*function_name:line_number*`*name* // search for a block\ local variable

`*source_file_name*`*name*// search for file static variable or function name
``*name* // search for file statics in all files

Here, *source_file_name* is a name of a compilation unit. The Debugger accepts two forms of filenames:

source_file_name.*language_specific_suffix*	`animate.c`
source_file_name.o	`animate.o`

Examples; where `animate.c` is a source file, `change_glyph()` a function, `item` a nonunique variable name, and `color` a file static variable name:

```
(debugger) print change_glyph`item
(debugger) print `animate.o`change_glyph`item
(debugger) print `animate.0`change_glyph:230`item
(debugger) display `animate.c`color
```

Nested Functions in Pascal

To qualify a symbol within a nested function in Pascal, extend the qualifier to include the name of the nested function. Here is the generic syntax:

function_name[`*function_name* ...]`*symbol*

For commands that target a Pascal nested function, the last name is a name of the function:

```
(debugger) stop in function_name[`function_name...]
`function_name
```

C++ Double Colon Scope Resolution Operator

Use the double colon operator (`::`) to qualify a C++ member function or top level function with

- An overloaded name (same name used with different argument types)
- An ambiguous name (same name used in different classes)

The syntax is:

class_name`::`*function_name*

```
(debugger) func hand::draw
```

4.4.2 *Linker Names*

The Debugger provides a special syntax for looking up symbols by their linker names (mangled names in the case of C++). You prefix the symbol name with a '#' character (and use the ksh escape character '\' before any '$' characters).

Examples:

```
(debugger) stop in #.mul
(debugger) whatis #\$FEcopyPc
(debugger) print `foo.c `#staticvar
```

4.4.3 Ambiguous or Overloaded Function Names in a Command Pane

If you issue a command for an overloaded or ambiguous target (as shown in the figure), the command pane reports the situation and the Debugger pops up the Overload Display window. (See Section 4.3.2, "Selecting from a List of C++ Ambiguous Function Names," on page 4-48.)

```
(debugger) whatis draw
More than one function or variable named `draw'.
>
```

- **Select from the list of member functions in the pop-up window and click Apply.**
  The command pane echoes a number, which corresponds to the number of the function you selected on the list, counting from the top of the list.

4.4.4 Scope Resolution Search Path

When you issue a debugging command with a *symbol* target name that requires the Debugger to search for *symbol*, the search order is as follows:

1. The Debugger first *searches within the scope of the current function.* If the program is stopped in a nested block, the Debugger searches within that block, then within the scope of all enclosing blocks declared by that function. If it does not find the symbol, the search continues along this path.
2. For Pascal only: the next immediately enclosing function.
3. For C++ only: class members of the current function's class.
4. The immediately enclosing "compilation unit": generally, the file containing the current function.
5. The global scope.
6. If none of the above searches are successful, the Debugger assumes that you are referencing a private variable, that is, a "file static" variable or function. The Debugger optionally searches for a file static symbol in every compilation unit depending on the value of the dbxenv "lookaside" setting.

The Debugger uses whichever occurrence of the symbol it first finds along this search path. If the Debugger cannot find a variable, it reports an error.

4.5 *Locating Symbols: the* `whereis` *and* `which` *Commands*

In a program, the same name may refer to different types of program entities and occur in many locations. The Debugger `whereis` command lists the fully qualified name—hence, the location—of all symbols of that name. The Debugger `which` command tells you which occurrence of a symbol the Debugger will use if you give that name as the target of a debugging command.

4.5.1 `whereis`*: Printing a List of Occurrences of a Symbol*

To print a list of all the occurrences of a specified symbol:

- **In the command pane, type** `whereis` *symbol*.
  symbol may be any user-defined identifier. For example:

```
(debugger) whereis table
in loadable object "Blocks"
class:  table
class:  table
class:  table
class:  table
class:  table
class:  table
function:       `table::table
in loadable object "/usr/lib/libc.so.1.6"
variable:       `hsearch.o`table
(debugger)
```

Notice in the previous example that the output includes the name of the loadable object(s) where the program defines *symbol*, as well as the kind of entity each object is: in the example, class, function, or variable.

Note – Because the Auto-load facility reads in information from the Debugger symbol table as it is needed, the `whereis` command knows only about occurrences of a symbol that are already loaded. As a debugging session gets longer, the list of occurrences may grow.

4.5.2 *Seeing a Preview of Which Symbol the Debugger Will Use*

The `which` command tells you which symbol with a given name it will use if you specify that name (without fully qualifying it) as the target of a debugging command.

To see which symbol the Debugger will use:

- **Type, in the command pane,** `which` *name.*
  For example:

```
(debugger) func
wedge::wedge
(debugger) which draw
`block_draw`wedge::draw
```

If a Symbol Name is Not in Scope

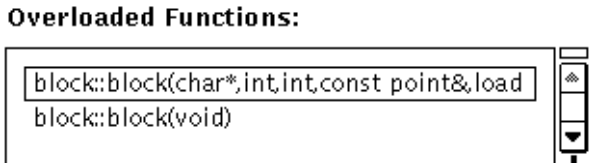
If a specified symbol name is not in a local scope, the `which` command searches for the first occurrence of the symbol along the *scope resolution search path*. (See "Scope Resolution Search Path" on page 4-54.) If `which` finds the name, it reports the fully qualified name in the command pane.

If the Search for a Symbol Name Finds Multiple Occurrences

If, at any place along the search path the search finds multiple occurrences of *symbol* at the same scope level, The Debugger prints a message in the command pane reporting the ambiguity:

```
(debugger) func main
(debugger) which block::block
Class block has more than one function member named
block.
```

At the same time, the Debugger pops up the Overload Display window listing the ambiguous symbols names:



In the context of the `which` command, choosing from the list of occurrences does not affect the state of the Debugger or the program. Whichever occurrence you choose, the Debugger merely echoes the name in the command pane.

Remember that the `which` command gives you a preview of what happens if you make *symbol* (in this example, `block`) the target of a command that must operate on *symbol* (for example, a `print` command). In the case of ambiguous names, the Overload Display window indicates that the Debugger does not yet know which occurrence of two or more names it will use. The Debugger lists the possibilities and waits for you to choose one.

4.6 Looking Up Variables, Members, Types, and Classes

The Debugger `whatis` command prints the declarations or definitions of identifiers, structs, types and C++ classes, or the type of an expression. The identifiers you can look up include variables, functions, fields, arrays, and enumeration constants. For example:

`whatis` [*-n*] [*-r*] non-type identifier
`whatis -t` [*-r*] type identifier
`whatis -e` [*-r*] expression

See Section 4.6.3, "Using whatis to See Inherited Members," on page 4-60 for a description of *-r*.

4.6.1 Displaying Declarations of Variables and Members

To print out the declaration of an identifier:

♦ **Type in the command pane**

```
(debugger) whatis identifier
```

Qualify the identifier name with file and function information as needed. Here are some examples.

`whatis` *a Member Function*

```
(debugger) whatis block::draw
void block::draw(unsigned long pw);

(debugger) whatis table::draw
void table::draw(unsigned long pw);

(debugger) whatis block::pos
class point *block::pos();
```

Notice that `table::pos` is inherited from block:

```
(debugger) whatis table::pos
class point *block::pos();
```

`whatis` *a Data Member*

```
(debugger) whatis block::movable
int movable;
```

`whatis` *a Variable or a Field*

On a variable, `whatis` tells you the variable's type:

```
(debugger) whatis the_table
class table *the_table;
```

On a field, `whatis` tells you the field's type:

```
(debugger) whatis the_table->draw
void table::draw(unsigned long pw);
```

`whatis` *on the* `this` *Pointer*

When you are stopped in a member function, you can lookup the `this` pointer. Notice that in this example, the output from the `whatis` shows that the compiler automatically allocated this variable to a register.

```
(debugger) stop in brick::draw
(debugger) cont
```

// expose the blocks window (if exposed, hide then expose) to force program to hit the breakpoint.

```
(debugger) where 1
brick::draw(this = 0x48870, pw = 374752), line 124 in
      "block_draw.cc"
(debugger) whatis this
register class brick *this;
```

Looking Up or Evaluating a Member of the `this` *Object*

Continuing with the `what is` *on the* `this` *Pointer* example: to see the type of a member of the `this` pointer (or to evaluate it):

```
(debugger) whatis this
register class brick *this;
(debugger) whatis this->movable
int movable;
(debugger) print this->movable
this->movable = 1
```

Or simply:

```
(debugger) whatis movable
int movable;
(debugger) print movable
movable = 1
```

4.6.2 Looking Up Definitions of Types and Classes

To print the declaration of a type or C++ class:

- **Type in the command pane**

```
(debugger) whatis -t type_or_class_name
```

Using `whatis` ***On a Class Name***

```
(debugger) whatis -t class block
class block {
     block::block();
     block::block(char *name, int w, int h, class point&pos,
          class load_bearing_block *blk);
     char *block::type();
     char *block::name();
     int block::is_movable();
     int block::w();
     int block::h();
     class point *block::pos();
// 15 members removed from this example
     char *nm;
     int movable;
     int width;
     int height;
     class point position;
     class load_bearing_block *supported_by;
 Panel_item panel_item;
};
```

4.6.3 *Using* `whatis` *to See Inherited Members*

The `whatis` command takes an option, `-r` (for recursive), that displays the declaration of a specified class together with the members it inherits from parent classes.

```
(debugger) whatis -t -r class_name
```

The output from a `whatis -r` query may be long, depending on the class hierarchy and the size of the classes. The output begins with the list of members inherited from the most ancestral class. Note the inserted comment lines separating the list of members into their respective parent classes.

Here are two examples, using the class table, a child class of the parent class load_bearing_block, which is, in turn, a child class of block. The first whatis command does not use the -r option, the second one does.

Using `whatis` *on a Class Name without the* `-r` *option*

Without `-r`, `whatis` reports the members declared in class `table`:

```
(debugger) whatis -t class table
class table : public load_bearing_block {
     table::table(char *name, int w, int h, class point
          &pos);
     char *table::type();
     void table::draw(unsigned long pw);
};
```

Using `whatis -r` *on a Child Class to see Members it Inherits*

```
(debugger) whatis -t -r class table
class table : public load_bearing_block {
     /* from base class table::load_bearing_block::block */
     block::block();
     block::block(char *name, int w, int h, class point &pos, class
           load_bearing_block *blk);
     char *block::type();
     char *block::name();
     int block::is_movable();
// 20 members removed from this example
     class point  position;
     class load_bearing_block *supported_by;
     Panel_item panel_item;
     /* from base class table::load_bearing_block */
     load_bearing_block::load_bearing_block();
     load_bearing_block::load_bearing_block(char *name, int w, int h,
           class point &pos, class load_bearing_block *blk);
// 10 members removed from this example
     class list *support_for;
     /* from class table */
     table::table(char *name, int w, int h, class point &pos);
     char *table::type();
     void table::draw(unsigned long pw);
};
```

Setting Breakpoints and Traces 5

This chapter describes how to set, clear, and list breakpoints and traces, and how to use watchpoints. The various forms of these commands are called, collectively, *event management* commands. Event management refers to the general capability of the Debugger to perform certain actions when certain events take place in the program being debugged.

The most common example of this association of a program event with a Debugger action is the simple breakpoint set at a line.

This chapter is organized into the following sections:

Setting Breakpoints	*page 5-64*
Setting Multiple Breaks in C++ Programs	*page 5-68*
Tracing Code	*page 5-70*
Listing and Clearing Event Management Commands	*page 5-72*
Setting Event Filters	*page 5-75*
Event Efficiency	*page 5-76*

5

5.1 Setting Breakpoints

There are three types of breakpoint action commands:

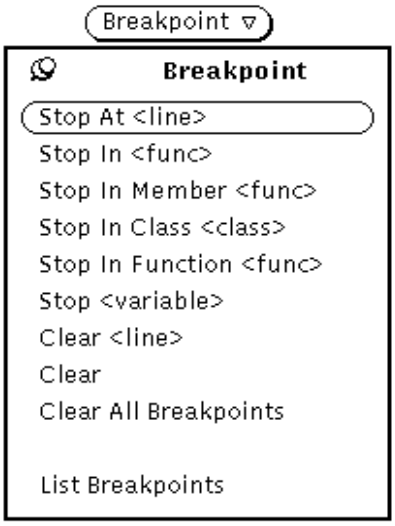
- `stop` **type breakpoints —** If the program arrives at a breakpoint created with a `stop` command, the program halts. The program cannot resume until you issue another debugging command.
- `when` **type breakpoints —** the program halts and the Debugger executes one or more debugging commands, then the program continues (unless one of the commands is `stop`).
- `trace` **type breakpoints —** the program halts and an event-specific `trace` information line is emitted, then the program continues.

Note – If the line selected in the source display or specified in a `stop` or `when` Debugger command is not an executable line of source code, the Debugger sets the breakpoint at the next line after the specified line that is executable.

The Breakpoint Menu

The Breakpoint menu contains items for setting location type breakpoints, including multiple breakpoints for C++ code, and for listing and clearing breakpoints.

To set conditional type breakpoints and `when` type breakpoints and traces, use the appropriate Debugger command in the command pane.



Stop At <line> — set breakpoint at selected line.

Stop In <func> — set breakpoint in selected procedure or function.

Stop In Member <func> — set breakpoint in all member functions with same name as selected member function across all classes.

Stop In Class <class> — set breakpoints in all member functions of <class>.

Stop In Function <func> — set breakpoints in all top level functions with same name (overloaded) as selected top level function.

Stop <variable> — set breakpoint at a change in the value of the specified <variable>.

Clear <line> — clear breakpoint at selected line.

Clear — clear breakpoint at current stop location (regardless of current selection).

Clear All Breakpoints — clear breakpoints at all locations.

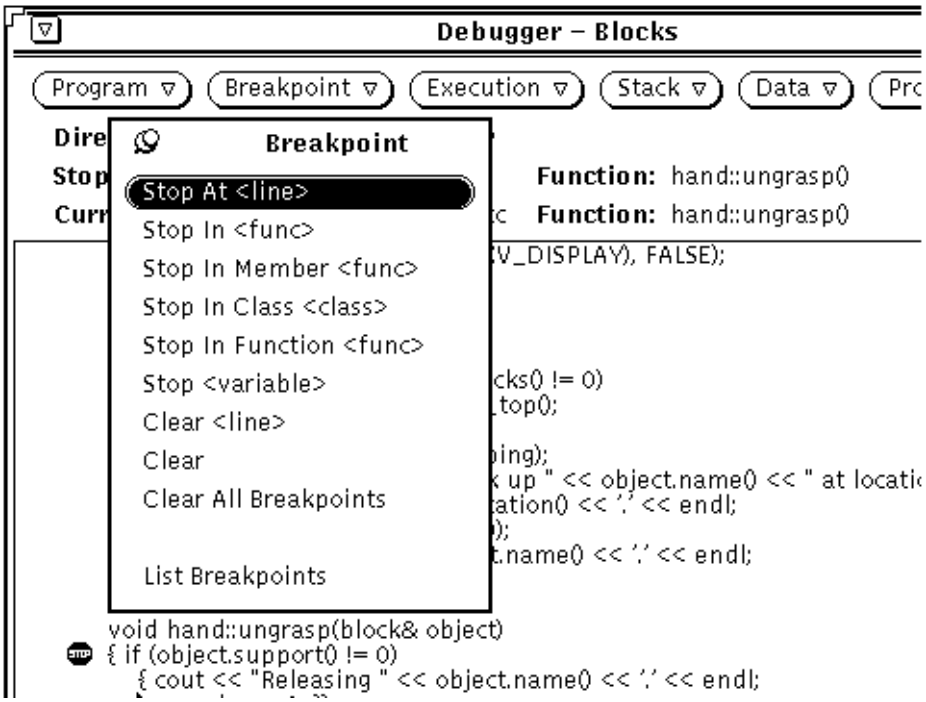
List Breakpoints — list all currently active breakpoints and traces.

5.1.1 Setting a Breakpoint at a Line of Source Code

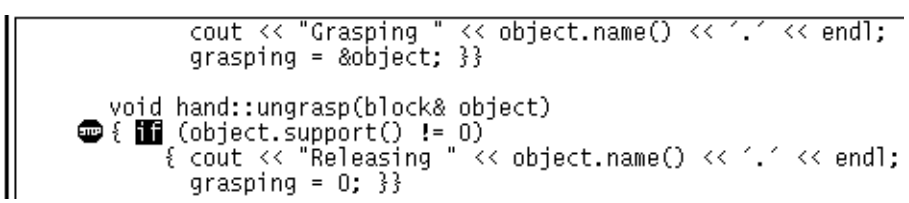
To set a breakpoint at the start of a line of source code:

- **Select any character in the line, then choose Stop At <line> from the Breakpoint menu.**

.



You can also use the stop at button in the Debugger base window display. The Debugger shows the "stop sign" breakpoint glyph to the left of the line at which you set the breakpoint.



You can set a breakpoint at a line from the command pane, using the Debugger `stop at` command:

`stop at [`*filename*`:]`*n*

where `n` is a source code line number and *filename*`:` is an optional program filename qualifier. For example,

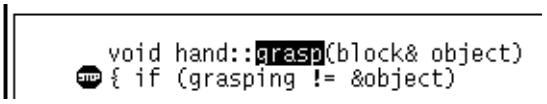
```
(debugger) stop at main.cc:3
```

To use the `stopi` command for setting a breakpoint at a machine instruction, see "Setting Breakpoints at Machine-instruction Level" on page 15-194.

5.1.2 Setting a Breakpoint in a Function

To set a simple breakpoint on a function:

- **Select the name of the function, then choose Stop In <func> from the Breakpoint menu.**
  You can also use the stop in button in the Debugger base window. The Debugger shows the breakpoint glyph to the left of the first line in the function.



When selecting a function name, do not select the parentheses or the parameters.

5.1.3 Setting a Breakpoint in a Dynamically Linked Library

The Debugger provides full debugging support for code that makes use of the programmatic interface to the run-time linker; that is, code that calls `dlopen()`, `dlclose()` and their associated functions. The run-time linker binds and unbinds shared libraries during program execution. Debugging support for `dlopen()/dlclose()` allows you to step into a function or set a breakpoint in functions in a dynamically shared library just as you can in a library linked when the program is started.

Two exceptions:

- You cannot set a breakpoint in a `dlopen`'ed library before that library is loaded by `dlopen()`.

(Solaris 2.x)

- When a library is loaded by `dlopen()`, an initialization routine named `_init()` is called. This routine may call other routines in the library. The Debugger cannot place breakpoints in the loaded library until after this initialization is completed. In specific terms, this means you cannot have the Debugger stop at `_init()` in a library loaded by `dlopen`.

5.2 Setting Multiple Breaks in C++ Programs

The nature of debugging object-oriented programming is such that you may want to check for problems related to calls to members of different classes, calls to any members of a given class, or calls to overloaded top-level functions. You can use one of three keywords—`inmember`, `inclass`, `infunction`—with a `stop` or `when` command to set multiple breaks in C++ code.

The Breakpoints menu contains the `stop` command versions of these keyword commands: `Stop in Member`, `Stop in Class`, `Stop in Function`. Enter the `when` variants from the command pane.

When you set multiple breakpoints, the command pane reports them as a single set of breakpoints.

```
(debugger) stop inclass hand
(2) stop inclass hand
(debugger)
```

5.2.1 Set Breakpoints in Member Functions of Different Classes

To set a breakpoint in each of the object-specific variants of a particular member function (same member function name, different classes),

- **Select the member function name in the source display and then choose Stop in Member from the Breakpoint menu.**
  To set a `when` type breakpoint, use the `when` command with the keyword `inmember`. For example, in the Blocks demo program, a member function `draw()`, is defined in each of five different classes (hand, brick, ball, wedge, table). To place a breakpoint in each `draw()` function:

  ```
  (debugger) when inmember draw {cmd;}
  ```

5.2.2 Setting Breakpoints in Member Functions of Same Class

To set a breakpoint in all member functions of a specific class:

- **Select the class name in the source display and then choose Stop in Class from the Breakpoint menu.**
  To set a `when` type breakpoint, use the `when` command with the keyword `inclass`. For example, in the Blocks demo program, to set a breakpoint in all member functions of the class hand:

```
(debugger) when inclass hand {cmd;}
```

Note – Breakpoints are inserted in only the class member functions defined in this class. It does not include those that it may inherit from base classes.

5.2.3 Setting Multiple Breakpoints in Nonmember Functions

To set multiple breakpoints in nonmember functions with overloaded names (same name, different type or number of arguments):

- **Select the function name in the source display and then choose Stop in Function from the Breakpoint menu.**
  To set a `when` type break, use the `when` command with the keyword `infunction`. For example, if a C++ program has defined two versions of a function named `sort()`, one which passes an `int` type argument, the other a `float`, then, to place a breakpoint in both functions:

```
(debugger) when infunction sort {cmd;}
```

5.2.4 Setting a `when` Type Breakpoint at a Line

A `when` type breakpoint command accepts other Debugger commands as side-effect commands. `when` accepts commands like `list`, allowing you to write your own version of trace.

```
(debugger) when at 123 { list $lineno; }
```

Note – `when` operates with an implied `continue`. In the example above, after listing the source code at the current line, the program continues executing.

5.3 Tracing Code

Tracing displays information in the command pane about the line of code about to be executed or a function about to be called. `Trace` monitors the execution of a specified line of code, all lines within a specified function, or even the whole program. The information printed as each line comes up for execution depends on the type of trace you set and the line of code to be executed.

The location in the program where a `trace` begins is called the *tracepoint*. The Debugger turns off tracing once the program leaves the scope of the trace.

5.3.1 Setting `trace` Commands

Set `trace` commands from the command pane. The following table shows the command syntax for the types of traces that you can set. The information a `trace` provides depends on the type of *event* associated with it.

If you use a command:	**Then, `trace` prints:**
`trace`	echo every line in program as it is about to be executed.
`trace in` *function*	echo every line while the program is in the function
`trace [at]` *line_number*	line number and the line itself, as that line becomes the next line to be executed.
`trace` *expression* `at` *line_number*	value of *expression* when *line_number* is next line to be executed.
`trace` *function*	name of the function that called *function*; line number, parameters passed in, and return value.

`trace inmember` *member_function*	name of the function that called *member_function* of any class; its line number, parameters passed in, and its return value.
`trace inclass` *class*	name of the function that called any member_function in *class*; its line number, parameters passed in, and return value.
`trace infunction` *function*	name of the function that called any nonmember function in C++ program; its line number, parameters passed in and its return value.
`trace` *variable* `[in` *function*`]`	new value of *variable*, if it changes, and the line at which it changed.

5.3.2 Controlling the Speed of a `trace`

The Debugger tries to follow a `trace` in the source display by placing an arrow glyph to the left of the line that is about to be executed. In many programs, code execution is so fast that you would not see much. An environment variable allows you to control the time interval between the execution of each line of code within the scope of the `trace`. The default interval is `0.5` seconds.

To set the interval between execution of each line of code during a `trace`:

1. **Choose Debugger Events from the Category menu of the Properties Window.**

2. **Type the time interval in the Trace Speed control text field and press Apply.**

5.4 Listing and Clearing Event Management Commands

Often, you set more than one breakpoint or tracepoint during a debugging session. The Debugger supports commands for listing and clearing them.

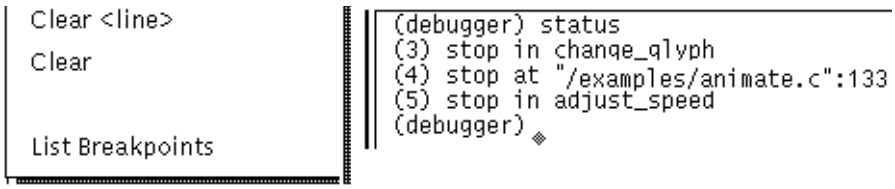
5.4.1 Listing Breakpoints and Traces

The List Breakpoints item on the Breakpoints menu lists all currently active breakpoint commands in the command pane.

To display a list of all active breakpoints in the command pane:

- **Choose List Breakpoints from the Breakpoint menu.**
  List Breakpoints is the same as the Debugger `status` command.

  The Debugger echoes the command and lists the breakpoints in the command pane.



  Notice the ID number in parentheses to the left of each listed breakpoint. You can use these IDs to clear the corresponding breakpoints.

  As noted in the section on C++ multiple breakpoints, the Debugger reports multiple breakpoints set with the `inmember`, `inclass`, and `infunction` keywords as a single set of breakpoints with one status ID number.

5.4.2 Clearing a Breakpoint at a Line

To clear a single breakpoint at a specific line:

1. **Select the line in the source display where the breakpoint is set.**
2. **Choose Clear <line> from the Breakpoint menu.You can also use the clear button from the Debugger base window.**

5.4.3 Clearing a Breakpoint from the Current Focus Line

To clear a breakpoint set at the current focus line:

- **Choose Clear from the Breakpoint menu.**

5.4.4 Deleting Specific Breakpoints Using Status ID Numbers

When you list event management commands in the command pane (using either List Breakpoints or the Debugger `status` command), the Debugger assigns an ID number to each event handler. Using the `delete` command in the command pane, you can remove event handlers by ID number, or use the keyword `all` to remove all event handlers currently set anywhere in the program.

To delete event handlers by ID number,

- **Use the** `delete` *ID_number* **command in the command pane.**

```
(debugger) delete 3 5
```

To delete all event management commands set in the program currently loaded in the Debugger,

- **Give the** `delete` **command an argument of** `all`**:**

```
(debugger)  delete all
```

5.4.5 Watchpoints

Watchpointing is the general capability of the Debugger to note when the value of a variable or expression has changed and fire an event.

(Solaris 2.x) *Stopping Execution When the Contents of an Address gets Written to*

To stop program execution when the contents of an address gets written to:

```
(debugger) stop modify &<variable>
```

(Solaris 2.x) ***Points to keep in mind when using*** `stop modify`

- The event occurs when a variable gets written to even if it is the same value.
- The event occurs at the instruction that wrote to the variable, although its new contents are set. The program location arrow typically points to the line that does the modification.
- You cannot use addresses of stack variables, for example, automatic function local variables.

Stopping Execution When a Specified Variable Changes

To stop program execution if the value of a specified variable changes:

♦ **Use the** `stop` *variable* **or the** `when` *variable* **command formula in the command pane:**

```
(debugger) stop variable

(debugger) when variable {cmd;...}
```

The `when` *variable* command accepts follow-on commands, as explained in "Setting a when Type Breakpoint at a Line" on page 5-69.

Points to Keep in Mind When Using `stop` *variable*

Note that, except for the first point, these points apply to using `trace` *variable* as well:

- The Debugger stops the program *at the line after* the line that caused a change in the value of the specified variable. So, when the program stops, the program location arrow does not point to the actual line that triggered the change in value. (Does not apply to `trace` *variable*.)
- If *variable* is local to a function, the variable is considered to have changed when the function is first entered and storage for *variable* is allocated. The same is true with respect to parameters. (Applies also to `trace` *variable*.)

  The Debugger implements `stop` *variable* by causing automatic single stepping together with a check on the value at each step. Stepping skips over library calls. So, if control flows in the following manner:

```
user_routine calls
   library_routine, which calls
      user_routine2 // changes variable
```

The Debugger does not trace the nested *user_ routine2.* It does not trace the nested call because tracing skips the library call and the nested call to `user_routine2`, so the change in the value of *variable* appears to have occurred after the return from the library call, not in the middle of user_routine2. (Applies also to `trace` *variable*.)

- The Debugger cannot set a breakpoint for a change in a block local variable, that is, a variable nested in {}. If you try to set a breakpoint (or trace) a block local "nested" variable, the Debugger issues an error informing you that it cannot perform this operation. (Applies also to `trace` *variable*.)

Stopping Execution if a Specified Condition Occurs

To stop program execution if a conditional statement evaluates to true:

♦ **Use** `stop -if` *condition***:**

```
(debugger) stop -if condition
```

5.5 Setting Event Filters

In the Debugger, most of the event management commands also support an *optional event filter* modifier statement. The simplest filter (see Chapter 6, "Event Management") instructs the Debugger to test for a condition:

- After the program arrives at a breakpoint or tracepoint, or
- After a watch condition occurs.

If this filter condition evaluates to true (non 0), the event command applies. If the condition evaluates to false (0), the Debugger continues program execution as if the event never happened. To set a breakpoint at a line or in a function that includes a conditional filter:

♦ **Add the optional** `-if` *condition* **modifier statement to the end of a stop or trace command**

The condition can be any valid expression, including function calls, returning Boolean or integer in the language current at the time the command is entered.

Note – The scope of the condition is the scope at the time of entry, not at the time of the event. So you may have to use syntax to specify the scope precisely.

New users sometimes confuse setting a conditional event command (a watch-type command) with using filters. Conceptually, "watching" creates a *precondition* that must be checked before each line of code executes (within the scope of the watch). But even a breakpoint command with a conditional trigger can also have a filter attached to it. Consider this case: first the syntax, then a specific example,

```
stop modify <address> -if condition
```

```
(debugger) stop modify &speed -if speed==fast_enough
```

This command instructs the Debugger to monitor the variable, *speed*; if the variable *speed* is written to (the "watch" part), then the `-if` filter goes into effect. The Debugger checks to see if the new value of *speed* is equal to `fast_enough`. If it is not, the program continues on, "ignoring" the `stop`.

In the Debugger syntax, the filter is represented in the form of an `[-if` *condition*`]` statement at the end of the formula:

```
stop in function [-if condition]
```

5.6 Event Efficiency

Various events have varying degrees of overhead in respect to the execution time of the program being debugged. Some events, like the simplest breakpoints have practically no overhead. Events based on a single breakpoint, like

`trace` <*lineno*>

`trace` <*expression*> `at` <*lineno*>

also have minimal overhead. Multiple breakpoints (such as `inclass`) that might sometimes result in hundreds of real breakpoints, have an overhead only during installation time. This is because dbx uses permanent breakpoints, that is, the breakpoints are retained in the process at all times and are not taken out on every stoppage and put in on every `cont`.

Note – In the case of `step` and `next`, by default all breakpoints are taken out before the process is resumed and reinserted once the step completes. If you are using many breakpoints or multiple breakpoints on prolific classes the speed of `step` and `next` slows down considerably. Use the dbxenv `stepevents` variable to control whether breakpoints are taken out and reinserted after each `step` or `next`.

The slowest events are those that utilize automatic single stepping. This might be explicit and obvious as in the plain `trace` command, which single steps through every source line. Other events, like `stop` <*expr*> or `trace` <*variable*> (a class of events called *watchpoints*) not only single step automatically but also have to evaluate an expression or a variable at each step.

These are very slow, but you can often overcome the slowness by bounding the event with a function using the `-in` modifier. For example:

- `trace -in mumble`

  Do not use `trace -in main` because the `trace` is effective in the functions called by `main` as well.

- `stop clobbered_variable -in lookup`

  Do use in the cases where you suspect that the `lookup()` function is clobbering your variable.

(Solaris 2.x)

The Faster `modify` *Event*

A faster way of doing watchpoints is to use the `modify` event. For information on the `modify` event, see Section 6.4, “Event Specifications,” on page 6-84.

Instead of automatically single-stepping the program, it uses a page protection scheme which is much faster. The speed depends on how many times the page on which the variable you are watching is modified, as well as the overall system call rate.

5

Event Management 6

This chapter describes *event management*. Event management refers to the general capability of the Debugger to handle events by performing certain actions when certain events take place in the program being debugged.

This chapter is organized into the following sections:

Event Management Commands	*page 6-80*
Commands to Manipulate the Event Handlers	*page 6-83*
Event Counters	*page 6-84*
Event Specifications	*page 6-84*
Event Specification Modifiers	*page 6-91*
A Note on Parsing and Ambiguity	*page 6-93*
Additional Process Control Commands & Enhancements	*page 6-94*
Predefined Variables	*page 6-95*
Examples	*page 6-98*

The most common example of the association of a program event with a Debugger action is setting a breakpoint on a particular line. A change in the value of a specified variable is another type of event that triggers a stop.

Event management is based on the concept of a "*Handler*". The name comes from an analogy with hardware interrupt handlers. Each event management command typically creates a handler. To do that an *event specification* and a series of side-effect actions need to be specified.

6

The most generic form of creating a handler is through the `when` command:

`when` *event-specification* {*action*; ... }

Although all event management can be performed through `when`, dbx has historically had many other commands. Those commands are still retained, either for backward compatibility, or because they are simpler and easier to use.

In many places examples are given on how a command (like `stop`, `step`, or `ignore`) can be written in terms of `when`. These examples are meant to illustrate the flexibility of the `when` and the underlying *handler* mechanism, but they are not always exact replacements.

6.1 Event Management Commands

The following commands (`when`, `stop`, and `trace`) create event handlers. An *event-spec* is a specification of an event as documented later in this chapter.

An attempt has been made to make the `stop` and `when` commands regular and to make them conform to the handler model. However, backward compatibility forces some deviations. For instance:

```
when cond body
when cond in func body
```

are really equivalent to

```
when step -if cond body
when next -if cond -in func body
```

The point being illustrated here is that *cond* is not a pure event; there is no internal handler for "conditions".

Every command returns a number known as a handler id (*hid*). This number can be accessed via the predefined variable `$newhandlerid`.

when

`when` *event-spec* [*modifier* ...] { *dbx-cmd* ; [...] }
`wheni` *event-spec* [*modifier* ...] { *dbx-cmd* ; [...] }

When the specified event occurs, execute the series of dbx commands. Once the commands have all executed, the process is automatically continued.

stop `stop` *event-spec* [*modifier ...*]
`stopi` *event-spec* [*modifier ...*]

When the specified event occurs, stop the process and notify the user. `stop` is shorthand for a common `when` idiom:

```
when event-spec { stop -update; whereami; }
```

trace `trace` *trace-event-spec*

The `trace` command is divided into three categories, and all categories can be filtered. The categories are described in this section.

Line Tracing

`trace`
Automatically steps and prints every line that is executed. Function calls and returns are traced as well.

`trace in` *func*

A "bounded" version of `trace`. Automatically performs a `next` and prints every line within the given function.

`trace [ at ]` *line*

Prints the line every time you go through it. Implemented using breakpoints.

Function Tracing

`trace` *func*

Prints information about every entry and return from the given function.

`trace inmember/infunction` *func*

Prints information about every entry and return from the given member or overloaded function.

`trace inclass` *class*

Prints information about every entry and return from members of the given class.

Variable and Expression Tracing

`trace` *expr* `at` *line*

Is equivalent to:

```
when at line { print expr; }
```

`trace` *expr* `[ in` *func* `]`

Automatically single-steps the program and if the given expression changes value, prints the new value. If `in` *func* is used, the stepping and checking is "bounded" by the given function.

If you do not provide an `in` but *expr* is comprised of local variables, dbx automatically determines the function and adds in an implicit `in` *func*. If the expression is too complicated and involves variables from multiple function scopes, the heuristics that dbx uses to determine the bounding function may fail.

Filtering of Traces

`trace` ... `-if` *cond*

All of the `trace` forms can be filtered by using the `-if` notation.

Improving `Trace` *Command Efficiency*

Most of the `trace` commands described can be hand-crafted by using the `when` command, ksh functionality, and event variables. This is especially useful if you want stylized tracing output.

`Trace` functions that depend on automatic single-stepping are inherently slow, particularly for tracing variable changes. Use the "`when modify`" command for a faster version. For a general discussion of speed and efficiency, see Section 5.6, "Event Efficiency," on page 5-76.

The `when`, `stop`, and `trace` commands are fully backward compatible. In addition, the `stop` and `when` commands have many new extensions as documented later in this chapter.

6.2 Commands to Manipulate the Event Handlers

The following list contains commands to manipulate event Handlers:

status

`status` [-s]
`status` [*-s*] *hid*

`status` with no argument lists all handlers, that is, `trace`, `when`, and `stop`; `status` *hid* lists the given handler. If the handler is disabled, its *hid* is printed surrounded by [ ] square brackets instead of the normal ( ) parentheses.

The output of `status` can be redirected to a file. Nominally, the format is unchanged during redirection. You can use the `-s` flag to produce output that allows a handler to be reinstated by using the `source` command.

The original technique of redirecting handlers by using `status` and `source`'ing the file was a way to compensate for the lack of handler enabling and disabling functionality (see `handler -enable`).

delete

`delete` *hid* [*hid* ...]
`delete all`
`delete -all`
`delete` 0
`delete -temp`

`delete` *hid* deletes the specified handler; `delete all` or `delete` *0* (zero) deletes all handlers. `delete -all` deletes all handlers including temporary handlers. `delete -temp` deletes all temporary handlers.

clear

`clear`
`clear` *line*

`clear` with no arguments deletes all handlers based on breakpoints on the current line; `clear` *line* deletes all handlers based on breakpoints on the designated line.

handler disable

`handler -disable` *hid* [...]
`handler -disable all` [...]

`handler -disable` *hid* disables the specified event; `handler -disable all` disables all handlers.

handler enable

`handler -enable` *hid* [...]
`handler -enable all`

`handler –enable` *hid* enables the specified handler; `handler –enable all` enables all handlers.

6.3 Event Counters

Event handlers have trip counters. There is a count limit and the actual counter. Whenever the event occurs, the counter is incremented. The event fires only if the count reaches the limit, at which point the counter is automatically reset to 0. The default limit is 1. Whenever a process is rerun, all event counters are reset.

The count limit can be set using the `–count` modifier as explained in Section 6.5, "Event Specification Modifiers," on page 6-91. Otherwise, the following commands can be used to manipulate event handlers individually.

handler

`handler –count` *hid*
`handler –count` *hid new-count*
`handler –reset` *hid*

`handler –count` *hid* returns the count of the event in the form *current-count/limit* (same as printed by `status`). *limit* might be the keyword `infinity`. Use the ksh modifiers `${#}` and `${##}` to split the printed value.

`handler –count` *hid new-count* assigns a new count limit to the given handler.

`handler –reset` *hid* resets the count of the handler to 0 (zero).

6.4 Event Specifications

Event specifiers are used by the `stop` and `when` commands to denote event types and parameters. The format is that of a keyword to represent the event type and optional parameters. Exceptions are *expression* (included to maintain backwards compatibility) and the `trace` command, where the event specifier syntax is not as regular.

When no keyword is recognized, an expression is assumed. Also, depending on whether `stop` or `when` is used, *expression* is either a variable (value) to be watched, or has to be a boolean valued expression to be used as a condition.

A list of all types, their eventspec syntax, and semantics follows:

in *func*
The function has been entered and the first line is about to be executed. If the `-instr` modifier is used, it is the first instruction of the function about to be executed. (Do not confuse `in func` with the `-in` *func* modifier. For information on the `-in` *func* modifier, see "Event Specification Modifiers" on page 6-91.)

The *func* specification can take a formal parameter signature to help with overloaded function names, or template instance specification. For example, you can say:

```
stop in mumble(int, float, struct Node *)
```

at *lineno*
at *filename:lineno*
The designated line is about to be executed. If *filename* is provided, then the designated line in the specified file is about to be executed.

infunction *func*
Equivalent to `in` *func* for all overloaded functions named *func*, or all template instantiations thereof.

inmember *func*
inmethod *func*
Equivalent to `in` *func* for the member function named *func* for every class.

inclass *classname*
Equivalent to `in` *func* for all member functions that are members of *classname*.

expression
With `stop`: the value of *expression* has changed. *expression* must be an lvalue.

With `when`: the *expression* evaluates to true. *expression* must be a boolean value.

To implement this event, dbx automatically single-steps the application and checks for the condition on each single step.

Note that event specifications are keyword driven. This one, *expression*, is an exception for backward compatibility. Whenever no keyword is recognized, an expression is assumed.

See Section 5.4.5, "Watchpoints," on page 5-73 for important points to remember when using `stop`.

returns This event is just a breakpoint at the return point of the current *visited* function. The visited function is used so that you can use the `returns` event spec after doing a number of '`up`'s. The plain return event is always `-temp` and can only be created in the presence of a live process. As an example, consider that

```
step up
```

is equivalent to:

```
when returns {
    stop; echo returned to $func
}
cont
```

returns *func* This event fires each time the given function returns to its call site. This is not a temporary event. The return value is not provided, but you can find it through:

`$o0` on SPARC
`$eax` on x86

It is another way of saying:

```
when in func ( stop returns; )
```

step The `step` event fires when PC reaches the first instruction of a source line. For example, the `trace` command is equivalent to:

```
when step { echo $lineno: $line; }
```

When enabling a step event you basically instruct dbx to single-step automatically next time `cont` is used.

The `step` *command* can be implemented as follows:

```
alias step="when step -temp { whereami; stop; }; cont"
```

next Similar to `step` except that functions are not stepped into.

throw `throw` **[***type***]**

This event fires when a C++ `throw` statement is executed with an expression of type *type*. The rules for matching an expression of a specific type are as follows:

A handler type T matches a `throw` type E if any of the following is true,

- T is same as E;
- T is const or volatile of E;
- E is const or volatile of T;
- T is ref of E or E is ref of T;
- T is a public or protected base of E;
- T and E are both pointer types and E can be converted to T by standard pointer conversion.

If no T type is provided, any C++ throw statement causes this event.

Note – While handlers of type `(X)` and `(X&)` both match an exception of type `X`, the semantics are different. The use of a handler with type (`X`) results in the invocation of its copy constructor and possible truncation of the object (happens when the exception is derived from `X`).

---

sig *sig* When the specified signal is first delivered to the child this event fires. *sig* can either be a decimal number or the signal name in either upper or lower case, with the "SIG" prefix being optional. This is completely independent of the `catch/ignore` commands although the `catch` command can be implemented as follows:

```
function simple_catch
{
    when sig $1 {
        stop;
        echo Stopped due to $sigstr $sig
        whereami
        }
}
```

The `ignore` command disables anything established by `catch`, it doesn't mask out signals.

Note – When the sig event is received, the process is still alive. Only if you `cont` the process with the given signal is the signal forwarded to it.

---

sig *sig sub-code* When the specified signal with the specified sub-code is first delivered to the child this event fires.

Just as with signals, the sub-code can be entered as a decimal number, in capital or lower case; the prefix is optional.

A list of the names of all signals and sub-codes and their numerical equivalents can be accessed using "help signals."

fault ***fault***
(Solaris 2.x)

This event fires when the specified fault occurs. The faults are architecture dependent, but a set of them is known to dbx as defined by `proc` (4):

`FLTILL`	Illegal instruction
`FLTPRIV`	Privileged instruction
`FLTBPT`	Breakpoint instruction
`FLTTRACE`	Trace trap (single step)
`FLTACCESS`	Memory access (such as alignment)
`FLTBOUNDS`	Memory bounds (invalid address)
`FLTIOVF`	Integer overflow
`FLTIZDIV`	Integer zero divide
`FLTPE`	Floating-point exception
`FLTSTACK`	Irrecoverable stack fault
`FLTPAGE`	Recoverable page fault

These faults are taken from `sys/fault.h`. *fault* can be any of those listed above, in upper or lower case, with or without the `FLT-` prefix, or the actual numerical code. Be aware that `BPT`, `TRACE`, and `BOUNDS` are used by dbx to implement breakpoints, single-stepping, and watchpoints. Handling them may interfere with the inner workings of dbx.

stop The process has stopped. Whenever the process stops such that the user gets a prompt, particularly in response to a `stop` handler, this event is fired.

sync The process being debugged has just been `exec`'ed. All memory specified in `a.out` is valid and present but pre-loaded shared libraries have not been loaded yet. For example, `printf`, although known to the Debugger, has not been mapped into memory yet.

A `stop` on this event is ineffective; however, you can use this event with the `when` command.

syncrtld This event is fired after a `sync` (or `attach` if the process being debugged has not yet processed shared libraries). It fires after the startup code has executed and the symbol tables of all preloaded shared libraries have been loaded.

A `stop` on this event is ineffective; however, you can use this event with the `when` command.

attach The process being debugged has just attached to the process to be debugged.

lastrites The process being debugged is about to expire. There are only three reasons that this can happen:

- The `_exit`(2) system call has been called. (This happens either through an explicit call, or when `main`() returns.)
- A terminating signal is about to be delivered.
- The process is being killed by the dbx `kill` command.

dlopen [*lib-path*]
dlclose [*lib-path*] These events are fired after a `dlopen`() or a `dlclose`() call succeeds. A `dlopen`() or `dlclose`() call can cause more than one library to be loaded. The list of these libraries is always available in the predefined variable `$dllist`. The first word in `$dllist` is actually a "+" or a "-", indicating whether the list of libraries is being added or deleted.

lib-path is the full pathname of a shared library you are interested in. If it is specified, the event only fires if the given library was loaded or unloaded. In that case `$dlobj` contains the name of the library. `$dllist` is still available.

If *lib-path* is not specified, then the events always fire whenever there is any `dl`-activity. `$dlobj` is empty but `$dllist` is valid.

modify ***addr-exp*** **[** **,** ***byte-size*** **]**

(Solaris 2.x) The specified address range has been modified. This is the general "watchpoint" facility. Watchpointing is the general capability of the Debugger to note when the value of a variable or expression has changed, and fire an event.

The syntax of the event-specification for watchpoints is:

`modify` **<***addr-exp***>** [, **<***byte-size-exp***>**]

<*addr-exp***>** is any expression that can be evaluated to produce an address. If a symbolic expression is used, the size of the region to be watched is automatically deduced, or you can override that with the `,' syntax. You can also use nonsymbolic, typeless address expressions; in which case, the size is mandatory. For example:

```
stop modify 0x5678, sizeof(Complex)
```

There are two styles of watchpoints in dbx. The older, slower style uses automatic single-stepping and is invoked by using:

`trace` **<***expr***>**
`stop` **<***cond***>**
`when` **<***variable***>** {`...`}

Limitations

Addresses on the stack cannot be watched.

The event is not fired if the address being watched is modified by a system call.

Shared memory (MAP_SHARED) cannot be watched, because the Debugger cannot catch the other processes stores into shared memory. Also, the Debugger cannot properly deal with SPARC `swap` and `ldstub` instructions.

sysin *code*|*name*

(Solaris 2.x)

The specified system call has just been initiated and the process has entered kernel mode.

The concept of "system call" supported by dbx is that provided by `procfs(4)`. These are traps into the kernel as enumerated in "`/usr/include/sys/syscall.h`".

This is not the same as the ABI notion of system calls. Some ABI system calls are partially implemented in user mode and use non-ABI kernel traps. However, most of the generic system calls (the exception being signal handling) are the same between "`syscall.h`" and the ABI.

sysout *code*|*name*

(Solaris 2.x)

The specified system call is finished and the process is about to return to user mode.

sysin
sysout

(Solaris 2.x)

Without arguments, all system calls are traced. Note that certain dbx features, for example `modify` event and RTC, cause the child to execute system calls for their own purposes and that these show up if traced.

6.5 *Event Specification Modifiers*

An Event Specification Modifier sets additional attributes of a handler, the most common kind being event filters. Modifiers have to appear after the keyword portion of an eventspec. They all begin with a '–', preceded by blanks. The non '–' version provides backward compatibility.

Modifiers consist of the following:

-if `-if` *cond*
`if` *cond*

The condition is evaluated when the event specified by the *event-spec* occurs. The event is fired only if the condition evaluates to nonzero. The non '–' version provides backward compatibility.

The condition is sometimes called a filter. A handler created with a filter is known as a filtered handler.

-in `-in` *func*
`in` *func*

The handler is enabled only while within the given function, or any function called from *func*. See Example Section 6.9.3, “Enable Handler While Within the Given Function (in func),” on page 6-98.

The number of times the function is entered is reference counted so as to properly deal with recursion.

The handler that has been modified by the `-in` modifier is said to be "bounded by *func*".

-disable `-disable`

Create the handler in the disabled state (see “`handler -disable`”).

-count `-count` *n*
`-count infinity`

Have the handler count from 0. Each time the event occurs, `count` is incremented until it reaches *n*. Once that happens, the handler fires. The `status` command prints out the count in the form

curcount/limit

Counts of all enabled handlers are reset when a program is run or rerun. More specifically, they are reset when the `sync` event occurs (see “`handler -count`”).

-temp `-temp`

Create a temporary handler. Once the event is fired it is automatically deleted. By default, handlers are not temporary. If the handler is a counting handler, it is automatically deleted only when the count reaches 0 (zero).

You use the `delete -temp` command to delete all temporary handlers.

-instr `-instr`

Makes the handler act at an instruction level. This replaces the traditional '`i`' suffix of most commands. It usually modifies two aspects of the event handler:

- Any message prints assembly level rather than source level information.
- The granularity of the event becomes instruction level. For instance, `step -instr` implies instruction level stepping.

-thread `-thread` *tid*

(Solaris 2.x) The event is fired only if the thread that caused it matches *tid*.

-lwp `-lwp` *lid*

(Solaris 2.x) The event is fired only if the thread that caused it matches *lid*.

6.6 A Note on Parsing and Ambiguity

Syntax for event-specs and modifiers is

- Keyword driven
- Based on ksh conventions, mainly that everything is split into "words" delimited by spaces

For the sake of backward compatibility, expressions can have spaces embedded in them. This can cause ambiguous situations. For example, consider the following two commands:

```
when a -temp          # first example
```

and

```
when a-temp           # second example
```

In the first example, even though the application might have a variable named *temp*, the dbx parser resolves the *event-spec* in favor of `-temp` being a modifier. In the second example, `a-temp` is collectively passed to a language specific expression parser and there must be variables named *a* and *temp* or an error occurs.

Use parentheses to force parsing. For instance, "`when` (`a -temp`)" is equivalent to the second example.

6.7 Additional Process Control Commands & Enhancements

stop `stop`
`stop -update |-noupdate`

This form of the `stop` command is valid only inside the body of a `when`. Whereas normally the process is continued after the body has executed, the `stop` command prevents that. Normally the `stop` command causes an update message to be sent to various interested parties, like the "display" mechanism, and under the GUI, the Data Inspector and Stack Inspector. `-noupdate` disables the message.

In general, the following are equivalent:

`stop` *event-spe*

is equivalent to

`when` *event-spec* `{ whereami; stop; }`

and

`stopi` *event-spec*

is equivalent to

`when` *event-spec* `-instr { whereami -instr;stop; }`

step `step [ -sig` *sig* `]`

The `step` command is equivalent to:

```
when step -temp { stop; }; cont
```

The `step` command can take a `sig` argument. A `step` by itself cancels the current signal just like `cont` does. To forward the signal, you must explicitly give the signal. You can use the variable `$sig`, as in:

```
step -sig $sig
```

to step the program sending it the current signal.

cancel `cancel`

Only valid within the body of `when`. The `cancel` command cancels any signal that might have been delivered, and lets the process continue. For example:

```
when sig SIGINT { echo signal info;  cancel; }
```

6.8 Predefined Variables

Certain read-only ksh predefined variables are provided. The following are always valid:

`$pc`	current program counter address (hexadecimal)
`$ins`	disassembly of the current instruction
`$lineno`	current line number in decimal
`$line`	contents of the current line
`$func`	name of the current function
`$vfunc`	name of the current "visiting" function
`$class`	name of the class to which `$func` belongs
`$vclass`	name of the class to which `$vfunc` belongs
`$file`	name of the current file
`$vfile`	name of the current file being visited
`$loadobj`	name of the current loadable object
`$vloadobj`	name of the current loadable object being visited
`$funcaddr`	address of `$func` in hex
`$caller`	name of the function calling `$func`
`$dllist`	after `dlopen` or `dlclose` event, contains the list of load objects just dlopened or dlclosed.

		The first word of `dllist` is actually a "+" or a "-" depending on whether a `dlopen` or a `dlclose` has occurred.
	`$newhandlerid`	id of the most recently created handler
	`$proc`	process id of the current process being debugged
(Solaris 2.x)	`$lwp`	lwp id of the current LWP
(Solaris 2.x)	`$thread`	thread id of the current Thread
	`$prog`	full pathname of the program being debugged

As an example, consider that `whereami` can be roughly implemented as:

```
function whereami {
      echo Stopped in $func at line $lineno in file $(basename $file)
        echo "$lineno\t$line"
```

6.8.1 Event-Specific Variables

The following are only valid within the body of a `when`.

`$handlerid`

During the execution of the body, `$handlerid` is the id of the `when` command to which the body belongs. As a simple example consider that

```
when X -temp { do_stuff; }
```

is equivalent to

```
when X  { do_stuff; delete $handlerid; }
```

The following are only valid within the body of a `when` and in addition are event specific.

For Event `sig`

`$sig`	signal number that caused the event
`$sigstr`	name of `$sig`

`$sigcode`	subcode of `$sig` if applicable
`$sigcodestr`	name of `$sigcode`

For Event `exit`

`$exitcode`	value of the argument passed to `_exit`(2) or `exit`(3) or the return value of `main`.

For Events `dlopen` *and* `dlclose` *(Only if a Param Was Provided)*

`$dlobj`	pathname of the load object dlopened or dlclosed

(Solaris 2.x)

For Events `sysin` *and* `sysout`

`$syscode`	system call number
`$sysname`	system call name

6.9 Examples

6.9.1 Set Watchpoint for store to Array Member

(Solaris 2.x) To set a watchpoint on `array`[99]:

```
(dbx) stop modify &array[99]
(2) stop modify &array[99], 4
(dbx) run
Running: watch.x2
watchpoint array[99] (0x2ca88[4]) at line 22 in file
"watch.c"
     22    array[i] = i;
```

6.9.2 Simple Trace

To implement a simple trace:

```
(dbx) when step { echo at line $lineno; }
```

6.9.3 Enable Handler While Within the Given Function (`in` *func)*

For example:

```
trace -in foo
```

is equivalent to something like:

```
# create handler in disabled state
when step -disable { echo Stepped to $line; }
t=$newhandlerid    # remember handler id
when in foo {
    # when entered foo enable the trace
     handler -enable "$t"
     # arrange so that upon returning from foo,
     # the trace is disabled.
     when returns { handler -disable "$t"; };
}
```

6.9.4 Determine the Number of Lines Executed in a Program

To see how many lines were executed in a small program:

```
(dbx) stop step -count infinity # step and stop when count=inf
(2) stop step -count 0/infinity
(dbx) run
...
(dbx) status
(2) stop step -count 133/infinity
```

We obviously never stop—the program terminates. 133 is the number of lines executed. This process is very slow though. This technique is more useful with breakpoints on functions that are called many times.

6.9.5 Determine the Number of Instructions Executed by a Source Line

To count how many instructions a line of code executes (this was a requested RFE).

```
(dbx) ...                                  # get to the line in question
(dbx) stop step -instr -count infinity
(dbx) step ...
(dbx) status
(3) stop step -count 48/infinity # 48 instructions were executed
```

If the line you are stepping over makes a function call, you end up counting those as well. You can use the `next` event instead of `step` to count instructions, excluding called functions.

6.9.6 Enable Breakpoint after Event Occurs

Enable a breakpoint only after another event has fired. Suppose things go bad in function `hash`, but only after the 1300'th symbol lookup:

```
(dbx) when in lookup -count 1300 {
      stop in hash
      hash_bpt=$newhandlerid
      when lastrites -temp { delete $hash_bpt; }
}
```

Note that `$newhandlerid` is referring to the just executed `stop in` command.

6.9.7 Set Automatic Breakpoints for `dlopen` Objects

To have breakpoints in dlopened objects be managed automatically:

```
(dbx) when dlopen /home/myname/mylib.so {
      # delete old one if it exists.
      if [ -n "$B1" ]
      then   delete "$B1"
      fi
      # create new one
      stop in func
      B1="$newhandlerid"
}
```

6.9.8 Reset Application Files for `replay`

If your application processes files that need to be reset during a `replay`, you can write a handler to do that for you each time you run the program:

```
(dbx) when sync { sh regen ./database; }
(dbx) run < ./database
...                     # during which database gets clobbered
(dbx) save
...
(dbx) restore           # implies a RUN, which implies the SYNC event,
                        # which causes regen to run
```

6.9.9 Check Program Status

To see quickly where the program is while it's running:

```
(dbx) ignore sigint
(dbx) when sig sigint { where; cancel; }
```

Then type ^C to see a stack trace of the program without stopping it.

This is basically what the collector hand sample mode does (and more of course). Use SIGQUIT (^\) to interrupt the program because ^C is now used up.

6.9.10 Catch Floating Point Exceptions

To catch only specific floating-point exceptions, for example, IEEE underflow:

```
(dbx)ignore FPE                    # turn off default handler
(dbx) help signals | grep FPE      # can't remember the subcode name
...
(dbx) stop sig fpe FPE_FLTUND
...
```

Running, Stepping, Continuing 7

This chapter describes how to run, attach to, continue, and rerun a program in the Debugger; and how to single-step through lines of program code. The commands —Run (`run`, `rerun`), Next (`next`), Step (`step`), and Continue (`cont`)— are called *process control* commands. Used together with the event management commands (`stop`, `when`, `trace`) described in Chapter 5, "Setting Breakpoints and Traces" on page 5-63, and Chapter 6, "Event Management" on page 6-79, you can control the run-time behavior of a program executing under the Debugger.

This chapter is organized into the following sections:

Running a Program in the Debugger	*page 7-103*
Single-step Execution: next, step and call	*page 7-107*
Continuing a Program	*page 7-109*
Using Cntl-C to Stop a Process	*page 7-111*

7.1 Running a Program in the Debugger

When you first load a program into the Debugger, the Debugger visits the program's "main" block (`main()` for C and C++, `MAIN()` for FORTRAN and `program()` for Pascal). (See Section 1.3.3, "Interaction of dbx with the Dynamic Linker," on page 1-8 for more information on what happens at program startup.) The Debugger waits for you to issue further commands; you can visit code or use event management commands.

You may choose to set breakpoints in the program before running it in the Debugger. When ready, use the `run` command to start program execution.

7.1.1 Running a Program Without Arguments

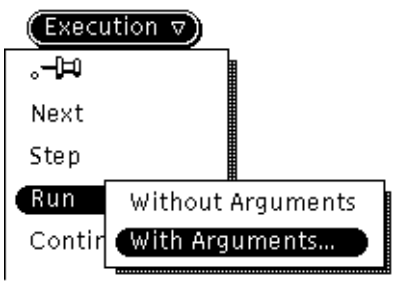
To run a program in the Debugger without arguments:

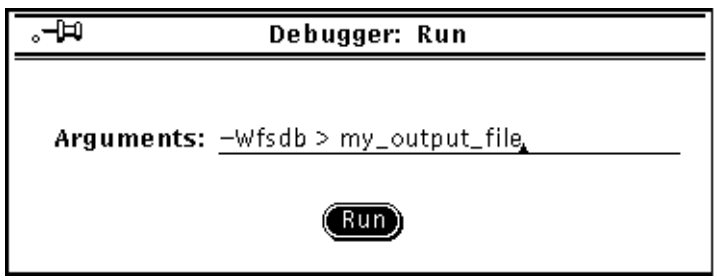
- **Choose Run from the Execution menu.**

7.1.2 Adding Arguments to the `Run` Command

To run a program with command line arguments:

- **Choose With Arguments from the Run pull-right menu to display the Run pop-up window; type in the arguments and click on Run. You can also use the run button in the Debugger base window.**





7.1.3 Redirecting Input and/or Output

You can redirect input and/or output to and from a specified file by specifying the input or output filenames in the Run With Arguments line pop-up window or in the command pane.

To redirect input and/or output to and from a file for the Run command,

- **Type the filename after any arguments, using the appropriate redirection operators:**

Redirect output only:

Arguments: -Wfsdb > my_output_file

Redirect input only:

Arguments: -Wfsdb < my_input_file

Here is the syntax for the corresponding Debugger command:

`run` [*arguments*] [`<` *input_file* | `>` *output_file*]

Note that output from the Run command overwrites an existing file even if you have set `noclobber` for the shell in which you are running the Debugger.

7.1.4 Rerunning a Program

To run a program again, use the Run command to restart execution:

- **Choose Run:No Arguments or Run:With Arguments from the Execution menu.**
  If you use Run With Arguments, Run uses whatever arguments are on the Argument text field line in the Run pop-up window.

  If you choose Run:No Arguments, the program does not use arguments entered previously in the current debugging session. (The dbx `rerun` command is the equivalent of Run:No Arguments. dbx "`run`" reuses previous arguments.)

7.1.5 Attaching the Debugger to a Running Process

You may need to debug a program that is already running. For example, you may want to debug a running server and you do not want to quit it; or a program may be looping indefinitely and you would like to examine it under the control of the Debugger without killing it. You can *attach* the Debugger to a running program by using the program's *pid* number as an argument to the `debug` command or to the Debugger start-up command.

Once you have debugged the program, you can then use the `detach` command to take the program out from under the control of the Debugger without terminating the process.

Note – If you quit the Debugger after having attached it to a running process, the Debugger implicitly detaches before terminating.

To attach the Debugger to a program that is running independently of the Debugger:

1. **Determine the** *process_ID* **of the program**
   One way is to use the `ps` utility,
2. **Then,**
   a. **If the Debugger is already running, type in the command pane:**
   `(debugger)` **`debug`** *program_name process_ID*

(Solaris 2.x) You can substitute a dash "-" for the *program_name*. If you use a "-" instead of the program_name, the Debugger automatically finds the program associated with the pid and loads it. Note that you cannot use `run` because the full pathname is not known.

   b. **If the Debugger is not running, start the Debugger with the** *process_ID* **(`pid`) number as an argument:**

`machine%` **`debugger`** *program_name process_ID*

(Solaris 2.x) You can substitute a dash "-" for the *program_name*, as with the `debug` command.

After you have attached the Debugger to a program, the program stops executing. You can examine it as you normally would any program loaded into the Debugger. You can use any event management or process control commands to debug it.

7.1.6 Detaching a Process from the Debugger

When you have finished debugging the program, use the `detach` command to detach the Debugger from the program. The program then resumes running independently of the Debugger.

To detach a process from running under the control of the Debugger:

- **Use the Debugger** `detach` **command**

`(debugger)` **`detach`**

7.2 *Single-step Execution:* `next`, `step` *and* `call`

The Debugger supports two basic single step commands: `next` and `step`, plus a variant of `step`, called `step up`. Both `next` and `step` cause the program to execute one source line.

If the line executed contains a function call, `next` executes that call and returns from it ("steps over" the call). `step` stops at the first line in a called function.

`step up` (available only as a dbx command) returns the program to the caller function after you have stepped into a function.

`pop` (available only as a dbx command) pops the top frame off the stack and adjusts the frame pointer and the stack pointer accordingly. The `pop` command also changes the program counter to the beginning of the source line. Type `help pop` in the command pane for more information.

7.2.1 *Issuing* Next *and* Step *Commands*

To single-step, stepping over functions:

- **Choose Next from the Execution Menu. You can also use the next button from the Debugger base window.**

The Program Location glyph advances and points to the new "next line."

To single step, stepping into functions:

- **Choose Step from the Execution menu. You can also use the step button from the Debugger base window.**

The source display shows the function and points at the first line.

Note that each time you single step to a line in a different function, the value of the Visiting Function information field changes automatically to match this new program location function.

To return to the caller after stepping into a function:

♦ **Type in the command pane**

```
(debugger) step up
```

Specifying a Number of Single-steps

Note that the Debugger commands `step` and `next` accept a number arguments that let you specify an exact number of source lines to execute before stopping. The default is 1.

To single step a specified number of lines of code:

♦ **Use the appropriate debugger command,** `next` **or** `step` **followed by the number of lines** `[`*n*`]` **of code you want executed before stopping:**

```
(debugger) step n
```

7.2.2 Calling a Function

When a program is stopped, you can call a function using the Debugger `call` command. The `call` command accepts values for the parameters that must be passed to the called function.

Note – By default, after every `call` command, the Debugger automatically calls `fflush(stdout)` to ensure that any information stored in the I/O buffer is printed. (A user may call a function *explicitly*, using the `call` command, or *implicitly*, either by evaluating an expression containing function calls, or using a conditional modifier (such as, "`stop in glyph -if animate()`"). To turn off automatic flushing, set Auto Flush control to Off. Auto Flush is located on the Debugging Environment Attributes Property Window. You can also use `dbxenv autoflush` or `$DBX_autoflush`.

To call a procedure from the Debugger shell:

♦ **Type the name of the function and supply its parameters:**

```
(debugger) call change_glyph(1,3)
```

Here is the syntax for `call`:

`call` *function_name* `([`*parameters*`])`

Notice that while the parameters are optional, you must type in the parentheses after the *function_name*, for example,

```
(debugger)   call type_vehicle()
```

You can include backquotes as well as `classname::funcname()` for static member functions and `object.funcname` for member functions using "object" as the "this" parameter.

Unless the called function contains a breakpoint, when you use `call`, the Debugger behaves "next-like," returning from the called function. However, if the program hits a breakpoint in the called function, the Debugger stops the program at the breakpoint. If you now issue a `Stack Trace` command, the stack trace shows that the call originated from the *Debugger* command level.

If the source file in which the function is defined was compiled with the `-g` flag, or if the prototype declaration is visible at the current scope, the Debugger checks the number and type of arguments and issues an error message if there is a mismatch.

Note – For C++, the Debugger handles default arguments and function overloading as well.

If the above two conditions are not met, the Debugger does *not* check the number of parameters. The parameters are simply pushed on the stack as given in the parameter list.

Automatic resolution of the C++ overloaded functions is done if possible. If any ambiguity remains (for example, functions not compiled with `-g`), the Overloaded pop-up window is displayed.

7.3 Continuing a Program

To begin program execution again from the point where a program is stopped,

- **Choose Continue from the Execution menu. You can also use the `cont` button or type `cont` in the command pane.**

Here is the syntax for the Debugger `cont` command:

```
cont [at line-number] [sig signal-number]
```

The next subsection explains the `at` option. See Chapter 17, "Working with System Signals", for an explanation of how to use the `sig` option.

7.3.1 Skipping Lines of Code with `cont at line`

The Debugger `cont` command has a variant, `cont at` *line_number*, which allows you to specify a line at which to resume program execution other than the current program location line.

Among other uses, `cont at` *line* allows you to skip over one or a few lines of code, that you know to be causing problems, without having to recompile.

To continue a program at a specified line:

- Enter the `cont at` *n* command in the command pane. For example,

```
(debugger) cont at 124
```

Note that the source line number specified is evaluated relative to the file in which the program is stopped; the line number given must be within the scope of the function.

7.3.2 *Using* `cont at <line>` *with the* `assign` *Command*

One useful application of `cont at` is in conjunction with an `assign` command. Using `cont at` *line* with `assign`, you can avoid executing a line of code that contains a call to a function that may be incorrectly computing the value of some variable.

To resume program execution at a specific line:

1. **Use** `assign` **to give the variable a correct value.**
2. **Use** `cont at` *line* **to skip the line that contains the function call that would have computed the value incorrectly.**

Here is an example. Assume that a program is stopped at line 123. Line 123 calls a function, `how_fast()`, that computes incorrectly a variable, `speed`. You know what the value of speed should be, so you can assign a value to `speed`. Then continue program execution at line 124, skipping the call to `how_fast()`.

```
(debugger) assign speed = 180; cont at 124;
```

If you use this command with a `when` breakpoint command, the program skips the call to `how_fast()` each time the program attempts to execute line 123.

```
(debugger) when at 123 { >/dev/null assign speed = 180; cont at
124;}
```

7.4 *Using* `Cntl-C` *to Stop a Process*

You can stop a process running in the Debugger using `cntl-C (^C)`. When you stop a process using `^C`, the Debugger ignores the `^C`, but the child process sees it as a `SIGINT` and stops. You can then inspect the process as if it had been stopped by a breakpoint.

7.4.1 *Generic Case*

To stop a running process (after having chosen Continue or Run):

- **Type** `^C` **in the command pane or the PIO window.**

7.4.2 If the Process was Attached While It was Running

To stop a process that was attached to the Debugger while it was running:

- **Type `^c` either in the window in which the process is running or in the Debugger command pane.**

7.4.3 Continuing Again, After Stopping Using `^c`

To resume execution after stopping a program with `^c`:

- **Choose Continue from the Execution menu (or use Debugger `cont`)** You do not need to use the `cont` optional modifier, `sig` *signal_name*, to resume execution. The `cont` button (or `cont` command) resumes the child process after cancelling the pending signal.

Saving and Restoring a Debugging Run 8

This chapter describes how to save all or part of a debugging run and replay it later. The Debugger provides three commands for handling this functionality:

- `save [-`*number*`] [`*file*`]`
- `restore [`*file*`]`
- `replay [–`*number*`]`

The Replay item on the Execution menu is the same as the default Debugger command, `replay -1`. By saving and restoring a run except for the last debugging command entered, Replay undoes the last action.

This chapter is organized into the following sections:

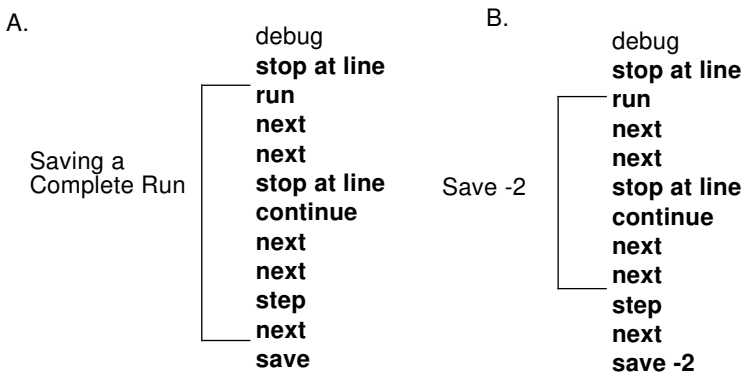
Saving a Debugging Run	*page 8-113*
Restoring a Saved Run	*page 8-115*
Saving and Restoring with the replay Command	*page 8-116*

8.1 Saving a Debugging Run

The `save` command saves to a file all of the debugging commands issued from the last `run`, `rerun`, or `debug` command up to the `save` command. This segment of a debugging session is called a *debugging run*. (Note that when you load a program into the Debugger, the Debugger itself issues a `debug` command, thereby starting the first debugging run.)

The `save` command saves more than the list of debugging commands issued. It also saves debugging information associated with the state of the program at the start of the run — information about breakpoints, display lists, and the like. If you want to restore a saved run, the Debugger uses the information in the save-file.

You can save part of a debugging run; that is, the whole run minus a specified number of commands from the last one entered. Example A shows a complete saved run. Example B shows the same run saved, minus the last two steps:



If you are not sure where you want to end the run you are saving, use the `history` command to see a list of the debugging commands issued from the top of the session. The default for the history record is 15 commands. For use with the save and restore facility, you may want to set the number of commands recorded higher. Type `help history` in the command pane for history options.

To save all of a debugging run up to the `save` command:

♦ **Type in the command pane,**

```
(debugger) save
```

To save part of a debugging run

♦ **Type in the command pane,**

```
(debugger) save –number
```

where *number* is the number of commands back from the save command that you do *not* want saved.

Saving a Series of Debugging Runs as Checkpoints

If you save a debugging run without specifying a *filename*, the Debugger writes the information to a special save-file. Each time you save, the Debugger overwrites this save-file. However, by giving the `save` command a *file_name* argument, you can save a debugging run to a file that you can restore later, even if you have saved other debugging runs since the one saved to *file_name*.

Using the *file_name* argument, you can save a series of debugging runs. Saving a series of runs gives you a set of *checkpoints*, each one starting farther back in the session. You can restore any one of these saved runs, continue, then reset the debugger back to the program location and state saved in an earlier run.

To save a debugging run to a file other than the default save-file:

♦ **Type in the command pane, use the** *file_name* **argument,**

```
(debugger) save file_name
```

8.2 Restoring a Saved Run

After saving a run, you can then restore the run using the `restore` command. The Debugger uses the information in the save-file. When you restore a run, the Debugger first resets the internal state to how it was at the start of the run, *then* it reissues each of the debugging commands in the saved run.

Note – The `source` command also reissues a set of commands stored in a file, but it does not reset the state of the Debugger; it merely reissues the list of commands from the current program location.

Prerequisites for An Exact Restoration of a Saved Run

For exact restoration of a saved debugging run, *all* of the inputs to the run must be exactly the same: arguments to a `run`-type command, manual inputs, and file inputs.

Note – If you save a segment and then issue a `run`, `rerun`, or `debug` command *before* you do a `restore`, `restore` uses the arguments to the *second*, post-save `run`, `rerun`, or `debug` command. If those arguments are different, you may not get an exact restoration.

To restore a saved debugging run:

- **Type in the command pane,**

```
(debugger) restore
```

To restore a debugging run saved to a file other than the default save-file:

- **Type in the command pane,**

```
(debugger) restore file_name
```

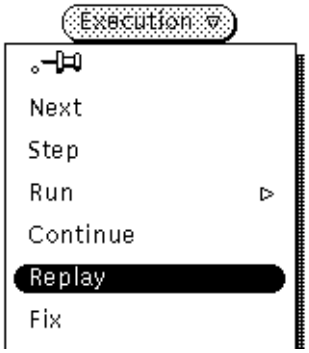
where *file_name* is the name of the file to which you saved the debugging run.

8.3 Saving and Restoring with the `replay` Command

The `replay` command is a combination command, equivalent to issuing a `save -1` followed immediately by a `restore`. The `replay` command takes a negative *number_of_ commands* argument, which it passes to the `save` portion of the command. By default, the value of *-number* is `-1`, so `replay` (and the Replay item on the Execution menu) works effectively as an "undo" command, restoring the last run up until (but not including the last command issued before the `replay` command).

To replay the current debugging run, minus the last debugging command issued:

♦ **Choose Replay from the Execution menu**



Replay — Replay is the same as the default Debugger line command, `replay`. Replay saves and restores the current run up to the last command issued prior to choosing Replay.

To replay the current debugging run and stop the run before the second to last command, use the Debugger `replay` command:

♦ **Type in the command pane,**

`(debugger)` **`replay`** **`–`***number*

where *number* is the number of commands back from the last debugging command.

8

Examining the Call Stack 9

This chapter describes how to examine the *call stack*, how to use the Stack Inspector, and how to debug a core file with the `where` command or the stack menu. The call stack represents all currently active routines — that is, routines that have been called but have not yet returned to their respective caller. The Stack Inspector performs operations on the stack and dynamically updates the stack when you visit another process or thread.

You can use the Stack Inspector in conjunction with dbx commands to manipulate the frame you are visiting. Use `dump` to show variables local to the frame. You use the `print` command or the Visual Data Inspector to examine the value of a variable, and you can use the `assign` command to change the value of a local variable.

This chapter is organized into the following sections:

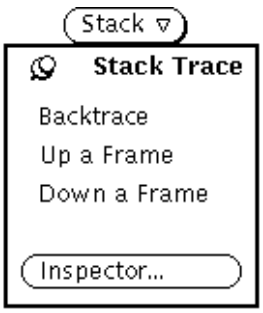
Examining the Stack	*page 9-119*
The Stack Inspector	*page 9-122*
Debugging a Core File	*page 9-125*

9.1 Examining the Stack

Because the call stack grows from higher memory to lower memory, *up* means going up the memory to the caller's frame and *down* means going down the memory. The current program location (the routine that was currently

executing when the program stopped at a breakpoint, after a single-step, or when the program faults, producing a core file) is in higher memory, while a caller routine, such as `main()`, is located lower in memory.

The Stack menu in the Debugger contains four items for examining the stack: Backtrace, Up a Frame, Down a Frame, and Inspector.



Backtrace — shows the call stack for your current process. Backtrace is equivalent to the `where` command.
Up a Frame — moves up one frame in the stack. Up a Frame is equivalent to the `up` command.
Down a Frame — moves down one frame in the stack. Down a Frame is equivalent to the `down` command.
Inspector — displays the Stack Inspector window.

To see all options to the `where` command,

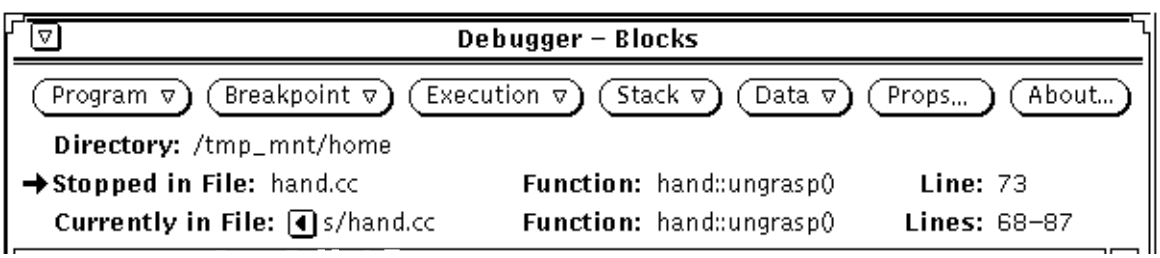
- **Type** `help where` **in the command pane.**

9.1.1 Walking the Stack and Returning Home

Moving up or down the stack is sometimes referred to as "walking the stack." When you visit a function by moving up or down the stack, the Debugger displays the function in the source display, marking it with a hollow arrow.

The location you start from, *home*, is the point where the program stopped executing and is indicated in the source display by a solid arrow. From home, you can move up or down the stack using the `up`, `down`, or `frame` commands.

To return "home," that is, to redisplay the function the program is stopped at,



- **Move the cursor over the arrow, which is to the left of the Stopped in File field in the Information fields area of the main Debugger window, and click SELECT.**
  You may want to create a Current Focus command button to do this operation automatically through a command.

The Debugger commands `up [`*number*`]` and `down [`*number*`]` both accept a *number* argument that instructs the Debugger to change the current function some *number* of frames up or down the stack from the current frame. The `-h` option includes all hidden frames in the count.

`up` [`-h`] [*number*]

Move up the call stack (towards `main`) *n* levels. If *number* is not specified, the default is one. This command allows you to examine the local variables in functions other than the current one.

`down` [`-h`] [*number*]

Move down the call stack (towards the current stopping point) *n* levels. If *n* is not specified, the default is one.

The `frame` command is similar to the `up` and `down` commands. It allows you to go directly to the frame as given by numbers printed by the `where` command. This command is available from the command line only.

`frame`
`frame` [`-h`] *number*
`frame` [`-h`] +[*number*]
`frame` [`-h`] -[*number*]

The `frame` command without an argument prints the current frame number. With *number*, the command allows you to go directly to the frame indicated by the number. By including a "+" or "-", the command allows you to move an increment of one level up (+) or down (-). If you include a positive or negative option with a number option, the command allows you to move up or down the specified number of levels. Including the `-h` option includes any hidden frames in the count.

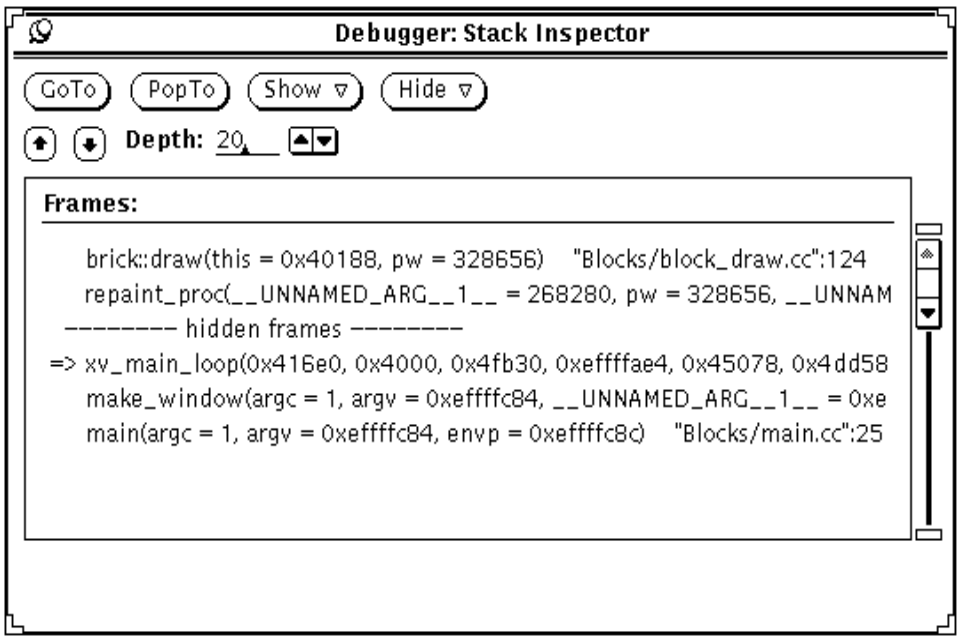
9.2 The Stack Inspector

Use the items shown in the Stack Inspector window to view the stack. The Debugger displays the call stack in this window and dynamically updates it when you visit another process or thread.

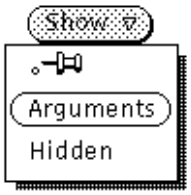
The window contains two command buttons, two menu buttons, up and down arrows that move the cursor in the Frames pane and the Depth field, which specifies the number of stack frames currently displayed in the Frames pane.

The GoTo button moves the arrow in the Frames pane to the selected frame. You select a frame by moving the cursor over the desired frame and clicking SELECT.

The PopTo command button is similar to the GoTo command button except that dbx unwinds the stack to the selected frame (or top frame if no frame is selected), and moves the Program Counter (PC) so execution resumes at the selected function.



9.2.1 The Stack Inspector Show Menu



Use the items listed in the Show menu to obtain information about the stack. The menu contains two items that toggle. Arguments is the default menu item.

Arguments — displays the following information:

- Function arguments
- Name of the source file and line number for the function, if compiled with the `-g` option

If no debugging information is available for a function on the call stack, the expanded information shows the:

- Function arguments (hexadecimal values of "in registers" `$i0` through `$i5`)
- Memory address of the function

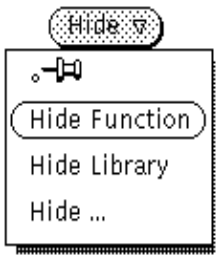
The command toggles to No Arguments.

No Arguments — displays a list of function names only (default).

Note – Obtaining full stack frame information requires more debugging information and slows down the Stack Inspector. Because No Arguments provides you with a list of function names only, the response time is much quicker than choosing Arguments.

Hidden — toggles between displaying all functions hidden by the Hide Function command in the Hide menu and re-hiding them.

9.2.2 The Stack Inspector Hide Menu



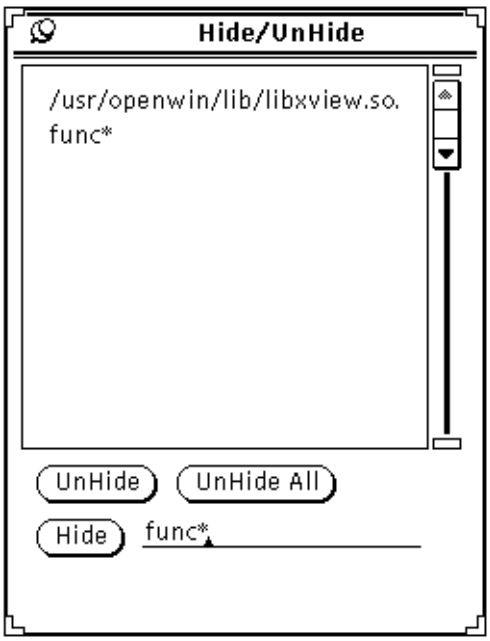
Hide Function — hides a selected function. This command hides functions by name. That is, all functions with the same name as the selected function are hidden.
Hide Library — hides all functions from the library that contains the selected function.
Hide — displays the Hide/Unhide window described in the next section.

Use the items listed in the Hide menu to hide stack inspector entries. Hide Function is the default menu item.

A placeholder in the Frames pane indicates that one or more functions are hidden.

Note – The top frame (frame number 1) and the first frame in a hidden library block are never hidden.

9.2.3 The Hide/Unhide Menu



Hide — adds the expression typed in the text field to the hidden list.
Unhide — deletes the selected expressions from the hidden list.
Unhide All — deletes all expressions from the hidden list.

Use the Hide/Unhide Menu to display a list of stack inspector *filters*. Filters are regular expressions that are matched to function names or library paths.

You can selectively hide or show functions by selecting the appropriate filter and choosing either the Hide Function or Hide Library command from the Hide menu.

9.3 Debugging a Core File

The `where` command or Stack menu is especially useful for learning about the state of a program that has crashed and produced a core file. When a program crashes and produces a core file, you can load the core file into the Debugger.

To load a core file into the Debugger:

- **Type in the command pane,**

```
(debugger) debug program_name corefile
```

If the Debugger is not already running, you can load a core file into the Debugger when you start the session:

♦ **In a Command or Shell tool, type**

```
% debugger program_name corefile
```

Once the core file is loaded in the Debugger, choose Backtrace from the Stack menu to examine the call stack at the time the program faulted. The Debugger `where` command is the command pane equivalent of the Backtrace menu item command.

When you debug a core file, you can also evaluate variables and expressions to see what values they had at the time the program crashed, but you cannot evaluate expressions that make function calls.

Evaluating and Displaying Data 10

This chapter describes how to evaluate data, display the value of expressions, variables, and other data structures, and how to assign a value to an expression.

In the Debugger you can perform three types of data checking:

- Evaluate data (print)
  Spot-check the value of an expression by using the Evaluate/Print commands
- Display data (display)
  Monitor the value of an expression each time the program stops by using the Display command

(Solaris 2.x)

- Inspect data (display plus relations)
  Examine program variables including complex structures by using Data Inspector. This feature is described in Chapter 11, "Visual Data Inspector."

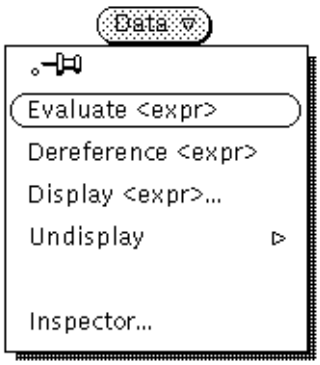
This chapter is organized into the following sections:

Evaluating and Dereferencing Variables and Expressions	*page 10-128*
Monitoring Expressions	*page 10-132*
Assigning a Value to an Expression	*page 10-134*
Evaluating Arrays	*page 10-134*

In addition to printing evaluations in the command pane, the Debugger features a separate Data Display window for monitoring data.

The Data Menu

You can perform most evaluation and display operations using the items on the Data menu.



Evaluate — displays the value of the selected expression Same as the print button.

Dereference — displays the value to which the selected pointer is a reference. Same as the print* button.

Display — displays the Data Display window, where the Debugger monitors the value of the selected expression. The Debugger updates the information automatically until you issue the Undisplay command. Same as the display button.

Undisplay — instructs the Debugger to stop monitoring the selected expression(s).

(Solaris 2.x) **Inspector —** displays the Data Inspector window. This feature is described in Chapter 11, "Visual Data Inspector."

10.1 Evaluating and Dereferencing Variables and Expressions

This section shows how to evaluate variables and expressions or dereference pointers to them.

If you evaluate a variable name that is defined within the scope of the *current function* (value of Currently in Function and of `func`), then the Debugger uses that variable. Make sure the variable the Debugger evaluates is in the function you think it is.

Verifying Which Variable the Debugger will Use

If you are not sure which variable the Debugger will evaluate, use the `which` command to see the fully qualified name the Debugger will use. (For details, see Section 4.5.2, "Seeing a Preview of Which Symbol the Debugger Will Use," on page 4-56.)

To see other functions and files in which a variable name is defined, use the `whereis` command. For details, see "Locating Symbols: the whereis and which Commands" on page 4-55.

Variables Outside the Scope of the Current Function

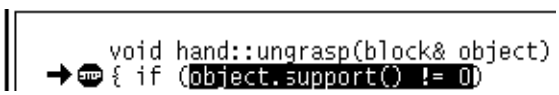
When you want to evaluate (or monitor) a variable that is outside the scope of the current function:

- Qualify the name of the function.
  See "Qualifying Symbols with Scope Resolution Operators" on page 4-52.
- Visit the Function; that is, change current function.
  See "Visiting Code" on page 4-47 for a description of how to change the current function.

10.1.1 Printing the Value of a Variable or an Expression

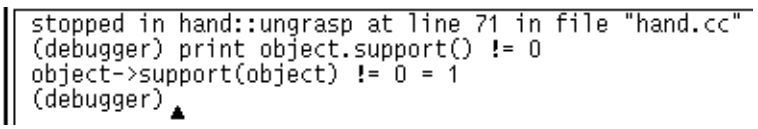
To evaluate a variable or expression:

1. **Select the target variable or expression in the source display.**



2. **Choose Evaluate from the Data menu.**
   Evaluate is the default item, so you can click on the Data menu button without displaying the menu. You can also click the Print command button.

The Debugger echoes the command and prints the evaluation in the command pane:



Evaluating Pascal Character Strings

To evaluate a Pascal character string in the Debugger, use the double quote ("") syntax to identify the string as a character string (as distinct from a character constant). This Debugger convention also applies to passing parameters when calling a Pascal function.

For example, to evaluate the Pascal character string, `abc`:

```
(debugger) print "abc"            //not print 'abc'
```

C++ Printing

You can use the commands `print` or `display` with a `-r` (recursive) option. The Debugger displays all the data members directly defined by a class and those inherited from a base class.

Evaluating Unnamed Arguments in C++ Programs

C++ allows you to define functions with unnamed arguments. For example:

```
void tester(int)
{
};

main(int, char **)
{
    tester(1);
};
```

Though you cannot use unnamed arguments elsewhere in a program, the Debugger encodes unnamed arguments in a form that allows you to evaluate them. The form is:

__UNNAMED_ARGUMENT_%n__
where the Debugger assigns an integer to %n.

To obtain an assigned argument name from the Debugger, issue the whatis command with the function name as its target:

```
(debugger) whatis tester
void tester(int _UNNAMED_ARGUMENT_0_);
(debugger) whatis main
int main(int _UNNAMED_ARGUMENT_1_, char **_UNNAMED_
ARGUMENT_2_);
```

To evaluate (or display) an unnamed function argument,

- **Use the encoded name as the argument to the** print **(or** display**) command.**
For example:

```
(debugger) print _UNNAMED_ARGUMENT_1_
_UNNAMED_ARGUMENT_1_ = 4
```

10.1.2 Dereferencing Pointers

When you dereference a pointer, you ask for the value stored in the container to which the pointer is a reference, not for the value of the pointer-variable.

To dereference a pointer:

1. **Select the indirection operator (*) together with the variable name:**

```
struct s {
 int a;
} s, *t;
```

2. Choose Dereference from the Data Menu.
You can also click the Print* button.

The Debugger echoes the command and prints the evaluation in the command pane; in this case, the value pointed to by t:

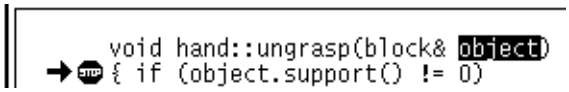
```
(debugger) print *t
*t = {
a = 4
}
(debugger)
```

10.2 Monitoring Expressions

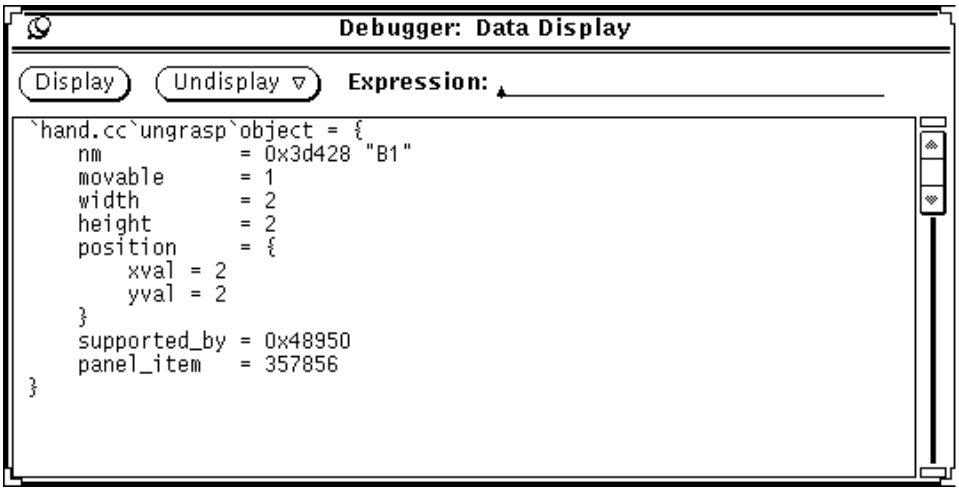
Monitoring the value of an expression each time the program stops is an effective technique for learning how and when a particular expression or variable changes. Display instructs the Debugger to monitor one or more specified expressions or variables. Monitoring continues until you turn it off with the Undisplay command. The Display command displays a separate Data Display pop-up window in which the Debugger displays the values of specified expressions.

To display the value of a variable or expression each time the program stops:

1. **Select the variable or expression in the source display:**
   For example,



2. **Choose Display from the Data menu to display the Data Display window, showing the values of the selected variable.**
   In this example, `object` is a reference to an instantiation of a block.



Now, each time the program stops, the Data Display window reports the value of the variable. The Data Display window has a text field labeled Expression; use this text field to enter the name of an additional expression you want to monitor. Pressing the Return key after the last character entered issues the new Display command.

You can monitor more than one variable at a time. The Data Display window displays each of the expressions you are currently viewing. (Note that the window has a scrollbar.)

10.2.1 Turning Off Display (Undisplay)

The Debugger continues to display the value of a variable you are monitoring until you turn off Display with the Undisplay command. You can turn off the display of a specified expression or turn off the display of all expressions currently being monitored.

To turn off the display of a particular variable or expression:

1. **Select the variable in either the Data Display window or the source display.**

2. **Click the Undisplay button in the Data Display window or choose Undisplay from the Data menu.**

To turn off the display of all currently monitored variables:

♦ **Type in the command pane,**

```
(debugger) undisplay 0
```

10.3 Assigning a Value to an Expression

To assign a value to a variable:

♦ **Type in the command pane**

```
(debugger) assign variable = expression
```

10.4 Evaluating Arrays

You evaluate arrays the same way you evaluate other types of variables. First, you select the array, then choose Evaluate from the Data menu. (See Chapter 11, "Visual Data Inspector", for more information on inspecting arrays.)

Here is an example, using a FORTRAN array:

```
integer*4 arr(1:6, 4:7)
```

To evaluate the array:

♦ **Type in the command pane the print command followed by the array.** Here is an example using a FORTRAN array:

```
(debugger) print arr(2,4)
```

Note – Memory limitations internal to the Debugger prevent the display of large arrays. While the Debugger now supports FORTRAN *array slicing* (see the following section), it does not yet offer this support for other languages.

10.4.1 Array Slicing for FORTRAN Arrays

For FORTRAN arrays, the Debugger `print` command allows you to evaluate part of a large array. FORTRAN array evaluation includes:

- *Array Slicing*

  Print any rectangular, *n*-dimensional box of a multi-dimensional array.

- *Array Striding*

  Print certain elements only, in a fixed pattern, within the specified slice (which may be an entire array).

You can slice an array, with or without striding (the default stride value is 1, which means print each element).

10.4.2 Syntax for FORTRAN Array Slicing and Striding

For *each* dimension of an array, the full syntax to the `print` command to slice the array is:

```
(debugger) print arr(exp1:exp2:exp3)
```

where:

exp1 = start_of_slice
exp2 = end_of_slice
exp3 = length_of_stride (the number of elements skipped is *exp3* – 1)

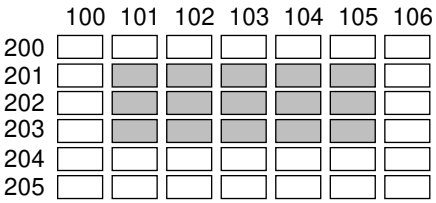
For an *n*-dimensional slice, separate the definition of each slice with a comma:

```
(debugger) print arr(exp1:exp2:exp3, exp1:exp2:exp3,...)
```

Slices

Here is an example of a two-dimensional, rectangular slice, with the default stride of 1 omitted:

```
print arr(201:203, 101:105)
```



This command prints a block of elements in a large array. Note that the commands omits exp3, using the default stride value of 1.

The first two expressions (`201:203`) specify a slice in the first dimension of this two-dimensional array (the three-row column). The slice starts at the row `201` and ends with `203`.The second set of expressions, separated by a comma from the first, defines the slice for the 2nd dimension. The slice begins at column `101` and ends after column `105`.

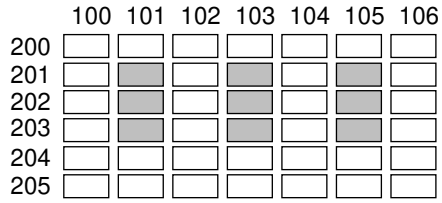
Strides

When you instruct `print` to *stride* across a slice of an array, the Debugger evaluates certain elements in the slice only, skipping over a fixed number of elements between each one it evaluates.

The third expression in the array slicing syntax, (*exp3*), specifies the length of the stride. The value of *exp3* specifies the elements to print; the number of elements skipped is equal to *exp3* −1. The default stride value is 1, meaning: evaluate all of the elements in the specified slices.

Here is the same array used in the previous example of a slice; this time the print command includes a stride of 2 for the slice in the second dimension.

```
print arr(201:203, 101:105:2)
```



A stride of 2 prints every 2nd element, skipping every other element.

Shorthand Syntax

For any expression you omit, `print` takes a default value equal to the declared size of the array. Here are examples showing how to use the shorthand syntax.

For a one-dimensional array:

`print arr`	Prints entire array, default boundaries.
`print arr(:)`	Prints entire array, default boundaries and default stride of 1.
`print arr(::`*exp3*`)`	Prints the whole array with a stride of *exp3*.

For a two-dimensional array:

`print arr`	Prints the entire array.
`print arr (:,::3)`	Prints every third element in the second dimension of a two-dimensional array.

10

Visual Data Inspector 11

This feature is available only on Solaris 2.x**.**

This chapter describes the Visual Data Inspector (VDI), a new Debugger component that extends the data display and monitoring capability described in Chapter 10, “Evaluating and Displaying Data.” VDI shows pointer/reference relationships among data as well as the current data values.

For information on evaluating or displaying the value of expressions, variables, and other data structures, see Chapter 10, “Evaluating and Displaying Data.”

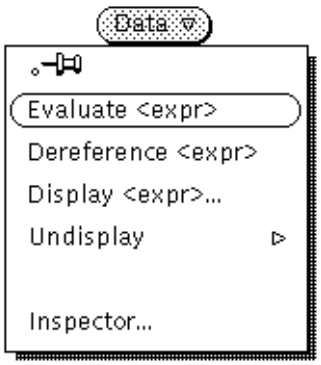
This chapter is organized into the following sections:

Starting the Visual Data Inspector (VDI)	*page 11-139*
The VDI Window	*page 11-140*
Examining Variables and Complex Structures	*page 11-145*

11.1 Starting the Visual Data Inspector (VDI)

♦ **Choose Inspector from the Data menu**

You start the VDI from the Data menu on the Debugger base window. Data menu options Evaluate, Dereference, Display, and Undisplay are described in Chapter 10, “Evaluating and Displaying Data.”



Evaluate — displays the value of the selected expression. (Same as the print button.)

Dereference — displays the value to which the selected pointer is a reference in the command pane. (Same as the print* button.)

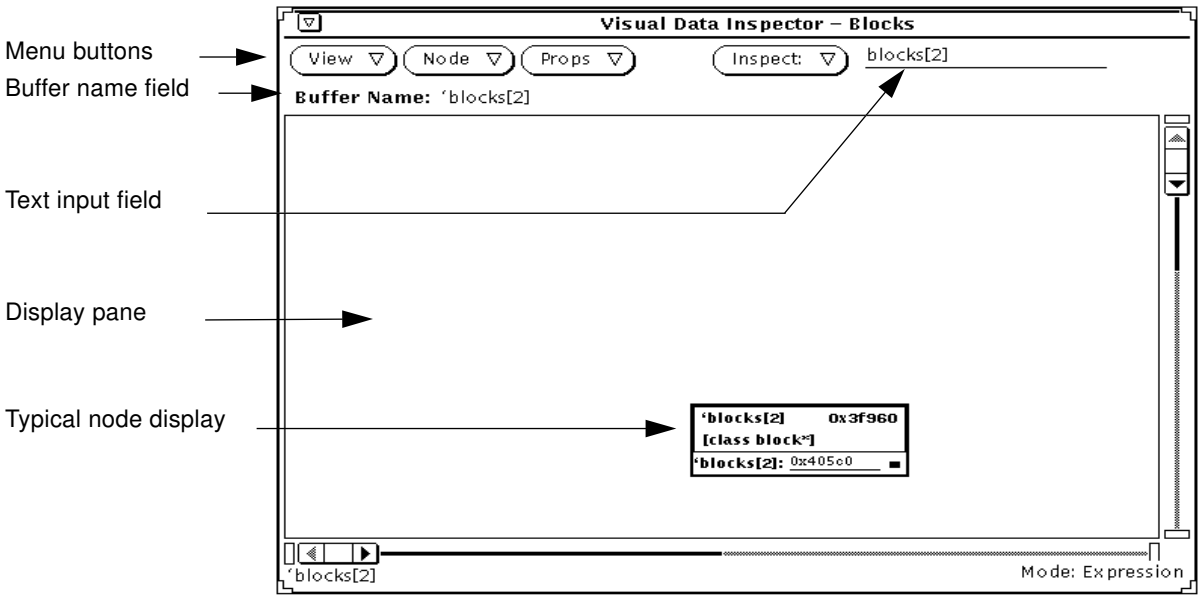
Display — displays the Data Display window, where the Debugger monitors the value of the selected expression.

Undisplay — instructs the Debugger to stop monitoring the selected expression.

Inspector — displays the Visual Data Inspector window, where the Debugger monitors and graphically displays program variables and data structures.

11.2 The VDI Window

The VDI window consists of a row of buttons, text-input field, buffer name field, graphical display pane, and a footer message area. The display pane displays nodes and pointer lines that connect nodes. A display node is a graphical display of an expression that you entered in the text input field or from the command line. You use the scrollbars to navigate in the display pane.



The window header shows the name of the executable program being inspected. The left footer shows the fully qualified name of the currently selected node. The node name may have been truncated in the display node header (as indicated by a "<" on the left of the name field). The right footer shows the VDI Inspect mode. (See Section 11.2.4, "The Inspect Menu," on page 11-145.)

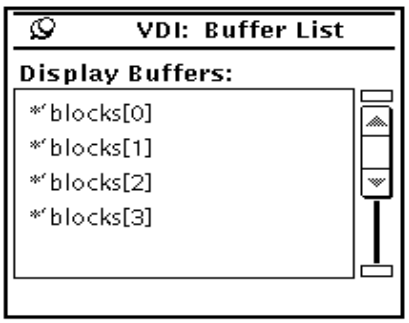
Nodes can be grouped in *buffers*. You create, select, delete, and arrange buffers in the display pane from the View menu.

11.2.1 The View Menu

You control the display of buffer containers and nodes from the View menu.

Next Buffer — switches to the next available buffer.
Previous Buffer — switches to the available buffer opened before the current buffer.
List Buffers — displays the Buffer List window that shows all buffers by name.
New Buffer — creates an empty display pane.
Delete Buffer — deletes the viewed buffer.
Layout — arranges the display nodes in the buffer.

Here is a typical VDI Buffer List window.

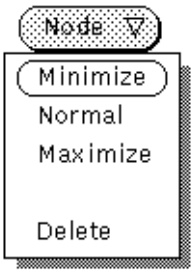


The VDI Buffer List window contains a list of open buffers.

♦ **Click on the buffer name to select a buffer.**

11.2.2 The Node Menu

You select how nodes are displayed from the node menu. You can select only one node at a time. The selected node is changed by the node option.



Minimize — place the selected node into an icon that remains in the same graphical position in the pane. This is useful when viewing multiple nodes.

Normal — place the selected node into the standard graphical form which displays up to five fields. You can change the number of fields displayed by setting the X resource `vdi*displayCount:<int>` to the desired number.

Maximize — expands the selected node to display all fields. Maximize has no effect on nodes with fewer fields than are displayed by default.

Delete — removes the selected node from the display pane.

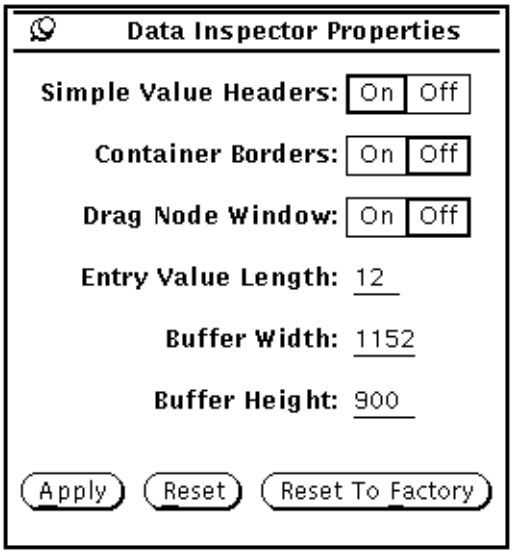
11.2.3 The VDI Props Menu

You use the Props menu to change the Visual Data Inspector environment and to display the current version of the Debugger.

Properties — displays the VDI Properties window.
Version — displays the current version of the Debugger in the left footer of the VDI window.

The VDI Properties Window

You use the Properties window to change the Visual Data Inspector properties.



Simple Value Headers — select On to display header information for each node. Select Off to display only a single value and name.

Container Borders — select On to display borders around structures that are nested. Select Off to remove borders around the containers.

Drag Node Window — select On to drag nodes as solid objects. Select Off to drag node outlines.

Entry Value Length — specify the number of visible characters displayed in the node's data fields. This value only applies to newly created nodes. To change an existing length, the node must first be deleted, then inspected again.

Buffer Width and Buffer Height — allow the buffer dimensions to be increased beyond the default screen size. This attribute applies to the current buffer and buffers created after the new values are applied.

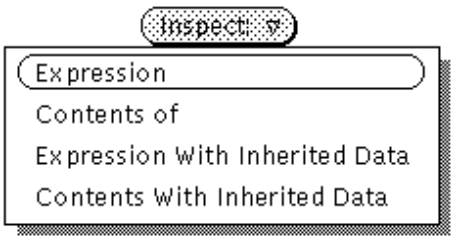
Apply — apply the changes, if any.

Reset — restore the options to their original states before any properties were modified.

Reset To Factory — restore the settings to their original state as the factory defaults.

11.2.4 The Inspect Menu

Selection of these menu items sets the current mode for evaluating program variables. The current mode is displayed in the right footer area. This mode is persistent until a new option is chosen.



Expression — inspect the expression exactly as requested.

Contents of — inspect the contents of a pointer (dereference the pointer).

Expression With Inherited Data — for a C++ object, this is equivalent to the `print -r` option. This displays all data members of the object, including inherited data members.

Contents With Inherited Data — inspect the contents of a pointer to a C++ object (dereference the pointer), showing all data members of the object, including inherited data members.

11.3 Examining Variables and Complex Structures

This section describes how to select and display program variables and complex data structures.

11.3.1 Starting an Inspection Session

You start a Visual Data Inspector session by preparing the Debugger:

- Load the program containing the target variable into the Debugger.
- Select the target variable in the source display.

- Set a breakpoint in the source display.

You can set a breakpoint by clicking on the stop at button and then the run button in the Debugger base window. The program loaded in the Debugger runs and then stops at the set breakpoint.

This example from the source display shows a selected variable and set breakpoint used in this chapter.

```
    int x = 0;
➔ blocks[0] = the_table;
    blocks[1] = new brick("B1", 2, 2, point(x, 0), the_table); x += 2;
    blocks[2] = new brick("B2", 2, 2, point(x, 0), the_table); x += 2;
    blocks[3] = new brick("B3", 4, 4, point(x, 0), the_table); x += 4;
    blocks[4] = new brick("B4", 2, 2, point(x, 0), the_table); x += 2;
    blocks[5] = new wedge("W5", 2, 4, point(x, 0), the_table); x += 2;
    blocks[6] = new brick("B6", 4, 2, point(x, 0), the_table); x += 4;
    blocks[7] = new wedge("W7", 2, 2, point(x, 0), the_table); x += 2;
    blocks[8] = new ball ("L8", 2, 2, point(x, 0), the_table); x += 2;

    make_window (argc, argv, envp); }
```

♦ **Choose Inspector from the Data menu to start the VDI.**

Enter the name of the variable to be inspected into the text input field. Note that multiple expressions in the text field are not allowed. You can enter the name in one of three ways:

- Type the variable name in the Inspect text input field.
- Use the mouse to select the variable in the Debugger source display and drag the name to the text input field.
- Use the mouse to select the variable in the source display and use the keyboard keys Copy and Paste to enter the variable name in the text input field.

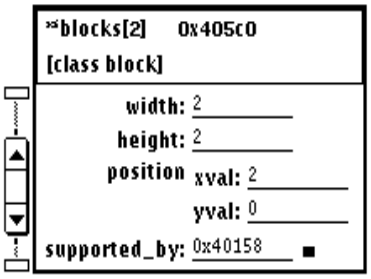
♦ **You then click on the Inspect button or use carriage return to start the inspection. A stopwatch is displayed while inspection is in progress.**

♦ **Alternatively, type** `inspect` *`variable`* **in the command pane.**

The selected variable is displayed in the VDI display pane as a graphical node. More than one variable can be inspected in each buffer.

11.3.2 The VDI Node

The VDI displays nodes in the display pane. The `*`blocks[2]` node is shown in a scrolled position to display the pointer and Pointer glyph.



***`blocks[2]** — variable being inspected. The single quote (`) indicates a global variable.

0x405c0 — address of the variable.

[class block] — data type of the node.

Fields and values — up to five fields (default) are displayed in Normal view.

■ **Pointer-glyph** — single click here to view connecting nodes. This is used to follow pointers.

Scrollbar — shown in Normal view when the node selected has more than five fields (default setting).

11.3.3 The VDI Display Pane

You can display nodes in three formats (see Figure 11-1):

- **Minimize** — displays node `*`blocks[2]` as an icon.
- **Normal** — displays node `*`blocks[2]` in the Normal format. For the default display count, up to five node fields can be displayed without scrolling. The floating scrollbar is displayed with a selected node that has more fields than the current display count resource.
- **Maximize** — displays node `*`blocks[0]` in the maximize option. All node fields are displayed in the maximize format. Use the scrollbar if the list of fields exceeds the display pane space.

This illustration shows multiple nodes and the three display formats.

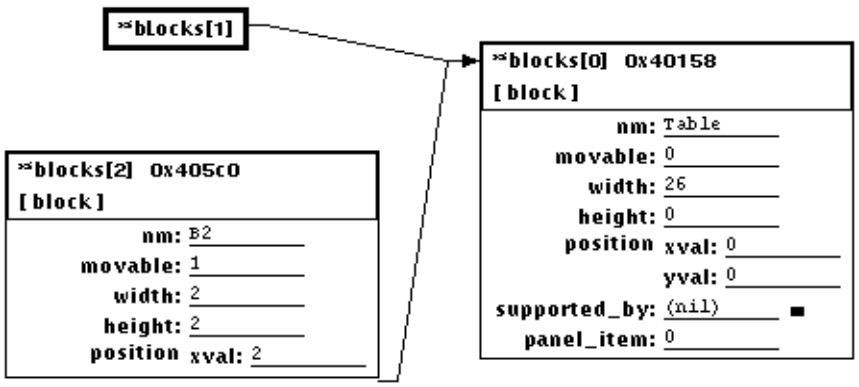


Figure 11-1 Display Pane Node Formats

11.3.4 Array Inspection

The VDI displays up to two-dimensional arrays of data in a form similar to a spreadsheet. If the size of the array is larger than the viewable area, horizontal and/or vertical scrollbars are provided. If the array dimensions are greater than two, the array is displayed as **<** `nD array`**>**, where *n* is the number of dimensions. Individual array data can be inspected in the normal manner.

Displayed Elements in an Array

The number of displayed elements in an array are set at creation time. If the number of elements exceeds the specified limit, a scrollbar is provided to scroll the array contents. The maximize option in the Node menu does not function on arrays; however, arrays can be minimized and reset to normal.

Limitations

1. Display of arrays with dimensions greater than two is of the form **<**`n`*D* `array`**>**.

2. There is a dbx limit to the maximum number of array elements that can be inspected (at publication time, the limit is 1000).

3. Arrays of pointers, arrays of pointers to pointers, and similar forms generated via `malloc` show only the first array address when inspected. This is a dbx limitation.

4. Arrays of pointers do not support pointer following. That is, no lines are drawn from internal array elements to external nodes in the display.

5. Display of arrays inside of a structure are presented as **<***n*`D array`**>**, similar to limitation (1).

6. You cannot rename a buffer.

Assignment and array element updating are supported. Editing the value of a displayed array element also updates the value in the Debugger. Editing the value of an array element in the Debugger also updates the value in the displayed array element.

The display of array nodes is similar to the display of structure nodes. They can be moved, deleted and changed to an icon. A node can be in multiple buffers.

11

Process/Thread Inspector 12

This feature is available only on Solaris 2.x.

This chapter describes how to find information about threads by using the Process/Thread Inspector.

This chapter is organized into the following sections:

Introduction to the Multithreaded Debugger	*page 12-151*
Displaying the Process Inspector	*page 12-152*
The Process/Thread Inspector Window	*page 12-152*
Viewing the Context of Another Thread	*page 12-155*
LWP Information	*page 12-158*

12.1 Introduction to the Multithreaded Debugger

The Debugger recognizes a multithreaded program by checking whether it was compiled with `-lthread`.

When it detects a multithreaded program, the Debugger tries to `dlopen libthread_db.so`, a special library for thread debugging. The standard path for this library is `/usr/lib`; if it is located elsewhere, use the environment variable `THREAD_DB_DIR` to specify the correct path to `libthread_db.so`.

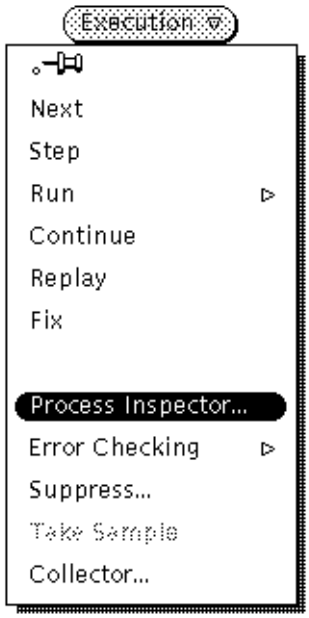
12

The Debugger is a synchronous debugger; that is, when any thread or lightweight process (LWP) stops, all other threads and LWPs "sympathetically" stop. This behavior is sometimes referred to as the "stop the world" model. For more information on threads and LWPs, see the "*SunOS 5.3 Guide to Multithread Programming*" manual.

12.2 Displaying the Process Inspector

To display the Process Inspector:

♦ **Choose Process Inspector from the Execution Menu.**



12.3 The Process/Thread Inspector Window

The Process Inspector is divided into two areas (see Figure 12-1 on page 154):

- Process
- Threads

The **Process** area shows the *pid* and *pathname* of your program. See Chapter 16, "Debugging Child Processes," for more information.

The **Threads** area shows information about the threads in the currently selected process.

The threads information is equivalent to the information from the command pane `threads` command. The Thread Inspector updates its information each time the program being debugged stops.

12.3.1 Thread Information

The following thread information is shown in Figure 12-1 on page 154:

- The **arrow** indicates the current thread.
- An * indicates that an event requiring user attention has occurred in this thread. The * is persistent; that is, it is reset only if the thread it is associated with resumes execution.

  An `o` indicates that an event has occurred that is of internal interest to the Debugger.
- **t@*num***, thread id, refers to a particular thread. The decimal number is the `thread_t` value passed back by `thr_create`.
- **b l@*num*** or **a l@*num*** (l is a lower-case L). Thread is bound to or active on the designated LWP. This means the thread is actually running. The state of a bound or active thread is really the state of its LWP; see Table 12-2.

  l@*num*, LWP id, refers to a particular LWP. The decimal number is equivalent to the value in `lwp_id_t`.

Note – At any given time the Debugger is focused on a single thread or LWP, known as the "current active entity." When the current active entity is nonexistent, it is assumed to be dead and shows up as `t@X` or `l@X`. For example, this happens when the Debugger steps over the return from a thread's start function or the process terminates.

- Start function of the thread as passed to `thr_create`. A ?() means the start function is not known; for example, if the thread is private to `libthread`.
- Thread state. (Table 12-1 describes the various thread states.)
- The function the thread is currently executing.

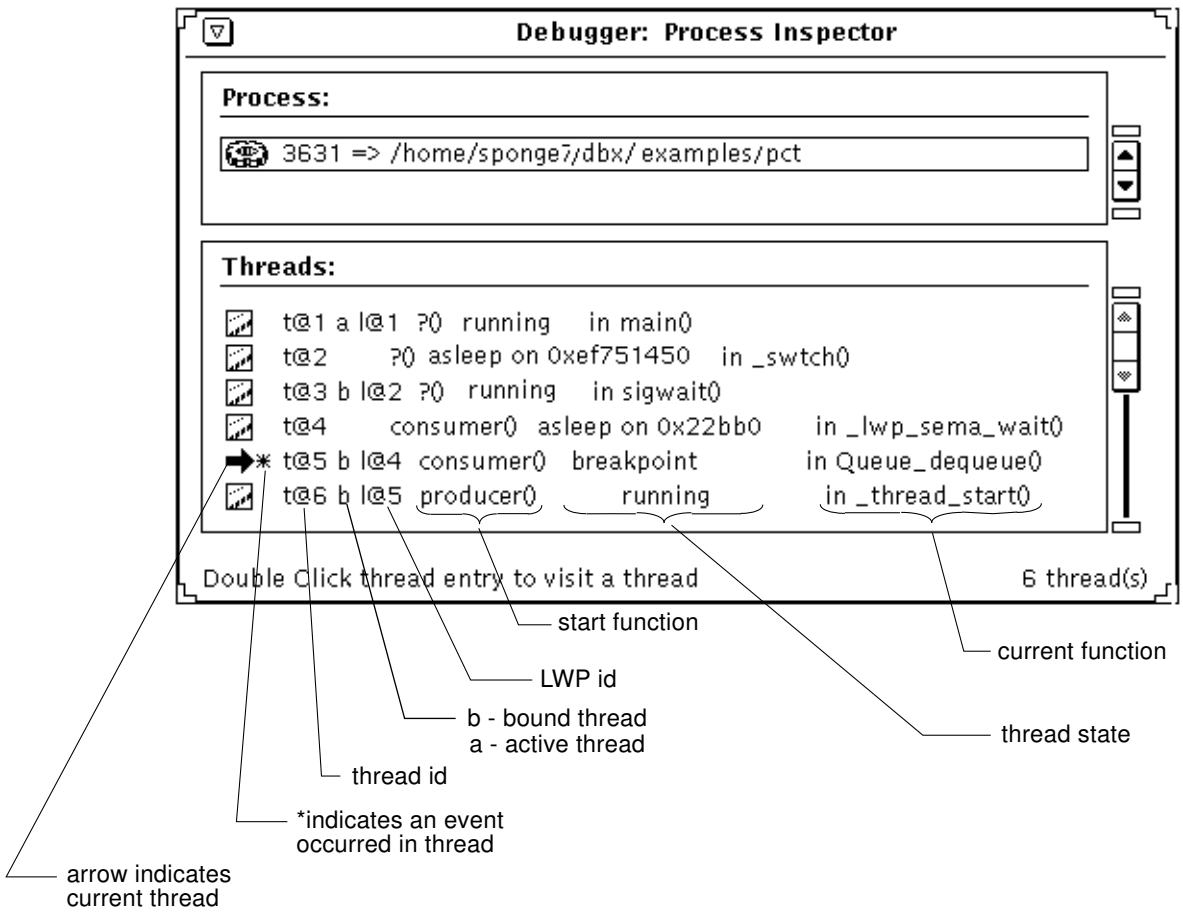


Figure 12-1 The Process/Thread Inspector Window

Table 12-1 shows thread state information.

Table 12-1 Thread States

Thread State	Description
suspended	Thread has been explicitly suspended.
runnable	Thread is runnable and is waiting for an LWP as a computational resource.
zombie	Thread has exited, but has not yet been "joined" or has not yet been reaped (if `THR_DETACHED` was used).
asleep on *syncobj*	Thread is blocked on the given synchronization object. Depending on what level of support `libthread` and `libthread_db` provide, *syncobj* might be as simple as a hexadecimal address or something with more information content.
lwpstate	A bound or active thread state is the state of the LWP associated with it. See Table 12-2 for a list of LWP states.
active	The thread is active on an LWP, but the Debugger cannot access the LWP.
unknown	The Debugger cannot determine the state.

12.4 Viewing the Context of Another Thread

To switch the viewing context to another thread, double click the Thread window on the text line of the desired thread (see Figure 12-2 on page 156).

The Debugger dynamically updates the call stack to reflect the context of the currently selected thread.

The Debugger also dynamically updates the source display context to that of the currently selected thread. The source display is empty if source for the selected thread has not been compiled with `-g`.

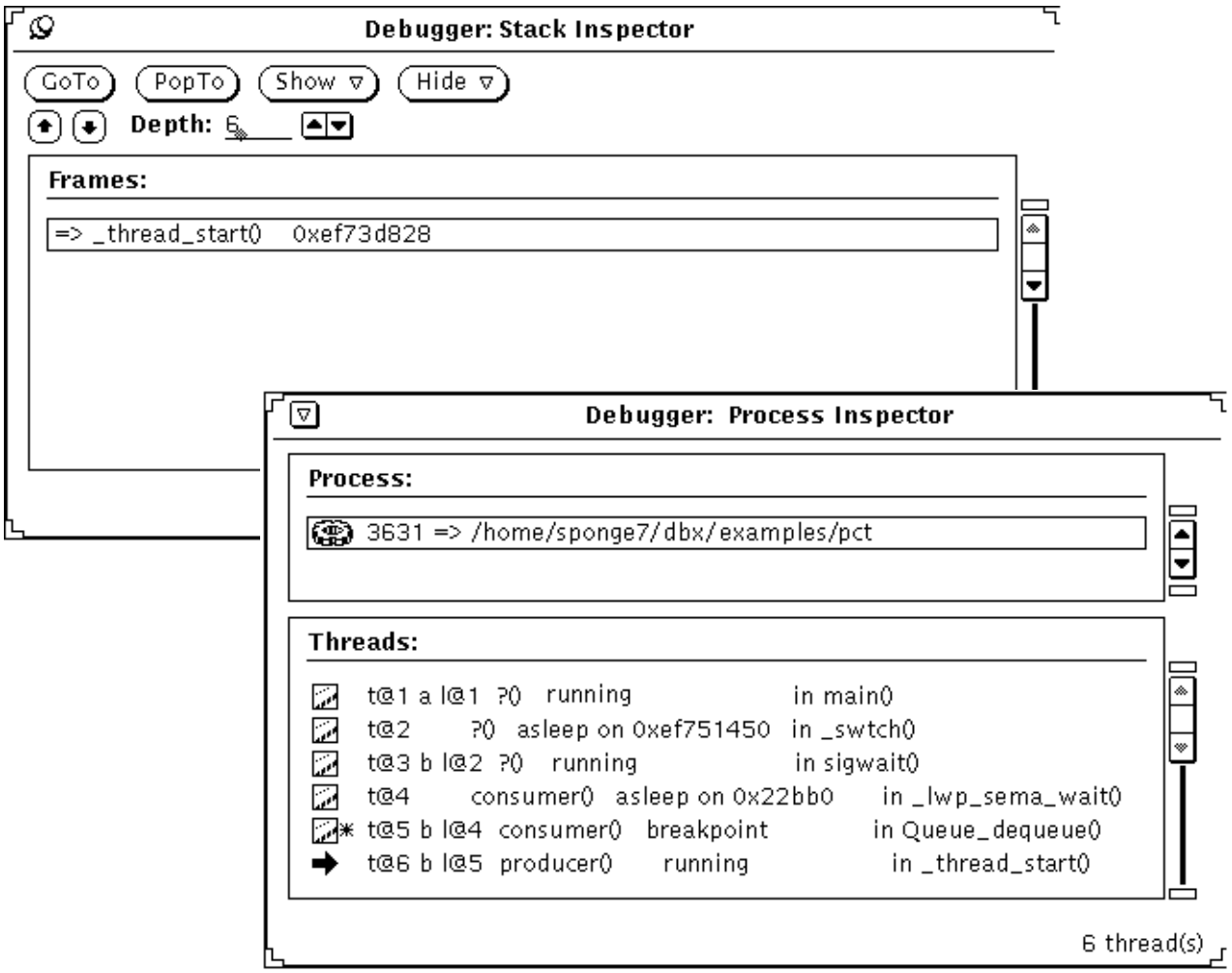
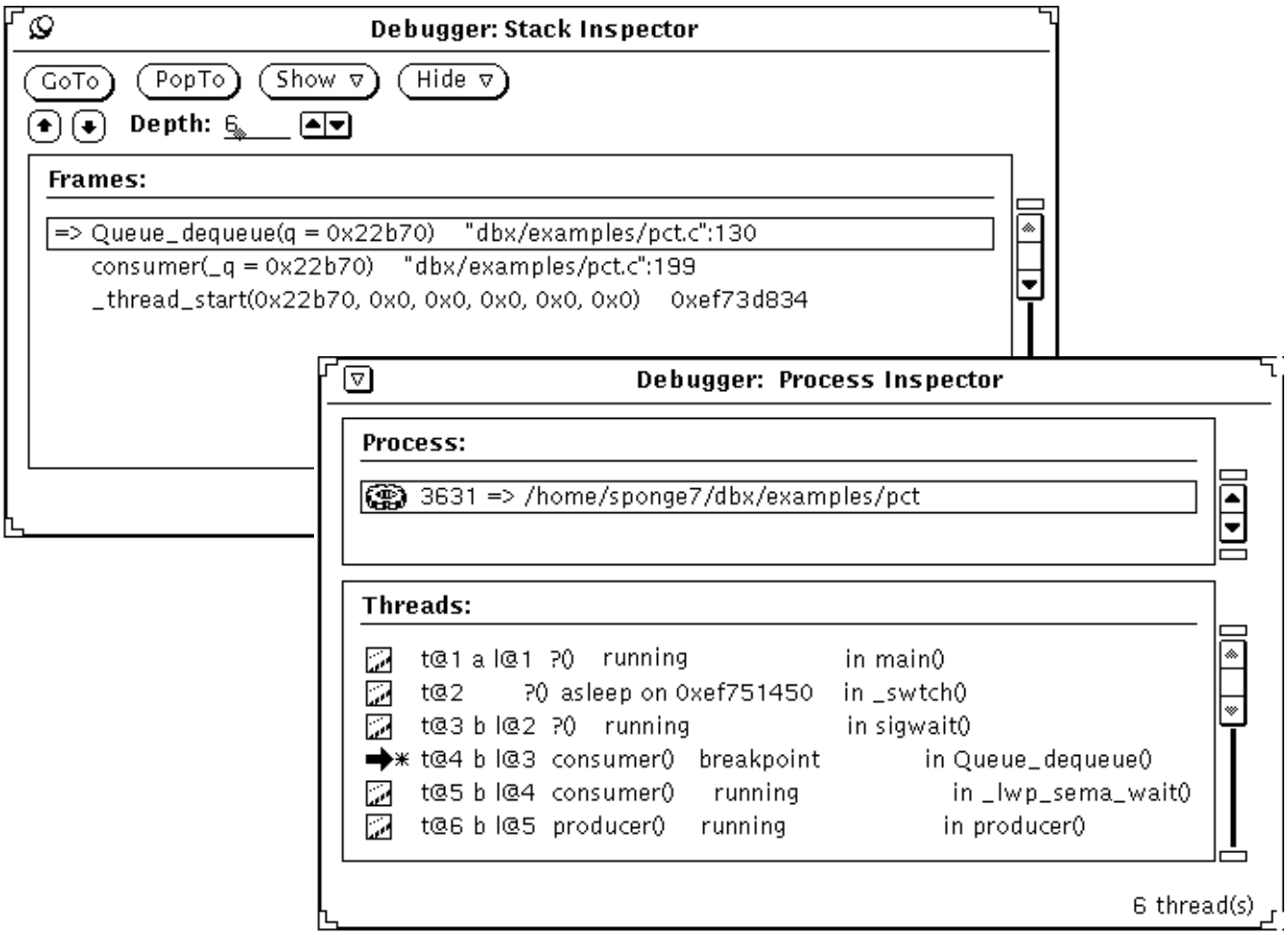


Figure 12-2 Viewing the Context of Another Thread


Debugger: Stack Inspector
GoTo
PopTo
Show
Hide
Depth: 6
Frames:
=> Queue_dequeue(q = 0x22b70) "dbx/examples/pct.c":130
consumer(_q = 0x22b70) "dbx/examples/pct.c":199
_thread_start(0x22b70, 0x0, 0x0, 0x0, 0x0, 0x0) 0xef73d834
Debugger: Process Inspector
Process:
3631 => /home/sponge7/dbx/examples/pct
Threads:
t@1 a l@1 ?() running in main()
t@2 ?() asleep on 0xef751450 in _swtch()
t@3 b l@2 ?() running in sigwait()
t@4 b l@3 consumer() breakpoint in Queue_dequeue()
t@5 b l@4 consumer() running in _lwp_sema_wait()
t@6 b l@5 producer() running in producer()
6 thread(s)

The Stack Inspector with Threads

The Stack Inspector is shown in this document to demonstrate how the stack information changes to the context of the "current" thread.

♦ **Choose Inspector from the Stack menu.**

The Debugger displays the call stack in the Stack Inspector window and dynamically updates it when you visit another process or thread. Use the features of the Stack Inspector to perform operations on the stack. See Chapter 9, "Examining the Call Stack," for more information.

12.4.1 Resuming Execution

Use the `cont` command to resume program execution. Currently, threads use synchronous breakpoints so all threads resume execution.

12.5 LWP Information

Normally, you need not be aware of LWPs. There are times, however, when thread level queries cannot be completed. In this case, use the `lwps` command to show information about LWPs.

```
(debugger) lwps
   l@1 running in main()
   l@2 running in sigwait()
   l@3 running in _lwp_sema_wait()
 *>l@4 breakpoint in Queue_dequeue()
   l@5 running in _thread_start()
(debugger)
```

- The * indicates that an event requiring user attention has occurred in this LWP.
- The **arrow** denotes the current LWP.
- **l@*num*** refers to a particular LWP.
- The next item represents LWP state as described in Table 12-2.
- **in *func_name*()** identifies the function that the LWP is currently executing.

Table 12-2 LWP State

LWP State	Description
running	LWP was running but was stopped in synchrony with some other LWP.
syscall *num*	LWP stopped on an entry into the given system call #.
syscall return *num*	LWP stopped on an exit from the given system call #.
job control	LWP stopped due to job control.
LWP suspended	LWP is blocked in the kernel.
single stepped	LWP has just completed a single step.
breakpoint	LWP has just hit a breakpoint.
fault *num*	LWP has incurred the given fault #.
signal *name*	LWP has incurred the given signal.
process sync	The process to which this LWP belongs has just started executing (for more information, see the discussion of `sync` event in Section 6.4, “Event Specifications”).
LWP death	LWP is in the process of exiting.

12

Customizing the Debugger 13

This chapter describes how to use the controls on the Properties window and how to use the initialization file, `.dbxrc` (or `.dbxinit`), to customize the Debugger. You can make adjustments to certain attributes of the debugging environment and to the size of the source, command, and data display panes. You can also add command buttons and Custom menu items to the interface. The initialization file lets you preserve your changes and adjustments from session to session.

This chapter is organized into the following sections:

Changing the Debugger Environment Attributes	*page 13-163*
Debugger Events	*page 13-167*
Changing the Debugger Window Configurations	*page 13-168*
Changing Settings for Run Time Check	*page 13-171*
Setting the Source/Object File Search Path	*page 13-171*
Setting Miscellaneous Properties	*page 13-173*
Using the Debugger Initialization File	*page 13-176*
Adding Buttons	*page 13-178*

The Props Button

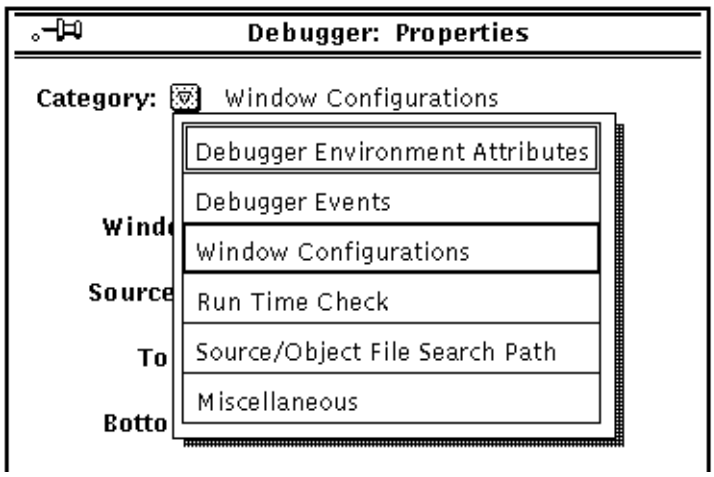
Clicking on the Props button displays the Properties window. The Properties window controls are arranged on six separate control panels:

- **Debugger Environment Attributes**
  Controls for each of the Debugger environment attributes: arguments to Matching, Auto Flush, Function Overload, Search Static Symbol, Short File Name, Stack Verbose, Stack Depth, and String Length.
- **Debugger Events**
  Controls for setting Auto Destruct, Follow Fork Inherit and Mode (Solaris 2.x), and the Trace Speed.
- **Window Configurations**
  Controls to adjust the size of each of the display panes and the font file the Debugger uses to display code.
- **Run Time Check**
  Control for setting Auto continue and entering an Error Log Filename.
- **Source/Object File Search Path**
  Controls for setting the current Debugger source/object code search path (a window interface to the Debugger `use` command).
- **Miscellaneous**
  Controls to set Compress Stabs (Solaris 2.x), Load Object Cache, and Collector Icon Animation on or off (Solaris 2.x). You can also set Source Display Synchronization, Executable Cache Size, change the Log Filename, and set the Make Arguments.

The Properties window initially displays the Debugger Environment Attributes control panel.

To display a different control panel from the one currently displayed:

♦ **Use the Category menu to display a different panel.**



13.1 Changing the Debugger Environment Attributes

To display the Debugger Environment Attributes control panel:

1. **Choose Props to display the Properties window. Debugger Environment Attributes is the default category.**
   If the Properties window already displays one of the other panels, use the Category menu to select the Debugger Environment Attributes window.
2. **Set the new value for each attribute, then press the Apply button.**

Figure 13-1 shows the Debugger Environment Attributes.

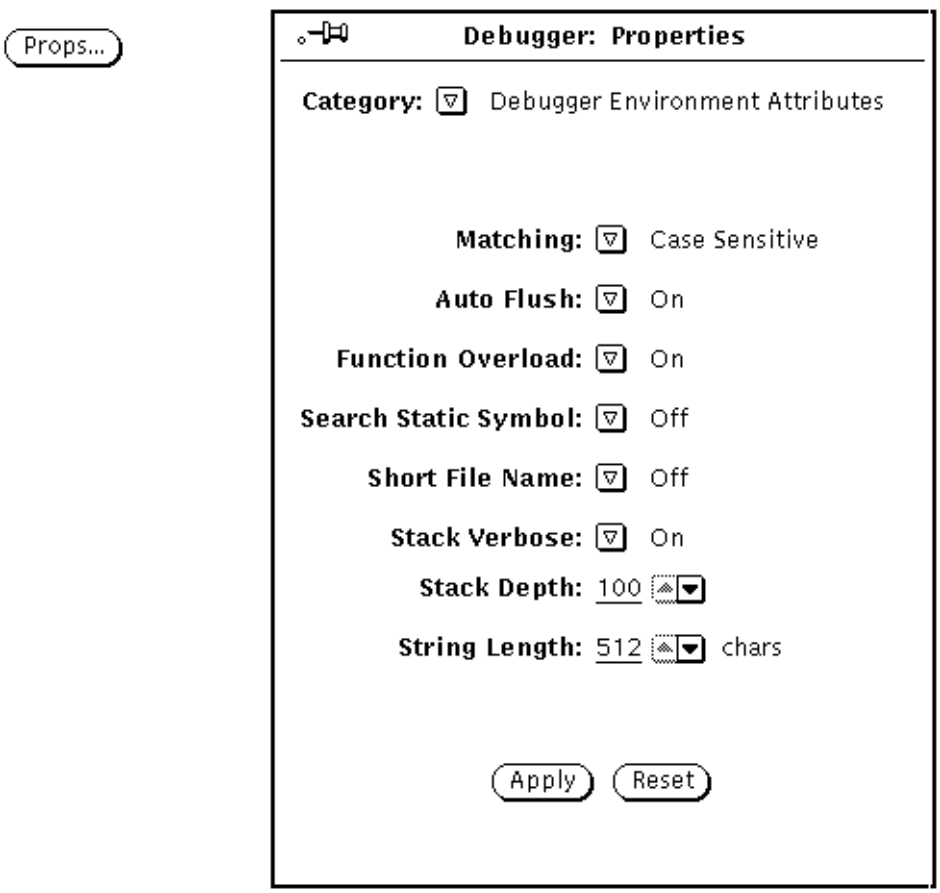


Figure 13-1 Debugger Environment Attributes Control Panel

Matching:

Set whether the Debugger distinguishes between upper- and lower-case characters when interpreting input.

`Case Sensitive`: treat upper and lower case distinctly.
`Case Insensitive`: process upper case characters as if lower case characters.

Default: Case Sensitive (C, C++, Pascal); Case Insensitive FORTRAN
Debugger command: `dbxenv case {sensitive | insensitive}`

Note – The Debugger detects when you load a FORTRAN module; it switches the setting automatically to `case insensitive` (with the exception that `MAIN()` must always be uppercase.) However, if during a session (or by setting `dbxenv case sensitive` in a `.dbxrc` file), you change the default manually, the Debugger does not override the manual setting when loading a FORTRAN module. Also, if the program was compiled using the `-u` option (case sensitive), the default becomes `dbxenv case sensitive`.

Auto Flush:

Set if you want the Debugger to call `fflush(stdout)` automatically after user calls to functions. `fflush(stdout)` flushes the I/O buffer, assuring that any information in it prints.

You can call a function *explicitly*, using the `call` command, or *implicitly*, by evaluating an expression containing a function or using a Debugger command with a conditional modifier containing a function (for example, in a command like `stop at glyph if animate()`).

To turn off automatic flushing, set the Auto Flush control to `Off`. Auto Flush is located on the Debugging Environment Attributes Properties Window.

On: Automatically call `fflush`() after user calls to functions.
Off: Do not call `fflush`() automatically after user calls to functions.

Default: On
Debugger command: `dbxenv autoflush {on | off}`

Function Overload:

Set to allow the Debugger to do automatic overload resolution of C++ functions.

Default: On
Debugger command: `dbxenv func_overload {on | off}`

Search Static Symbol:

Set to find file static symbols, even when they are not in the scope.

Default: Off
Debugger command: `dbxenv lookaside {on | off}`

Short File Name:

Set if you want the Debugger to truncate full pathnames. Shortened names are especially helpful in reducing the output from the `where` command.

Off: Display full pathnames for files.
On: display short (relative) pathnames for files.

Default: Off
Debugger command: `dbxenv shortfname {on | off}`

Stack Verbose:

Set to allow printing of arguments and line information in `where`.

Default: On
Debugger command: `dbxenv stackverbose {on | off}`

Stack Depth:

Set the default size for the `where` command.

Default: 100
Debugger command: `dbxenv stackmaxsize <num>`

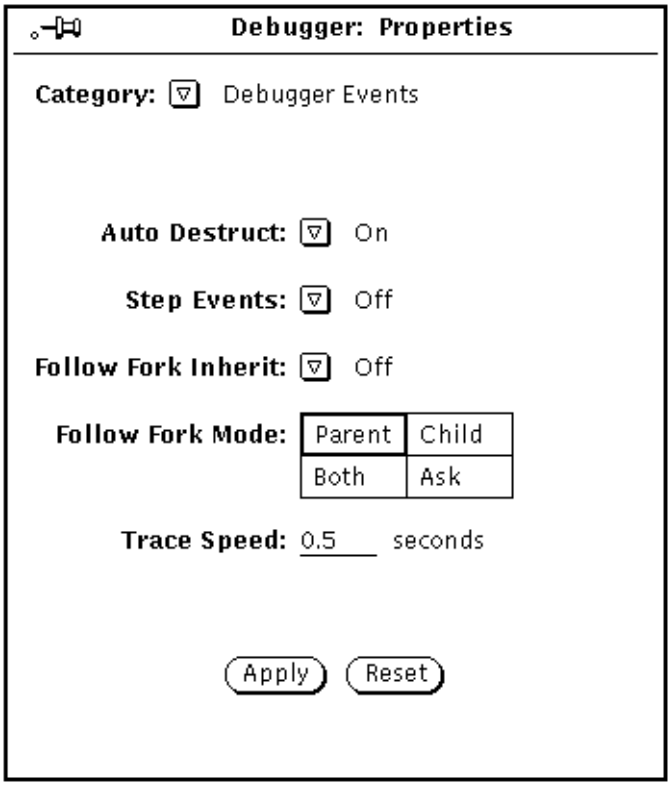
String Length:

Set maximum number of characters printed for character pointers, 'char *'.

Default: 512 characters.
Debugger command: `dbxenv stringlen` *num*ber

13.2 Debugger Events

You can set Auto Destruct, Step Events, and Follow Fork Inherit `on` or `off` from the Debugger Events Properties window (Solaris 2.x). You can also set Follow Fork Mode to Parent, Child, Both, or Ask (Solaris 2.x). And you can set the Trace Speed.



Auto Destruct

Set to allow automatic call of appropriate destructors for locals when "popping" a frame.

Default: On
Debugger command: `dbxenv pop_autodestruct {on | off}`

Step Events

Set `on` to allow breakpoints while using the `step` and `next` commands.

Default: Off
Debugger command: `dbxenv stepevents {on | off}`

(Solaris 2.x) **Follow Fork Inherit**

When following a child process, set Follow Fork Inherit `on` to inherit events and `off` not to inherit events.

Default: Off
Debugger command: `dbxenv follow_fork_inherit {on | off}`

(Solaris 2.x) **Follow Fork Mode**

Set the Follow Fork Mode so when a process executes a fork, vfork, or a fork1, you can:

- Follow the parent
- Follow the child
- Follow both the parent and the child (Debugger window only)
- Follow the parent or the child after being asked by the Debugger

Default: Parent
Debugger command: `dbxenv follow_fork_mode {parent | child | ask}`

Trace Speed

Set the minimum time between each printout of information to the command pane during a trace command.

Default: 0.5 seconds
Debugger command: `dbxenv speed number_seconds`

13.3 Changing the Debugger Window Configurations

You can adjust the size of the source display, the command pane, and the Data Display window pane. You control the number of vertical lines in each pane. You control the width of the pane by setting a maximum number of characters before the line wraps. Finally, you control the number of lines that appear in the source display above and below the lines you bring into view.

You can also use the base window resize-corners to alter the size of the windows and panes.

You can change the default font that the Debugger uses for displaying code with the Font File control. Figure 13-2 shows the default settings.

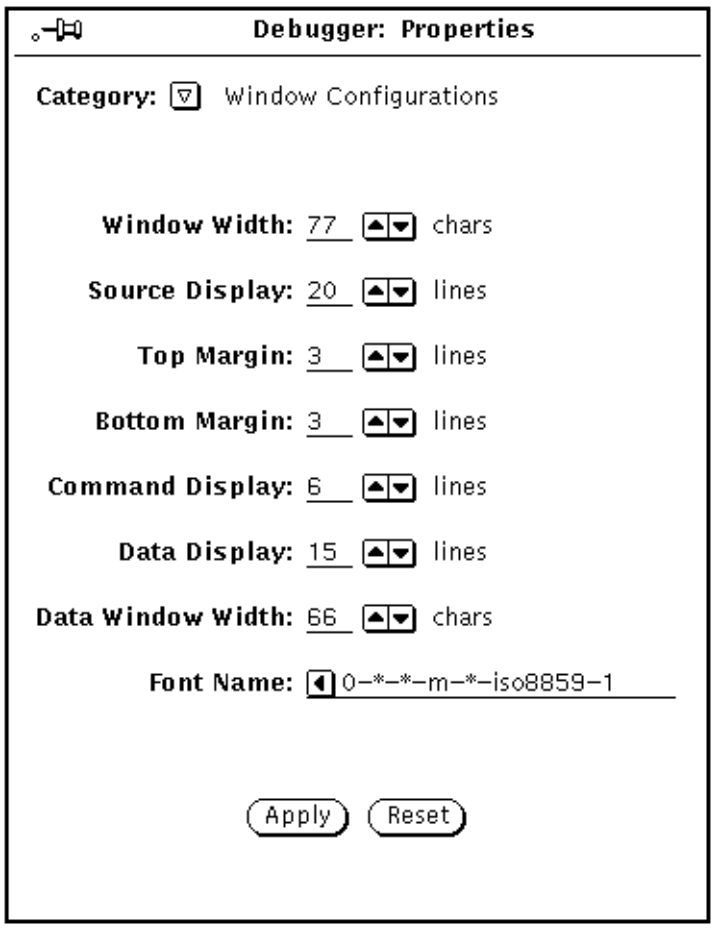


Figure 13-2 Window Configuration Control Panel

Window Width:

Set the width, in characters, of the source display or command pane.

Default: 77 characters
Command pane: `toolenv width` *num*

Source Display:

Set the number of lines of code displayed in the source display.

Default: 20 lines
Command pane: `toolenv srclines` *num*

Top Margin:

Set the minimum number of lines of code the Debugger displays above the line corresponding to the current program location each time the program stops.

Default: 3 lines
Command pane: `toolenv topmargin` *num*

Bottom Margin:

Set the minimum number of lines of code the Debugger displays below the line of code corresponding to the current program location each time the program stops.

Default: 3 lines
Command pane: `toolenv botmargin` *num*

Command Display:

Set the number of lines displayed in the command pane.

Default: 6 lines
Command pane: `toolenv cmdlines` *num*

Data Display:

Set the number of lines displayed in the Data Display Window.

Default: 15 lines
Command pane: `toolenv displines` *num*

Data Window Width:

Set the width, in characters, of the Data Display Window.

Default: 66
Command pane: `toolenv dispwidth num`

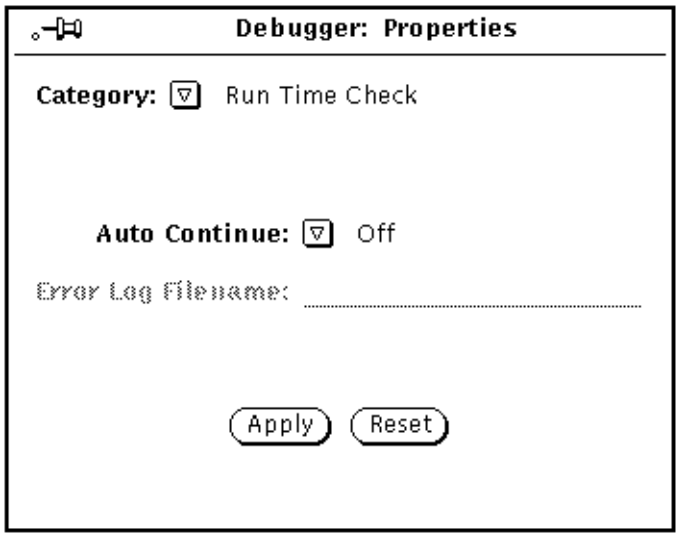
Font Name:

Set the name of the font file the Debugger uses to display code in the Debugger windows and panes.

Default: s*ystem_default_file*
Command pane: `toolenv font` *file*

13.4 Changing Settings for Run Time Check

(SPARC) You can set Auto Continue from the Run Time Check category window. With Auto Continue set to `on`, you can log Run Time Check errors to the `Error Log Filename` that you type in the Properties window.



Default: Off
Command pane: `dbxenv autocontinue {on | off}`
Command pane: `dbxenv errlogfile <filename>`

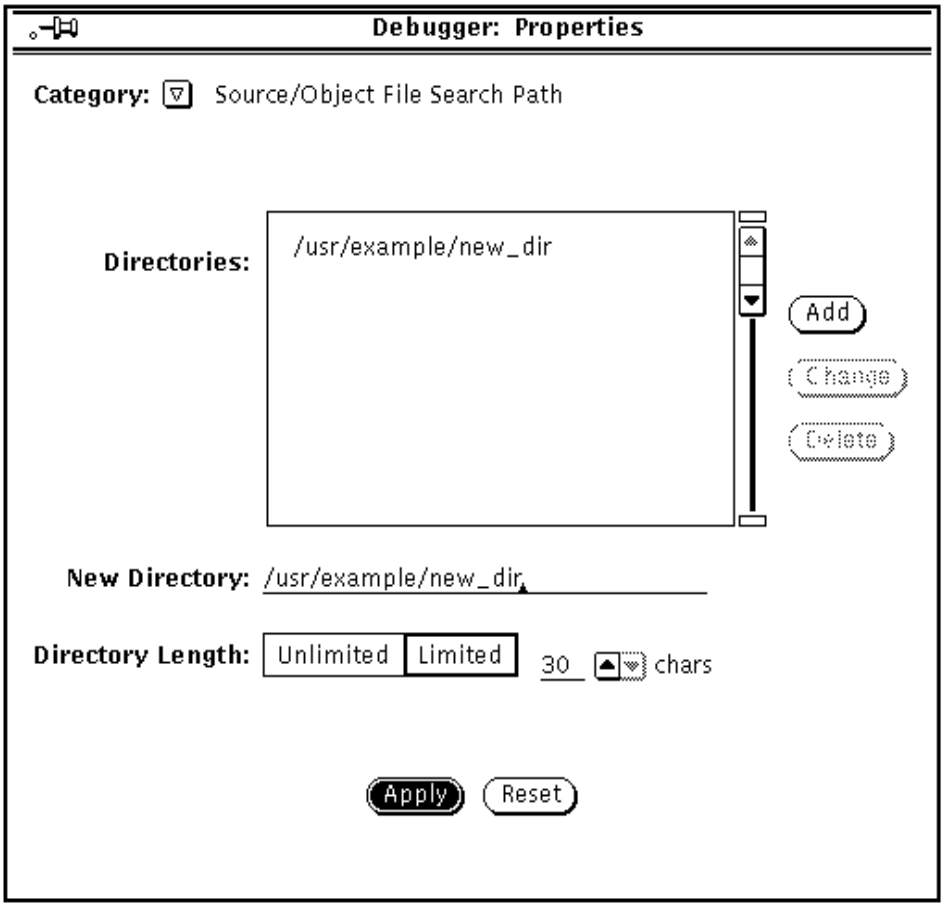
13.5 Setting the Source/Object File Search Path

You can check the path, add to the list, edit a pathname, or delete a pathname from the Source/Object File Search Path window.

To check the current File Search Path:

- **Display the Properties Window and choose Source/Object File Search Path from the Category menu.**

.



Note that you must click the Apply button to execute the changes you make to the search path. Until you click Apply, the search path does not change. You can click Reset *before* clicking Apply to discard your most recent edits.

To Add a directory to the Source/Object File Search Path:

1. **Type in the name of the directory you want to add in the New Directory text field; click Add.**
   The directory name appears in the Directories display box.
2. **Click Apply to apply the change.**

To Change a pathname already on the list:

1. **Select the path you want to edit by clicking on it in the Directories display box.**
   The control boxes the selected entry.
2. **Click the Change button.**
   The control writes the pathname on the New Directory text field.
3. **Edit the pathname on the New Directory text field; then click Apply.**

To delete a path from the search path list:

- **Select a pathname in the Directories box; click Delete; then click Apply.**

Adjusting the Display of Pathnames in the Directories Box

By default, the Directories box truncates pathnames on the left if they are larger than 30 characters. You can adjust this setting up or down to see more or less of the pathnames.

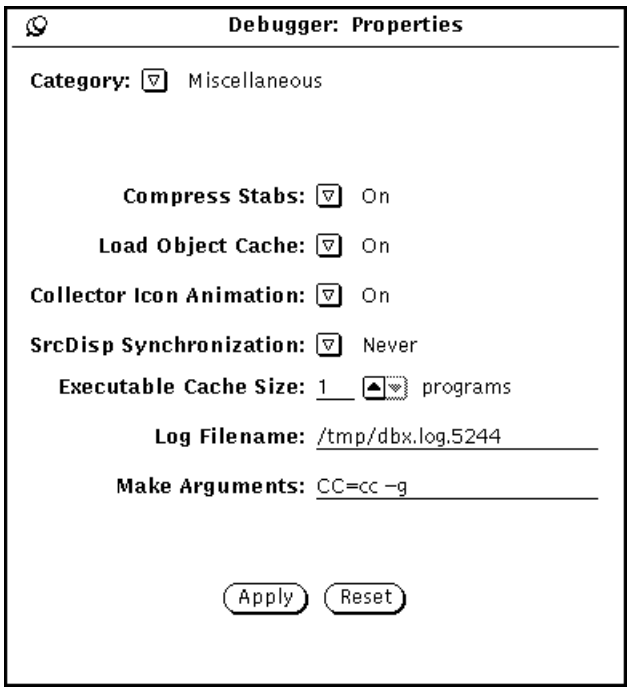
Also, if you set Directories Length to Unlimited, the box expands to accommodate the longest pathname, up to the maximum size of characters before truncating the name.

13.6 Setting Miscellaneous Properties

Choose Miscellaneous from the Debugger Properties menu to set Compress Stabs (Solaris 2.x), Load Object Cache, and Collector Icon Animation On or Off (Solaris 2.x). You can also set Source Display Synchronization, Executable Cache Size, change the Log Filename, and set the Make Arguments.

(Solaris 2.x) **Compress Stabs**

Set on to read debugging information for each include file only once.



Default: On
Debugger command: `dbxenv compress_stabs {on | off}`

Load Object Cache

Enable or disable the load object cache.

Default: On
Debugger command: `dbxenv lo_cache {on | off}`

(Solaris 2.x) **Collector Icon Animation**

Set the Animated Icon for the Collector Performance Tool to `On` or `Off`. This property is inactive unless the Collector is active.

Default: On
Debugger command: None

SrcDisp Synchronization

Set to allow display of code in the source display from selections made in SourceBrowser, CallGrapher, ClassBrowser, and ClassGrapher. If you issue a source display command from one of these tools, a `list` command is issued, causing the selected code to display in the Debugger, bypassing the SourceBrowser code display pane.

Always: The Debugger responds to an src display message every time you choose a source display command in one of these other tools.

Never: The Debugger ignores all src display messages when a source command is selected in one of these tools.

Default: Never
Debugger command: None, use the Properties window

Executable Cache Size

Set the size of the `a.out` load object cache. Set to <*n*> when debugging <*n*> programs from a single dbx. A <*n*> of zero still allows caching of shared objects.

Default: 1
Debugger command: `dbxenv aout_cache_size <num>`

Log Filename

Rename the Debugger log file.

Default: `/tmp/dbx.log.pid`

Make Arguments

Set the arguments passed to Make. You can also set arguments for Make in the Make window. Choose Make With Argument from the Program Make pull-right menu.

Default: SunOS 5.x `CC=cc -g` (ANSI C, with debugging)
Debugger command: `dbxenv makeargs "string"`

13

13.7 Using the Debugger Initialization File

The Debugger initialization file, `.dbxrc`, stores Debugger commands that are executed each time you start the Debugger. Typically, the file contains commands that customize the debugging environment, change the default window pane settings, and add command buttons or Custom menu items to the interface. You may also place Debugger commands in the `.dbxrc` file. Remember, if you customize the Debugger from the command pane, the customized settings apply only to the current debugging session.

During startup, the Debugger searches for `.dbxrc` first (ksh mode, see Chapter 23, "Korn Shell"). The search order is:

1. Current directory `./.dbxrc`
2. Home directory `$HOME/.dbxrc`

If `.dbxrc` is not found, the Debugger prints a warning message and searches for `.dbxinit` (dbx mode). The search order is:

1. Current directory `./.dbxinit`
2. Home directory `$HOME/.dbxinit`

To suppress the warning message, type in the command pane,

```
help .dbxrc >$HOME/.dbxrc
```

For additional information, type `help .dbxrc` in the command pane.

13.7.1 A Sample Initialization File

Here is a sample `.dbxrc` file:

```
dbxenv case insensitive
toolenv srclines 35
button ignore Edit
catch FPE
```

The first line changes the default setting for the case sensitivity control (the Matching control on the Debugging Environment Attributes control panel).

`dbxenv` refers to the set of debugging environment attributes.
`case` refers to the Matching control.
`insensitive` is the control setting.

The second line alters the default value for Window Configuration controls:

`toolenv` refers to the set of Window Configuration controls.
`srclines 35` sets the number of lines in the source display to 35.

The third line creates a new command button, Edit:

`button` creates a button command.
`ignore` instructs the Debugger to ignore the current mouse selection.
`Edit` is the name of the button.

The fourth line is a Debugging command, `catch`, which adds a system signal (`FPE`) to the default list of signals to which the Debugger responds, stopping the program.

13.7.2 *Setting the Graphical User Interface (GUI) Resources*

Here is a list of the Debugger X resources you can set in the `$HOME/.debugger-init` file.

```
Debugger.CommandWindow.Lines:<int>
Debugger.DataDisplay.Lines:<int>
Debugger.DataDisplay.Width:<int>
Debugger.SourceWindow.Lines:<int>
Debugger.SourceWindow.TopMargin:<int>
Debugger.SourceWindow.BotMargin:<int>
Debugger.Window.Width:<int>
Debugger.Window.FontName:<string>
```

Here is a list of the Debugger X resource defaults you can set in the `$HOME/.Xdefaults` file.

```
vdi*font:<string>
vdi*Node*font:<string>
vdi*Node*EntryValue*font:<string>
vdi*exprInput*font:<string>
vdi*Node*background:<string>
vdi*Node*borderColor:<string>
```

```
vdi*Node*port*background:<string>
vdi*Node*Header*background:<string>
vdi*Node*Header*borderColor:<string>

!
!   custom application resources
!
vdi*labelLength:<int>
vdi*displayCount:<int>
vdi*entryLength:<int>
```

13.8 Adding Buttons

The default interface for the Debugger includes a row of command buttons. This section describes how to add buttons. If you add buttons from the command line, the Debugger discards them when you quit the current session. To have buttons reinserted into the interface each time you start the Debugger, put the commands that create them in the `.dbxrc` file. You can add as many buttons as you like. As you add button commands, the Debugger adds new rows for them, as needed.

13.8.1 The Button Command

The syntax for adding a button command is `button` followed by a selection service modifier and the name of the button:

`button` *selection_service_modifier Name*

The selection service modifier tells the Debugger how to treat the current mouse selection (if there is one) when you click a new button. See Table 13-1 for a description of the selection service modifiers. For some types of commands, the current selection serves as the target of the command. For example, when you select a variable name and then click Print.

Other commands or menu items ignore the current selection. For example, if you create an Edit button to start your system editor in a Command tool, the current selection is irrelevant. The button command accepts one or a combination of the following selection service modifiers.

Table 13-1 Selection Service Modifiers for Button Commands

Modifier	Description
`command`	Issue the Debugger command selected in the command pane. This modifier works well as an alternative to using the history facility for reissuing a complex debugging command. You can select the entire debugging command, then click the button you created using the `command` modifier to reissue the command.
`expand`	If the selected characters in either pane begin with an alphanumeric character, dollar sign, or underscore, then first expand the selection, then use the expanded selection as the target for the command button or menu item.
`ignore`	Ignore the current mouse selection.
`lineno`	Use the line number associated with the current selection as the target of the command.
`literal`	Use the selected material exactly as selected for the command target. This modifier works the opposite of `expand`.

13

Editing a Program 14

This chapter describes how to edit source code in a separate command tool or from within the Debugger. You can edit code in the source display after converting the initially read-only pane to an editable pane. To edit the currently displayed file in the source display, choose Enable Edit from the source display floating pop-up menu.

This chapter is organized into the following sections:.

Starting an Editor in a Separate Command Tool	*page 14-181*
Enabling Editing in the Source Display	*page 14-182*

14.1 Starting an Editor in a Separate Command Tool

You start an editor in a separate command tool by starting the Debugger with the `-editor` option:

```
% debugger -editor
```

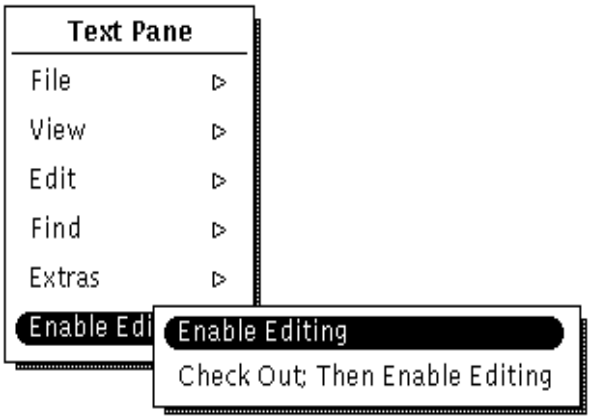
Starting the Debugger with the `-editor` option displays the Debugger user interface without the source display pane. The external editor which handles source display will be use.

14.2 Enabling Editing in the Source Display

By default, the source display is a browse-only pane.

To edit the file displayed in the source display,

♦ **Choose Enable editing from the source display floating pop-up menu. With the pointer in the source display, press the mouse MENU button.**



Note – When the source display is in editing mode, the debugging annotations (breakpoint and current focus markers) are not displayed.

14.2.1 Checking Out Files Under SCCS

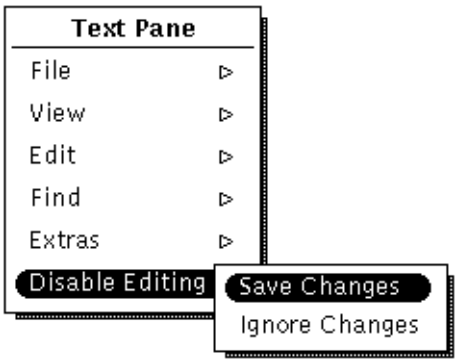
Notice that the Enable Edit item on the floating pop-up menu is a pull-right menu. The second item, Check Out; Then Enable Editing, automatically checks the file out of SCCS, if it is under this version control software. You cannot check it back into SCCS from within the Debugger.

14.2.2 Saving Changes, Disabling Editing

The Enable Editing item toggles to Disable Editing after you make the source display editable.

To save changes:

♦ **Choose Disable Editing from the source display floating pop-up menu.**



14.2.3 *Discarding Changes, Disabling Source Display Editing*

If you do not want to save changes made to the file currently displayed in the source display:

♦ **Choose Ignore Changes from the Disable Edit item on the floating pop-up menu.**

14

Part 3 — Advanced Debugging

Debugging at the Machine-Instruction Level

15

This chapter describes how to use event management and process control commands at the machine-instruction level, how to display the contents of memory at specified addresses, and how to display source lines along with their corresponding machine instructions. The `next`, `step`, `stop` and `trace` commands each support a machine-instruction level variant: `nexti`, `stepi`, `stopi` and `tracei`.

This chapter is organized into the following sections:

Examining the Contents of Memory	*page 15-187*
Stepping and Tracing at Machine-instruction Level	*page 15-192*
Setting Breakpoints at Machine-instruction Level	*page 15-194*
Sun-4 Register Information	*page 15-196*
x86 Register Information	*page 15-197*

15.1 Examining the Contents of Memory

Using addresses, you can examine the content of memory locations as well as print the assembly language instruction at each address. Using a command derived from `adb(1)`, the assembly language debugger, you can query for:

- The *address*, using the "=" (equal sign) character; or,
- The *contents* stored at an address, using the "/" (slash) character.

You can examine the contents of memory by using the `examine`, `dis`, and `listi` commands.

15.1.1 Using the `examine` Command

Here is the syntax that summarizes the examples shown in this section:

```
examine +
examine <addr>
examine <addr1>,<addr2>
examine <addr1>,<addr2>/<fmt>
examine <addr>/[<count>]<fmt>
examine <addr>=<fmt>
examine <addr>=
```

where

`examine` has a predefined alias, `x`.

+ displays the contents of the next address in the default format.

addr is any expression resulting in or usable as an address. Examples are:

0xff99	an absolute address
&*main*	address of a function
&*main+20*	offset from a function address
&*errno*	address of a variable
str	a pointer-value variable pointing to a string

Symbolic addresses used to display memory are specified by preceding a name with an ampersand (&). Registers are denoted by preceding a name with a dollar sign `($)`.

(SPARC) Table 15-2 lists the Sun-4 registers.

(x86) Table 15-3 lists the x86 registers.

Additionally, *addr* can be "+" which indicates the address after the last one displayed.

count is a repetition count in decimal. The increment size depends on the memory display format.

fmt is the address display format in which the Debugger displays the results of a query as shown in Table 15-1.

The output produced depends on the current display *fmt*. To change the display format, you supply a different code. (See "Alternative Memory Display Formats" on page 15-189.)

Here is an example of how to use an address with *count* and *fmt* options to display five successive disassembled instructions starting from the current stopping point.

(SPARC) This is a SPARC example:

```
(debugger) stepi
stopped in support at 0x117c8
(debugger) x 0x117c8/5i
support__5blockFv+8:    save    %sp, %g1, %sp
support__5blockFv+0xc:  st      %i0, [%fp + 68]
support__5blockFv+0x10: ld      [%fp + 68], %i3
support__5blockFv+0x14: ld      [%i3 + 24], %o0
support__5blockFv+0x18: ba      support__5blockFv+0x20
```

(x86) This is an x86 example:

```
(dbx) x &support/5i
support:          pushl   %ebp
support+1:        movl    %esp,%ebp
support+3:        subl    $4,%esp
support+6:        movl    8(%ebp),%ecx
support+0x9:      movl    24(%ecx),%ecx
```

To print the value stored at the next address after the one last displayed by examine:

♦ **Type**

```
(debugger) examine +/ i
```

Alternative Memory Display Formats

Notice in the previous example the inclusion of the display *fmt* specifier, `i`. Set the *fmt* specifier to tell the Debugger how to display information associated with the addresses specified.

The default format set at the start of each Debugger session is `x`, which displays an address/value as a 32-bit word in hexadecimal. Table 15-1 lists the possible formats.

Table 15-1 Memory Address Display Formats

Format	Description	
`i`	Display as an assembly instruction.	
`d`	Display as a halfword in decimal.	
`D`	Display as a word in decimal.	
`o`	Display as a halfword in octal.	
`O`	Display as a word in octal.	
`x`	Display as a halfword in hexadecimal.	
`X`	Display as a word in hexadecimal.	(default format)
`b`	Display as a byte in octal.	
`c`	Display a byte as a character.	
`w`	Display as a wide character.	
`s`	Display as a string of characters terminated by a null byt.	
`W`	Display as a wide-character string.	
`f`	Display as a single-precision floating point number.	
`F, g`	Display as a double-precision floating point number.	
`E`	Display as an extended-precision floating point number.	

15.1.2 *Using the* `dis` *Command*

Here is the syntax for the `dis` command. For syntax definitions, see the `examine` command.

`dis` <*addr*> [/<*count*>]
`dis` <*addr1*>, <*addr2*>
`dis`
`dis` /<*count*>

This command is equivalent to the `examine` command with '`i`' as the default display format.

`dis` without arguments displays 10 instructions starting at the *addr* +.
`dis` with only a <*count*> displays count instructions starting at the *addr* +.

15.1.3 Using the `listi` Command

To display source lines along with their corresponding assembly instructions, use `listi`.

Allowed forms are:

`listi`
`listi` < *num*>
`listi` <*first*>, <*last*>
`listi` <*procedure*>

where

`listi` displays 10 source lines with intermixed assembly.

<*num*> displays only that line. <*num*> can be a $ which denotes the last line of the file.

<*first*> and <*last*> bounds the range of source lines to display. <*last*> can be a $ which denotes the last line of the file.

<*procedure*> displays the start of the source for <*procedure*>.

(SPARC) Here is a SPARC example:

```
(debugger) listi 98,100
   98       int i = atoi(argv[1]);
main+0x50:     ld    [fp + 72], %10
main+0x54:     add   %10, 4, %10
main+0x58:     ld    [%10], %10
main+0x5c:     mov   %10, %o0
main+0x60:     call  _PROCEDURE_LINKAGE_TABLE_+0x60 [PLT]
main+0x64:     nop
main+0x68:     mov   %o0, %10
main+0x6c:     st    %10, [%fp + -8]
   99       foo(i);
main+0x70:     ld    [%fp + -8], %10
main+0x74:     mov   %10, %o0
main+0x78:     call  foo
main+0x7c:     nop
   100      exit(0);
main+0x80:     mov   0, %10
```

```
main+0x84:     mov   %10, %o0
main+0x88:     call  _PROCEDURE_LINKAGE_TABLE_+0x3C [PLT]
main+0x8c:     nop
```

(x86) Here is an x86 example:

```
(dbx) listi 98,100
   98            int i = atoi(argv[1]);
main+6:           movl    12(%ebp),%ecx
main+0x9:         pushl   4(%ecx)
main+0xc:         call    _PROCEDURE_LINKAGE_TABLE_+0x70
                           <0x8048664>
main+0x11:        addl    $4,%esp
main+0x14:        movl    %eax,-4(%ebp)
   99             foo(i);
main+0x17:        pushl   -4(%ebp)
main+0x1a:        call    foo <0x80486d4>
main+0x1f:        addl    $4,%esp
   100            exit(0);
main+0x22:        pushl   $0
main+0x24:        call    _PROCEDURE_LINKAGE_TABLE_+0x60
                           <0x8048654>
main+0x29:        addl    $4,%esp
```

Command `listi` is equivalent to `list -instr`.

15.2 Stepping and Tracing at Machine-instruction Level

Machine-instruction level commands behave the same as their source level counterparts except that they operate at the level of single instructions instead of source lines.

15.2.1 Single-Stepping the Machine-Instruction Level

To single-step from one machine-instruction to the next machine-instruction:

♦ **Use** `nexti` **or** `stepi` **in the command pane.**

`nexti` and `stepi` behave the same as their source-code level counterparts: `nexti` steps *over* functions, `stepi` steps *into* a function called from the "next" instruction (stopping at the first instruction in the called function). The command forms are also the same. See `next` and `step` for a description.

The output from `nexti` and `stepi` differs from the corresponding source level commands in two ways. First, the output includes the *address* of the instruction at which the program is stopped (instead of the source code line number); secondly, the default output contains the *disassembled instruction*. For example:

```
(debugger) func
hand::ungrasp
(debugger) nexti
ungrasp +0x18:   call  support
(debugger)
```

15.2.2 Tracing at the Machine-Instruction Level

Tracing techniques at the machine instruction level work the same as at the source code level except when you use `tracei`. For `tracei`, the Debugger executes a single instruction only after each check of the address being executed or the value of the variable being traced. `tracei` produces automatic *stepi*-like behavior: that is, the program advances one instruction at a time, stepping into function calls.

The Debugger supports two types of machine-instruction level traces:

- A trace of the execution of a specified address.
- A trace of a change in the value of a variable or expression at a specified address.

When you use `tracei`, it causes the program to stop momentarily after each instruction while the Debugger checks for the address execution or the value of the variable or expression being traced. Using `tracei` can slow execution considerably.

All types of machine-instruction level traces take a post-break conditional modifier. Here is the syntax:

`tracei` [`if` *cond*]
`tracei` *variable* [`at` *address*] [`if` *cond*]

where
variable refers to a memory location.
address refers to an instruction location.

To trace the execution of a variable at the machine-instruction level:

♦ **Type in the command pane,**

```
(debugger) tracei variable
```

The default output is the assembly instruction if the program changes the variable.

To trace the value of a variable when the program reaches an address:

♦ **Type in the command pane,**

```
(debugger) tracei variable at address
```

If the value stored at the specified address changes, the Debugger displays the trace. In this example, `tracei` reports each time the Blocks program "handle" object grasps a different solid object:

```
(debugger) tracei &object at 0x123b4
(debugger) cont
Move block B1 on top of block B2.
First I must get B1 out of the way.
at 0x123b4: : &object = 0xf7ffed78   // tracei output
Moving hand to pick up B1 at location 3:4.
Grasping B1.
Removing support relationship between B1 and B2.
Moving B1 to top of Table at location 10:0.
Adding support relationship between B1 and Table.
Releasing B1.
at 0x123b4: : &object = 0xf7ffef38    // tracei output
```

15.3 Setting Breakpoints at Machine-instruction Level

To set a breakpoint at machine-instruction level, use `stopi`. `stopi` accepts any <*event specification*> defined in Chapter 6.4, "Event Management", using the syntax:

`stopi` <*event specification*> [`if` *cond*]

Commonly used forms of the `stopi` command are:

`stopi` [`at` *address*] [`if` *cond*]
`stopi` [*variable*] [`if` *cond*]
`stopi in` *function* [`if` *cond*]

where
address refers to a memory location.

15.3.1 Setting a Breakpoint at an Address

To set a breakpoint at a specified address:

♦ **Type in the command pane,**

`(debugger)` **`stopi at`** *address*

For example,

```
(debugger) nexti
stopped in hand::ungrasp at 0x12638
(debugger) stopi at &hand::ungrasp
(3) stopi at 75320
(debugger)
```

15.4 Sun-4 Register Information

The following register information is for SPARC systems.

Table 15-2 Sun-4 Register Names

Register	Description
`$g0–$g7`	Global registers
`$o0–$o7`	“out” registers
`$i0–$i7`	“in” registers
`$l0–$l7`	“local” registers
`$fp`	Frame pointer, equivalent to register $i6
`$sp`	Stack pointer, equivalent to register $o6
`$y`	Y register
`$psr`	Processor state register
`$wim`	Window invalid mask register
`$tbr`	Trap base register
`$pc`	Program counter
`$npc`	Next program counter
`$f0–$31`	FPU “f” registers
`$fsr`	FPU status register
`$fq`	FPU queue

See the *SPARC Architecture Reference Manual* and the *Sun-4 Assembly Language Reference Manual* for more information on Sun-4 registers and addressing.

15.5 x86 Register Information

The following register information is for x86 systems.

Table 15-3 x86 Register Names

Register	Description
`$ss`	stack segment register
`$uesp`	user stack pointer
`$eflags`	flags
`$cs`	code segment register
`$eip`	instruction pointer
`$eax`	general register
`$ecx`	general register
`$edx`	general register
`$ebx`	general register
`$esp`	stack pointer
`$ebp`	frame pointer
`$esi`	source index register
`$edi`	destination index register
`$ds`	data segment register
`$es`	alternate data segment register
`$fs`	alternate data segment register
`$gs`	alternate data segment register

Additionally, commonly used registers are also aliased to their machine independent names:

`$sp`	stack pointer; equivalent of `$uesp`
`$pc`	program counter; equivalent of `$eip`
`$fp`	frame pointer; equivalent of `$ebp`

Registers for the 80386 lower halves (16 bits) are:

`$ax`	general register
`$cx`	general register
`$dx`	general register
`$bx`	general register
`$si`	source index register
`$di`	destination index register
`$ip`	instruction pointer, lower 16 bits
`$flags`	flags, lower 16 bits

The first four 80386 16-bit registers can be split into 8-bit parts:

`$al`	lower (right) half of register	`$ax`
`$ah`	higher (left) half of register	`$ax`
`$cl`	lower (right) half of register	`$cx`
`$ch`	higher (left) half of register	`$cx`
`$dl`	lower (right) half of register	`$dx`
`$dh`	higher (left) half of register	`$dx`
`$bl`	lower (right) half of register	`$bx`
`$bh`	higher (left) half of register	`$bx`

Registers for the 80387 are:

`$fctrl`	control register
`$fstat`	status register
`$ftag`	tag register
`$fip`	instruction pointer offset
`$fcs`	code segment selector
`$fopoff`	operand pointer offset
`$fopsel`	operand pointer selector
`$st0 – $st7`	data registers

Debugging Child Processes 16

This chapter describes how to debug a child process.

(Solaris 2.x) The Debugger has several facilities to help you debug processes that create children via `fork` (2) and `exec` (2).

This chapter is organized into the following sections:

Simple Attach	*page 16-199*
Follow exec (Solaris 2.x)	*page 16-200*
Follow fork Under dbx (tty Mode) (Solaris 2.x)	*page 16-200*
Follow fork Under the Debugger (GUI Mode) (Solaris 2.x)	*page 16-201*
Interaction with Events and Other Features	*page 16-201*

16.1 Simple Attach

In the simplest case, the child has already been created, so you can attach to it in one of the following ways:

- From the shell, when starting dbx or the Debugger, type

  $ **dbx <progname> <pid>**
  or,
  $ **dbx <progname> <pid> &**

- From the command line, type

(dbx) **debug <progname> <pid>**

(Solaris 2.x) In each case, <*progname*> can be substituted with the name "-", in which case the Debugger automatically finds the executable associated with the given process id (pid). After using a "-", a subsequent `run` or `rerun` does not work, because the Debugger does not know the full pathname of the executable.

16.2 *Follow* `exec` (Solaris 2.x)

If a child process executes (`exec(2)`) a new program, the process id does not change, but the process image does. dbx automatically takes note of an `exec()` and does an implicit reload of the newly executed program.

Because the first argument to `exec(2)` is a relative pathname, and dbx cannot reliably know the working directory of the child at the time of `exec()`, the new program is loaded using the "-" filename and a pid.

16.3 *Follow* `fork` *Under dbx (tty Mode)* (Solaris 2.x)

In analogy to the "Follow Exec" feature, if a child process does `fork(2)`, the process id changes, but the process image stays the same. Depending on how the dbxenv variable "`follow_fork_mode`" is set, dbx does the following:

parent This is the traditional behavior, dbx ignores the fork and follows the parent. Under Solaris 1.x, only `parent` is allowed.

child In this mode, dbx automatically switches to the forked child using the new pid. All connection to and awareness of the original parent is lost.

both This mode is only available under the graphical user interface (GUI). The GUI mode is described in the following section.

ask You are prompted to choose `parent`, `child`, or `both` whenever the Debugger detects a fork.

16.4 *Follow* `fork` *Under the Debugger (GUI Mode)* (Solaris 2.x)

When using the Debugger, in addition to the `follow_fork_modes` of *parent*, *child*, and *ask*, the mode *both* can be used. In this case, a new dbx is cloned (see the `debug` command option `-clone`). The Debugger can alternate between the two underlying dbx programs.

Dealing with multiple dbx programs attached to multiple processes is simplified through use of the Process/Thread Inspector (PTI). See Chapter 12, "Process/Thread Inspector", for more information. When a new dbx is cloned, a new line is displayed in the Process portion of the PTI window.

You can double-click on a line in the process portion of the PTI window to switch the Debugger to the dbx attached to the selected process.

You can also set `follow_fork_mode` from the Debugger Props button:

♦ **Choose Debugger Events from the Category menu.**

16.5 *Interaction with Events and Other Features*

All breakpoints and other events are deleted for any `exec()` or `fork()` process. You can override the deletion for follow fork by setting the dbxenv variable `follow_fork_inherit` to On. (You can set `follow_fork_inherit` to On under the Debugger Events category of the Props pop-up window.)

The Collector and RunTime Checking are disabled for an `exec()` or `fork()` process.

For Solaris 1.x, `follow_fork_inherit` is always Off.

Working with System Signals 17

This chapter describes how to use the Debugger to work with system signals. The Debugger supports a breakpoint command named `catch`. The `catch` command instructs the Debugger to stop a program when the Debugger detects any of the system signals appearing on the `catch` list.

Also, the Debugger `cont` command supports the `-sig` *signal_name* option, which allows you to resume execution of a program with the program behaving as if it had received the system signal specified in the `cont -sig` command.

This chapter is organized into the following sections:

Sequence of Events Involving Signals	*page 17-203*
Catching System Signals	*page 17-204*
Sending a System Signal in a Program	*page 17-206*

17.1 Sequence of Events Involving Signals

When a signal is to be delivered to a process that is being debugged, the signal is redirected to the Debugger by the kernel. When this happens, you usually get a prompt. You then have two choices:

1. **"cancel" the signal when the program is resumed. This is the default behavior of `cont`. This default behavior facilitates easy interruption and resumption using `SIGINT` (Control-C).**

2. "forward" the signal to the process by using

```
cont -sig sig
```

In addition, if a certain signal is received frequently, you can arrange for dbx to automatically forward the signal, by typing

```
ignore sig # "ignore" because you do not care to see it.
```

A default set of signals is automatically forwarded in this manner, see `ignore`.

17.2 Catching System Signals

By default, the `catch` list contains about 22 of the 33 detectable signals. (The numbers depend upon the operating system and version.) You can change the default `catch` list by adding or signals to or removing them from the default `catch` list.

To see the list of signals currently being trapped,

- **Type `catch` with no signal-name argument in the command pane:**

```
(debugger) catch
```

To see a list of the signals currently being *ignored* by the Debugger when the program detects them.

- **Type `ignore` with no signal-name argument in the command pane:**

```
(debugger) ignore
```

17.2.1 Default `catch` and `ignore` Lists

The default `catch` and `ignore` lists vary depending on the operating system.

(Solaris 2.x) The default `catch` list contains:

```
INT QUIT ILL TRAP ABRT SEGV SYS PIPE TERM USR1 USR2 PWR
URG POLL STOP TTIN TTOU VTARLM PROF XCPU XFSZ FREEZE THAW
```

The default `ignore` list contains:

```
HUP EMT FPE KILL ALRM CLD WINCH TSTP CONT WAITING LWP
```

(Solaris 1.x) The default `catch` list contains:

```
INT QUIT ILL TRAP ABRT BUS SEGV SYS PIPE TERM URG STOP
TTIN TTOU IO XCPU XFSZ VTALRM PROF LOST USR1 USR2
```

The default `ignore` list contains:

```
HUP EMT FPE KILL ALRM TSTP CONT CHLD WINCH
```

17.2.2 *Changing the Default* `catch` *and* `ignore` *Signal Lists*

You control which signals cause the program to stop by moving the signal names from one list to the other.

To move signal names from the one list to the other:

- **Supply a signal-name(s) argument that currently appears on one list as an argument(s) to the other list.**
  For example, to move the QUIT and ABRT signals from the `catch` list to the `ignore` list, type:

  ```
  (debugger) ignore QUIT ABRT
  ```

17.2.3 *Trapping the FPE Signal under Solaris 2.x*

Often programmers working with code that requires floating point calculations want to debug exceptions generated in a program. When a floating point exception like overflow or divide by zero occurs, the system returns a "reasonable" answer as the result for the operation that caused the exception. Returning a reasonable answer allows the program to continue executing quietly. (Solaris 2.x implements the IEEE Standard for Binary Floating Point Arithmetic definitions of "reasonable" answers for exceptions.)

Since a reasonable answer for floating point exceptions is returned under Solaris 2.x, exceptions do not automatically trigger the signal `SIGFPE`.

To find the cause of an exception, you need to set up a trap handler in the program so that the exception triggers the signal `SIGFPE`. (See `ieee_handler(3m)` for an example of a trap handler.) When you set up a

trap handler using `ieee_handler`, the trap enable mask in the hardware floating point status register is set. This trap enable mask causes the exception to raise `SIGFPE` at run time.

Once you have compiled the program with the trap handler, load the program into the Debugger. Before you can `catch` the `SIGFPE`, you must add `FPE` to the Debugger signal `catch` list, using the command,

```
(debugger) catch FPE
```

By default, `FPE` is on the `ignore` list.

After adding `FPE` to the `catch` list, run the program in the Debugger. When the exception you are trapping occurs, `SIGFPE` is raised and the Debugger stops the program. Now you can trace the call stack (use the Stack menu button or the Debugger `where` command) to help find the specific line number of the program where the exception occurs.

For more discussion and examples, see the floating point manual, *Numerical Computation Guide*, which comes with the compiler documentation.

17.3 Sending a System Signal in a Program

The Debugger `cont` command supports the `-sig` *signal_name* option, which allows you to resume execution of a program with the program behaving as if it had received the system signal specified in the `cont -sig` command.

For example, if a program has an interrupt handler for `SIGINT` (^c), you can type `^c` to stop the application and return control to the Debugger (as discussed in Section 7.4, "Using Cntl-C to Stop a Process," on page 7-111). Now, if you issue a `cont` command by itself to continue program execution, the interrupt handler never executes. To execute the interrupt handler, you send the signal, `sigint`, to the program:

```
(debugger) cont -sig int
```

The `stop`, `next`, and `detach` commands accept a `-sig` as well.

Collecting Performance Tuning Data 18

This feature is available only on Solaris 2.x and does not work with multithreaded applications.

This chapter describes how to start the Collector performance tuning tool from the Debugger and how to take a sample manually for use with the Analyzer. The Collector is the Analyzer tool that you use to prepare an application for data collection. During the execution of an application, the Collector gathers performance and test coverage data on an application of your choice. See the toolset manual entitled, *Performance Tuning an Application* for full details of the Collector and how to analyze the information it collects.

This chapter is organized into the following sections:

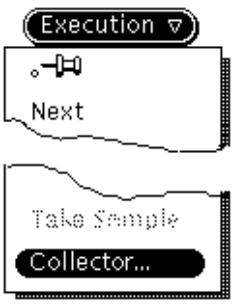
Starting the Collector from the Debugger	*page 18-208*
Taking a Sample for Use with the Analyzer	*page 18-210*

18

18.1 Starting the Collector from the Debugger

To start the Collector:

♦ **Choose Collector from the Execution menu.**



At any one time, you can have several Collector windows displayed, each performing a separate experiment.

.

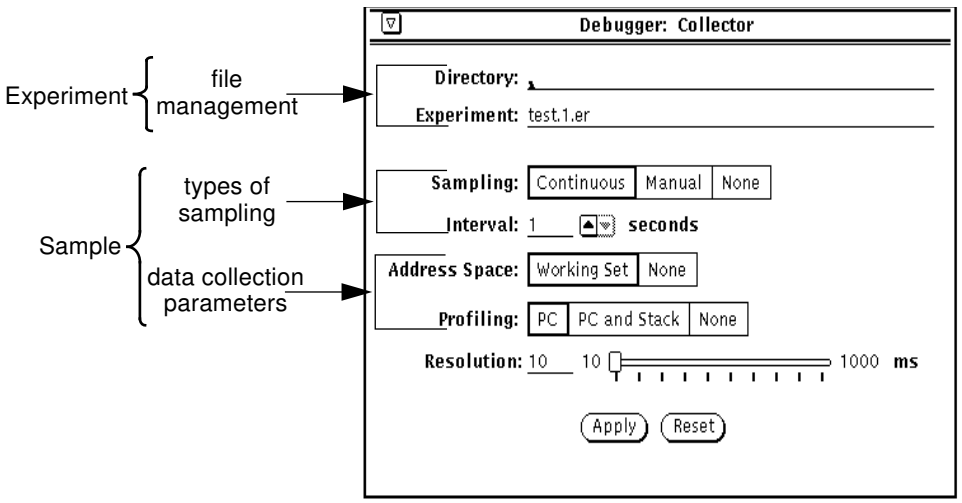


Figure 18-1 Collector Window

The Collector window (Figure 18-1) is displayed with default settings for experiments and samples. You need not change any of the settings or fields in the Collector window to start data collection. However, you can change any of these settings when the default settings do not satisfy your requirements. The settings can only be changed when the program you are debugging is stopped.

The Collector window consists of two parts: experiment and sample.

18.1.1 Experiment

An experiment consists of a set of samples taken on an application. In the experiment you execute the application, observing it with the Collector and storing the performance data into an experiment record. The performance data is collected in one or more samples, each of which is a measurement of a different period of time during the execution of the application. The experiment can contain one or more samples.

The experiment part of the Collector concerns itself with routine file management. You must name the experiment and specify a directory in which it is to be stored. See Figure 18-1.

18.1.2 Sample

A sample contains data collected over a specified period of time during the execution of the application. It provides information about how the application program behaves. You can take one sample or multiple samples.

Multiple samples are useful when the performance of the application varies over time. Instead of examining the aggregate behavior of the application's entire execution, you can take multiple samples to examine the performance behavior at specified intervals of the application's execution.

A sample may include data describing system calls, resource consumption, and referenced and modified pages in the application's address space

The Sample consists of two areas:

- Choosing the type of sampling.

  This area controls the samples that are taken. Each sample contains information on the target application over a specified period of time

- Choosing the types of data to collect.

  This area specifies the types of data to collect (collection parameters). Overview is the default data collection parameter.

18.2 Taking a Sample for Use with the Analyzer

Performance data is collected through the process of sampling. Each sample contains information on the target application over a specified period of time. Within the Collector, you can control the way in which samples are taken by choosing one of the sampling settings:

The default setting is Continuous samples every one second.

18.2.1 Manual Sampling

Manual sampling allows you to mark a sample boundary by issuing the Take Sample command from within the Debugger. Manual sampling with Take Sample is designed mainly so that you can collect data on real time, interactive events, such as displaying a window or choosing a menu item. The Manual sampling mode is especially useful if the application provides a visual response, either in the form of graphics or text.

For example, you can test the performance of an item on a menu as it completes its tasks. To perform this type of experiment, you perform the following list of operations:

1. Run the program in the Debugger; put the user interface into a typical state for choosing the menu item you want to test. Suspend the application.
2. Display the collector and set the sampling control to Manual. (Setting Sampling to Manual in the Collector window also activates the Take Sample item on the Debugger Execution menu.)
3. Continue the application running in the Debugger. *Collection of information starts as soon as the application is re-activated*.
4. Choose the menu item in the application.
5. When the application completes the task(s) invoked by choosing the item, choose Take Sample from the Debugger Execution menu. Take Sample marks the *endpoint* of that particular sample and the start of the next one. The data collected is from the time you started the Collector to the time you chose Take Sample.

To mark a sample boundary in Manual mode,

- **Choose Take Sample from the Execution menu.**

18.2.2 Storing the Collected Data

An experiment consists of a sample or set of samples taken on the application.

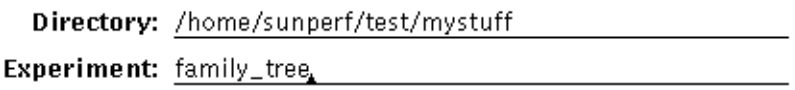


Figure 18-2 Experiment Area of the Collector Window

Experiment Directory

The experiment must be saved to a directory. Use the default directory name or create your own, more descriptive, directory name in which to save the experiment. To specify a different directory, enter the new name in the Experiment text field. Whether you use the default directory or specify another directory, when the Collector is first activated, the default directory is displayed in the Directory text field. If you delete the default directory name and forget to enter in a new name, then the experiment is automatically saved to the current working directory. You must enter a name before you can start data collection.

Experiment Name

All experiment names are derived from the application's name. For example, the experiment name for the Blocks is `blocks.1.er`. The collector assigns names automatically to each experiment, using suffixes and version tag numbers to make each name unique.

The Collector never overwrites an existing file. If there is another file by the same name, then the version tag is incremented each time you run the experiment. For example, if you run the experiment for `blocks.1.er` a total of three times, then the three experiments are named `blocks.1.er`, `blocks.2.er`, and `blocks.3.er`.

18

Runtime Checking 19

This feature is available only on SPARC systems.

Runtime Checking (RTC) allows you to automatically detect runtime errors in an application during the development phase.

This chapter is organized into the following sections:

Getting Started	*page 19-215*
Summary of Operation	*page 19-215*
Error Reporting	*page 19-219*
Memory Access Error Checking	*page 19-219*
Memory Access Errors	*page 19-220*
Using Memory Access Checking	*page 19-225*
Memory Leaks Checking	*page 19-227*
Memory Leak Errors	*page 19-229*
Error Suppression	*page 19-232*
Command Syntax	*page 19-237*
Running Commands from a Script	*page 19-242*
Functions Intercepted by Runtime Checking	*page 19-243*
Troubleshooting Tips	*page 19-243*

Because RTC is an integral feature of the Debugger, all functions of the Debugger, such as setting breakpoints, examining variables and so on, can be used with RTC. Compiling with the `-g` flag provides source line number correlation in the RTC error messages. RTC can also check programs compiled with the optimization `-O` flag.

No recompiling, relinking, or Makefile changes are required to take advantage of RTC.

The following list briefly describes the features of RTC:

- Detects memory access errors
- Detects memory leaks
- Works with languages other than C and C++
- Works on code that you do not have the source for, such as libraries

Requirements

Runtime Checking:

- Requires programs to be compiled using SPARCcompiler 3.0 (or later) compilers
- Requires dynamic linking with `libc`
- Requires use of the standard `libc malloc/free/realloc` or allocators based on those functions
- Requires programs that are not fully stripped. Programs stripped with `strip -x` are acceptable

Limitations

- Does not support attaching to a running process
- Does not do error checking on child processes of the debugged program
- Does not handle program text areas and data areas larger than 8Mb. A possible solution is to insert special files in the executable image to handle program text areas and data areas larger than 8Mb. See `help rtc8M` for more information
- Does not handle multithreaded programs

19.1 Getting Started

1. To use Runtime Checking with a command-line interface:

   Start dbx with the `-C` option:

   ```
   dbx -C yourprog
   ```

2. To use Runtime Checking with a graphical user interface:

   ```
   Start the Debugger with the -C option:

     debugger -C <yourprog> &
   ```

   Alternately, you can set the environment variable `SW_RTC`. This has the effect of specifying the `-C` option every time you invoke dbx or the Debugger:

   ```
   setenv SW_RTC                  # csh
   SW_RTC= ; export SW_RTC        # sh or ksh
   ```

Note – Using the `-C` option does not turn on checking, but enables you to turn it on from inside the Debugger.

   If you start dbx or the Debugger without the `-C` option or do not set the environment variable `SW_RTC`, and issue the `check` command, dbx checking does not enable error checking until you issue the next `run` command. However, all the shared libraries needed by the program are reloaded. A warning to that effect is also issued.

3. To use Runtime Checking in batch mode without invoking dbx:

   `bcheck [ -access | -all | -leaks] -o` *logfile* `-s` *script program[args]*

   `bcheck(1)` is a batch interface to the RTC feature of `dbx` (1). Its default action is to perform leaks checking only. See Section 19.2.3, “Batch Mode Operation (bcheck),” on page 19-218 for more information. Refer to the `bcheck` (1) man page for details on its use.

19.2 Summary of Operation

This section describes how to operate RTC from dbx, the Debugger, and in batch mode.

19.2.1 Operation from dbx

You must turn on the type of checking you want before you run the program.

To turn on memory leaks checking only, type

```
(dbx) check -leaks
```

To turn on memory access checking only, type

```
(dbx) check -access
```

To turn on both memory leaks checking *and* memory access checking, type

```
(dbx) check -all
```

To turn off memory leaks checking and memory access checking, type

```
(dbx) uncheck -all
```

Run the program being tested, with or without breakpoints. Here is a simple example showing memory access checking being turned on for a program called `hello`.

```
speedracer$ dbx -C hello
Reading symbolic information for hello
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for
/opt/SUNWspro/bin/../SW3.0/bin/../lib/librtc.so
Reading symbolic information for /usr/lib/libc.so.1
Reading symbolic information for /usr/lib/libintl.so.1
Reading symbolic information for /usr/lib/libdl.so.1
Reading symbolic information for /usr/lib/libw.so.1
(dbx) check -access
access checking - ON
(dbx) run
Running: hello
(process id 14582)
Enabling Error Checking... done
hello
execution completed, exit code is 1
```

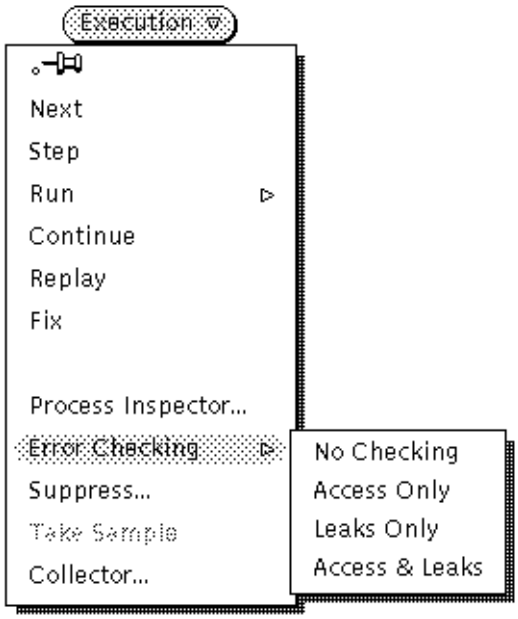
If access checking is on, the program runs as it normally would, except that it runs slower because each memory access is checked for validity. Each memory access is checked just before it actually occurs. If an invalid access is detected, an error is displayed, giving specific information about the error. The program is then suspended and control is returned to you.

You can then issue dbx commands, such as `where`, to get the current stack trace or `print` to examine variables. If the error is not a fatal error, you can continue execution of the program with the `cont` command. See Section 19.13, "Troubleshooting Tips," on page 19-243. The program continues to the next error or breakpoint, whichever is detected first.

19.2.2 Operation from the Debugger

To turn on the desired checking mode from the Debugger window:

- **Choose Error Checking from the Execution menu, pull right and make your selection.**



- **Select Run from the Execution menu or click on the Run button to start the program.**

19.2.3 Batch Mode Operation (bcheck)

bcheck(1) is a convenient batch interface to the RTC feature of dbx. It runs *program* under dbx, with the arguments specified by *args*, if any. By default, the RTC error output is placed in the default file program.errs.

You can use the -o *filename* option to specify a different name for the logfile. You use the -s *script* option before executing the program to read in the dbx commands contained in the file *script*. The *script* file typically contains commands like suppress and dbxenv to tailor the error output of bcheck.

bcheck can perform memory leaks checking, memory access checking, or both. Its default action is to perform leaks checking only.

Note – When using the bcheck utility, the default setting for dbxenv autosuppress is on. This setting eliminates redundancy in error reporting by only reporting unique memory access errors.

Examples:

Perform leaks checking only on hello:

```
bcheck hello
```

Perform access checking only on mach with the argument 5:

```
bcheck -access mach 5
```

Perform both access checking and leaks checking on cc:

```
bcheck -all cc -c prog.c
```

You can also enable batch mode directly from within dbx by setting the following dbx variable:

```
(dbx) dbxenv autocontinue on
(dbx) dbxenv errlogfile your_logfile_name
```

The program does not stop when runtime errors are detected in batch mode. All error output is redirected to your error log file *your_logfile_name*. But the program stops when breakpoints are encountered or if the program is interrupted.

In batch mode, the complete stack backtrace is generated and redirected to the errlog file. The number of stack frames can be controlled using the `dbxenv` variable `stackmaxsize`.

Note – If the file *your_logfile_name* already exists, `bcheck` erases the contents of that file before it redirects the batch output to that file.

You can also enable batch mode from within the Debugger:

- **Click on Props to display the Property sheet.**
- **Select Category➤Run Time Check.**
- **Select Autocontinue➤ On.**
- **Enter the log file name in Error Log Filename.**

19.3 Error Reporting

When RTC detects an error, it reports the type and location of the error and returns control to the user unless **autocontinue** is set to **On**. If the Debugger GUI is present, RTC shows the location of the error in the source code display.

You can narrow the scope of checking by suppressing a particular *error type* and expand the scope by unsuppressing that *error type* later.

You can perform any of the usual debugging activities; such as setting breakpoints and examining variables. Continue runs the program until the program encounters another error or a breakpoint, or until it terminates.

19.4 Memory Access Error Checking

RTC checks whether your program accesses memory correctly by monitoring each read, write, and memory free operation.

RTC inserts checkpoints in front of each `ld` or `st` instruction in your program.

A checkpoint leaves program behavior unchanged when a memory access is valid, but acts as a breakpoint when a memory access is invalid. In the case of an invalid memory access, RTC stops on the breakpoint and executes the invalid `ld` or `st` instruction only after you continue the program.

Note – `ld` includes all unprivileged forms of the `ld` instruction: `ldsb`, `ldsh`, `ldub`, `lduh`, `ld`, `ldd`, and `swap`,. Similarly, `st` includes all unprivileged forms of the `st` instruction: `stsb`, `stsh`, `stub`, `stuh`, `st`, and `std`.

RTC maintains a table that tracks the state of each block of memory being used by the program. When the program performs a memory operation, RTC checks the operation against the state of the block of memory it involves, to determine whether the operation is valid. The possible memory states are:

- Unallocated—initial state. Memory has not been allocated. It is illegal to read, write, or free this memory because it is not owned by the program.
- Allocated, but uninitialized. Memory has been allocated to the program but not initialized. It is legal to write to or free this memory, but is illegal to read it because it is uninitialized. For example, upon entering a function, stack memory for local variables is allocated, but uninitialized.
- Read-only. It is legal to read, but not write or free, read-only memory.
- Allocated and initialized. It is legal to read, write, or free allocated and initialized memory.

19.5 Memory Access Errors

RTC detects the following memory access errors:

- Read from uninitialized memory (rui)
- Read from unallocated memory (rua)
- Write to unallocated memory (wua)
- Write to read-only memory (wro)
- Misaligned read (mar)
- Misaligned write (maw)
- Duplicate free (duf)
- Bad free (baf)
- Misaligned free (maf)
- Out of memory (oom)

19.5.1 Memory Access Error Reporting

RTC prints the following information for memory access errors:

type	Type of error.
access	Type of access attempted (read or write).
size	Size of attempted access.
addr	Address of attempted access.
detail	More detailed information about addr. For example, if addr is in the vicinity of the stack, then its position relative to the current stack pointer is given. If addr is in the heap, then the address, size, and relative position of the nearest heap block is given.
stack	Call stack at time of error.
location	Where the error occurred. If line number information is available, this information includes *filename*, *line number*, and *function*. If line numbers are not available, RTC provides *function* and *address*.

The following examples show typical access error reports:

```
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff67c
    which is 1268 bytes above the current stack pointer
Variable is 'i'
Current function is main
    30      j = i
```

```
Read from unallocated (rua):
Attempting to read 4 bytes at address 0x22368
    which is just past heap block of size 16 bytes at 0x22358
Current function is read_out_of_bound
   27    return ptr[index];
```

The following discussion provides a more detailed description of each memory access error checked by RTC:

19.5.2 Read from Uninitialized Memory (rui)

Problem: Attempt to read from uninitialized memory.
Possible causes: Reading local or heap data which has not been initialized.

Example:
`foo()`
`{  int i, j;`
`   j = i;  /* Read from uninitialized memory (rui) */`
`}`

19.5.3 Read from Unallocated Memory (rua)

Problem: Attempt to read from non-existent, unallocated, or unmapped memory.
Possible causes: A stray pointer, overflowing the bounds of a heap block, or accessing a heap block that has already been freed.

Example:
`char c, *a = (char *)malloc(1);`
`c = a[1];  /* Read from unallocated memory (rua) */`

19.5.4 Write to Unallocated Memory (wua)

Problem: Attempt to write to non-existent, unallocated, or unmapped memory.
Possible causes: A stray pointer, overflowing the bounds of a heap block, or accessing a heap block that has already been freed.

Example:
`char *a = (char *)malloc(1);`
`a[1] = '\0';  /* Write to unallocated memory (wua) */`

19.5.5 Write to Read-Only Memory (wro)

Problem: Attempt to write to read-only memory.
Possible causes: Writing to a text address, writing to a read-only data section (.rodata), or writing to a page that has been mmap'ed as read-only.

Example:
`foo()`
`{  int *foop = (int *) foo;`
`   *foop = 0;   /* Write to read-only memory (wro) */`
`}`

19.5.6 Misaligned Read (mar)

Problem: Attempt to read data from an address without proper alignment.
Possible causes: Reading 2, 4, or 8 bytes from an address which is not half-word-aligned, word-aligned, or double-word-aligned, respectively.

Example:
`char *s = "hello world";`
`int *i = (int *)&s[1];`
`int j;`
 	
`j = *i;  /* Misaligned read (mar) */`

19.5.7 Misaligned Write (maw)

Problem: Attempt to write data to an address without proper alignment.
Possible causes: Writing 2, 4, or 8 bytes to an address which is not half-word-aligned, word-aligned, or double-word-aligned, respectively.

Example:
`char *s = "hello world";`
`int *i = (int *)&s[1];`
 	
`*i = 0;  /* Misaligned write (maw) */`

19.5.8 Duplicate Free (`duf`)

Problem: Attempt to free a heap block that has already been freed.
Possible causes: Calling `free`() more than once with the same pointer. In C++, using the "delete" operator more than once on the same pointer.

Example:
`char *a = (char *)malloc(1);`
`free(a);`
`free(a);  /* Duplicate free (duf) */`

19.5.9 Bad Free (`baf`)

Problem: Attempt to free memory that has never been allocated.
Possible causes: Passing a non-heap data pointer to `free`() or `realloc()`.

Example:
`char a[4];`
`char *b = &a[0];`
 	
`free(b);  /* Bad free (baf) */`

Problem: Attempt to free a NULL pointer.
Possible causes: Passing a NULL pointer to `free`().

Example:
`char *b = 0;`
 	
`free(b);  /* Free NULL ptr (baf) */`

19.5.10 Misaligned Free (maf)

Problem: Attempt to free a misaligned heap block.
Possible causes: Passing an improperly aligned pointer to `free`() or `realloc()`; changing the pointer returned by `malloc`.

Example:
`char *ptr = (char *)malloc(4);`
`ptr++;`
`free(ptr);  /* Misaligned free */`

19.5.11 Out of Memory (oom)

Problem: Attempt to allocate memory beyond physical memory available.
Cause: Program cannot obtain more memory from the system. The **oom** error is useful in tracking down problems that occur when the return value from `malloc()` is not checked for NULL, which is a common programming mistake.

Example:
`char *ptr = (char *)malloc(0x7fffffff);`
`/* Out of Memory (oom), ptr == NULL */`

19.6 Using Memory Access Checking

The simplest way to find access errors is to turn on access checking and run the program to produce a list of access errors.

Using RTC to find memory access errors is not unlike using a compiler to find syntax errors in your program. In both cases a list of errors is produced, with each error message giving the cause of the error and the location in the program where the error occurred. Also in both cases, you should fix the errors in your program starting at the top of the error list and working your way down. The reason is that one error can cause other errors in a sort of chain reaction. The first error in the chain is therefore the "first cause", and fixing that error may also fix some subsequent errors. For example, a read from an

uninitialized section of memory can create an incorrect pointer, which when dereferenced can cause another invalid read or write, which can in turn lead to yet another error, and so on.

Larger Programs

The simple strategy described above can be used for small- to medium-sized programs because the number of errors is usually relatively small. For the initial run on a large program however, the number of errors may be so large as to be overwhelming. In this case, it may be better to take a phased approach. This can be done by using the `suppress` command or the dbxenv variable, `dbxenv autosuppress on` to reduce the reported errors to a manageable number, fixing just those errors, and repeating the cycle; suppressing fewer and fewer errors with each iteration.

For example, you could focus on a few error types at one time. The most common error types typically encountered are **rui**, **rua**, and **wua**, usually in that order. **rui** errors are less serious errors (although they can cause more serious errors to happen later), and often a program may still work correctly with these errors. **rua** and **wua** errors are more serious because they are accesses to or from invalid memory addresses, and always indicate a coding error of some sort.

Suppressing Some Errors

You could start by suppressing **rui** and **rua** errors. After fixing all the **wua** errors that occur, run the program again, this time suppressing only **rui** errors. After fixing all the **rua** errors that occur, run the program again, this time with no errors suppressed. Fix all the **rui** errors. Lastly, run the program a final time to ensure there are no errors left.

If you want to suppress the last reported error, use `suppress -last`. You also can limit the number of errors reported without using the `suppression` command by using `dbxenv error_limit` *n* instead. For more information on error suppression, see Section 19.9, “Error Suppression,” on page 19-232.

When to Use RTC

Another way to avoid seeing a large number of errors at once is to use RTC earlier in the development cycle, as you are developing the individual modules that make up the program. Write a unit test to drive each module and use RTC incrementally to check each module one at a time. That way, you deal with a

smaller number of errors at a time. When you integrate all of the modules into the full program, you are likely to encounter few new errors. When you reduce the number of errors to zero, you need to run RTC again only when you make changes to a module.

19.7 Memory Leaks Checking

Memory Leaks

A memory leak is a dynamically allocated block of memory that has no pointers pointing to it anywhere in the data space of the program. Such blocks are orphaned memory. Because there are no pointers to the blocks, the program cannot even reference them, much less free them. RTC finds and reports such blocks.

Sometimes, the term "memory leak" is used to refer to *any* block that has not been freed. This is a much less useful definition of a memory leak, because it is a common programming practice not to free memory if the program will terminate shortly anyway. RTC does not report a block as a leak if the program still retains one or more pointers to it.

Possible Leaks

There are two cases where RTC may report a "possible" leak. The first case is when no pointers were found pointing to the beginning of the block, but a pointer found pointing to the *interior* of the block. This case is reported as an "Address in Block (aib)" error. If it was a stray pointer that happened to point into the block, this would be a real memory leak. However, some programs deliberately move the only pointer to an array back and forth as needed to access its entries. In this case it would not be a memory leak. Because RTC cannot distinguish these two cases, it reports them as Possible Leaks, allowing the user to make the determination.

The second type of Possible Leak is when no pointers to a block were found in the data space, but a pointer was found in a register. This case is reported as an "Address in Register (air)" error. If the register happens to point to the block accidentally, or if it is an old copy of a memory pointer that has since been lost, then this is a real leak. However, the compiler can optimize references and place the only pointer to a block in a register without ever writing the pointer

to memory. In such cases, this would not be a real leak. Hence, if the program has been optimized *and* the report was the result of the `showleaks` command, it is likely not to be a real leak. In all other cases, it is likely to be a real leak.

To suppress all reports of Possible Leaks, type: `suppress air aib`

Checking for Leaks

If memory leaks checking is turned on, a scan for memory leaks is automatically performed just before the program being tested exits. Any detected leaks are reported. The program should not be killed with the `kill` command. Here is a typical memory leak error message:

```
Memory leak (mel):
Found leaked block at address 0x21718 of size 4
At time of allocation, the call stack was:
[1] foo() at line 63 in test.c
[2] main() at line 47 in test.c
```

To check for leaks in the middle of execution, use the `showleaks` command. The `showleaks` command shows any new leaks since the last `showleaks` command. Showleaks works only when the `check -leaks` command is given before the program starts. After reporting the leaks, if any, a leaks summary is given.

Any particular memory leak is reported only once. If a `showleaks` command is issued, any leaks found are *not* reported again, either by another `showleaks` command, or the automatic leaks scan performed just before the program exits.

UNIX programs have a `main` procedure (called `MAIN` in f77) which is the top-level user function for the program. Normally, a program terminates either by calling `exit(3)` or by simply returning from `main`. In the latter case, all variables local to `main` go out of scope after the return, and any unique heap blocks they pointed to are reported as leaks.

It is a common programming practice not to free heap blocks allocated to local variables in `main`, because the program is about to terminate anyway, and then return from `main` without calling (`exit()`). To prevent RTC from reporting such blocks as memory leaks, stop the program just before `main` returns by setting a breakpoint on the last executable source line in `main`. When the

program halts there, use the RTC `showleaks` command to report all the true leaks in `main`, omitting the leaks that would result merely from `main`'s variables going out of scope.

19.8 Memory Leak Errors

Note – RTC only finds leaks of `malloc` memory. If your program does not use `malloc`, RTC cannot find memory leaks.

RTC detects the following memory leak errors:

- Memory Leak (`mel`)
- Possible leak — Address in Register (`air`)
- Possible leak — Address in Block (`aib`)

19.8.1 Memory Leak Error Reporting

RTC prints the following information for individual memory leak errors:

location	location where leaked block was allocated
addr	address of leaked block
size	size of leaked block
stack	at least *stackdepth* preceding functions in the call stack at time of allocation.

Because the number of individual leaks can be very large, RTC automatically combines leaks that were allocated at the same place into a single combined leak report. The decision to combine leaks, or report them individually, is contr