Debugging a Program

SunSoft
A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A

Part No.: 802-3517-10
Revision A, November 1995

© 1995 Sun Microsystems, Inc. 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution and decompilation. No part of this product or document 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® system and from the Berkeley 4.3 BSD system, licensed from the University of California. Third-party software, including font technology in this product, is protected by copyright and licensed from Sun's Suppliers.
RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 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, the Sun logo, Sun Microsystems, Sun Microsystems Computer Corporation, Solaris, the Sun Microsystems Computer Corporation logo, SunSoft, the SunSoft logo, ProWorks, ProWorks / TeamWare, ProCompiler, Sun-4, SunOS, 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 United States and may be protected as trademarks in other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X / Open Company, Ltd. OPEN LOOK is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. PowerPC™ is a trademark of International Business Machines Corporation. HP ® and HP-UX ® are registered trademarks of Hewlett-Packard Company. 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. in the United States and may be protected as trademarks in other countries. SPARCcenter, SPARCcluster, SPARCompiler, SPARCdesign, SPARC811, SPARCengine, SPARCprinter, SPARCserver, SPARCstation, SPARCstorage, SPARCworks, microSPARC, microSPARC-II, SPARCware, and UltraSPARC 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 GUI's and otherwise comply with Sun's written license agreements.

X Window System is a product of X Consortium, Inc.

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

Adobe PostScript

Contents

Part 4 — Appendixes

Preface

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

Before You Begin

The following restrictions apply:

The notation **Solaris** means that the feature is only supported for the Solaris Operating System.

The notation **Solaris 2.x** means that the feature is only supported for the Solaris 2.x Operating System.

This manual is written for application developers who want to use Debugger to aid in application development. See the appropriate README, `sparcworks or proworks,` for a description of operating environment requirements in which Debugger runs.

This manual assumes you are familiar with

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

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 events take place in the program being debugged.

- Solaris 2.x **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.

Solaris 2.x
- **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.

Solaris 2.x
- **Chapter 19, "Runtime Checking,"** describes how to automatically detect run time errors in an application during the development phase.

Solaris 2.x
- **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 22, "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.

Solaris 2.x
- **Chapter 24, "dbx and the Solaris Dynamic Linker,"** describes use of the debugger with dynamically-linked, shared libraries.

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 a table that describe operators, their precedence and associativity.
- **Appendix C, "Quick Start: Debugging Fortran 77,"** introduces some features that are useful in FORTRAN 77.
- **Appendix D, "Quick Start: Debugging Fortran 90,"** introduces some features that are useful in FORTRAN 90.

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:

Solaris 2.x

- **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)
  dbxinit(4)
  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 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.

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:

1.1 Solaris 2.x , Solaris 1.x, and HP-UX

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.

Solaris — 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

NOTE Throughout this manual, margin notations denote which operating environments are supported for various features and functions. If the margin note Solaris appears, it means that both Solaris 1.x and Solaris 2.x environments are supported. If the margin note Solaris 2.x appears, it means that only Solaris 2.x environments are supported. If the margin note PowerPC appears, it means that PowerPC is also supported in addition to Solaris 1.x and Solaris 2.x. Generally, unless the text specifically states that HP-UX is not supported, the implication is that HP-UX is supported.

Here is a list of the Debugger differences between the Solaris 2.x environment, the Solaris 1.x environment, and the HP-UX 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 2-20 for more details.
- The catch and ignore signal lists differ for each version.
- Features supported only on Solaris 2.x and HP-UX:
  - 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.
  - Solaris — 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 2-20 for details.

With Solaris 1.x and HP-UX 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/ProWorks 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:

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

- *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).
- *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.
- Solaris — Use *Run Time Checking* (RTC) to automatically detect run time errors in an application. RTC detects memory access errors and memory leaks and also provides memory profiling..
- *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.
- Solaris — 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 programs* as they `fork` (2) and `exec` (2).

1.3.3 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 2-22.

1.3.4 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.

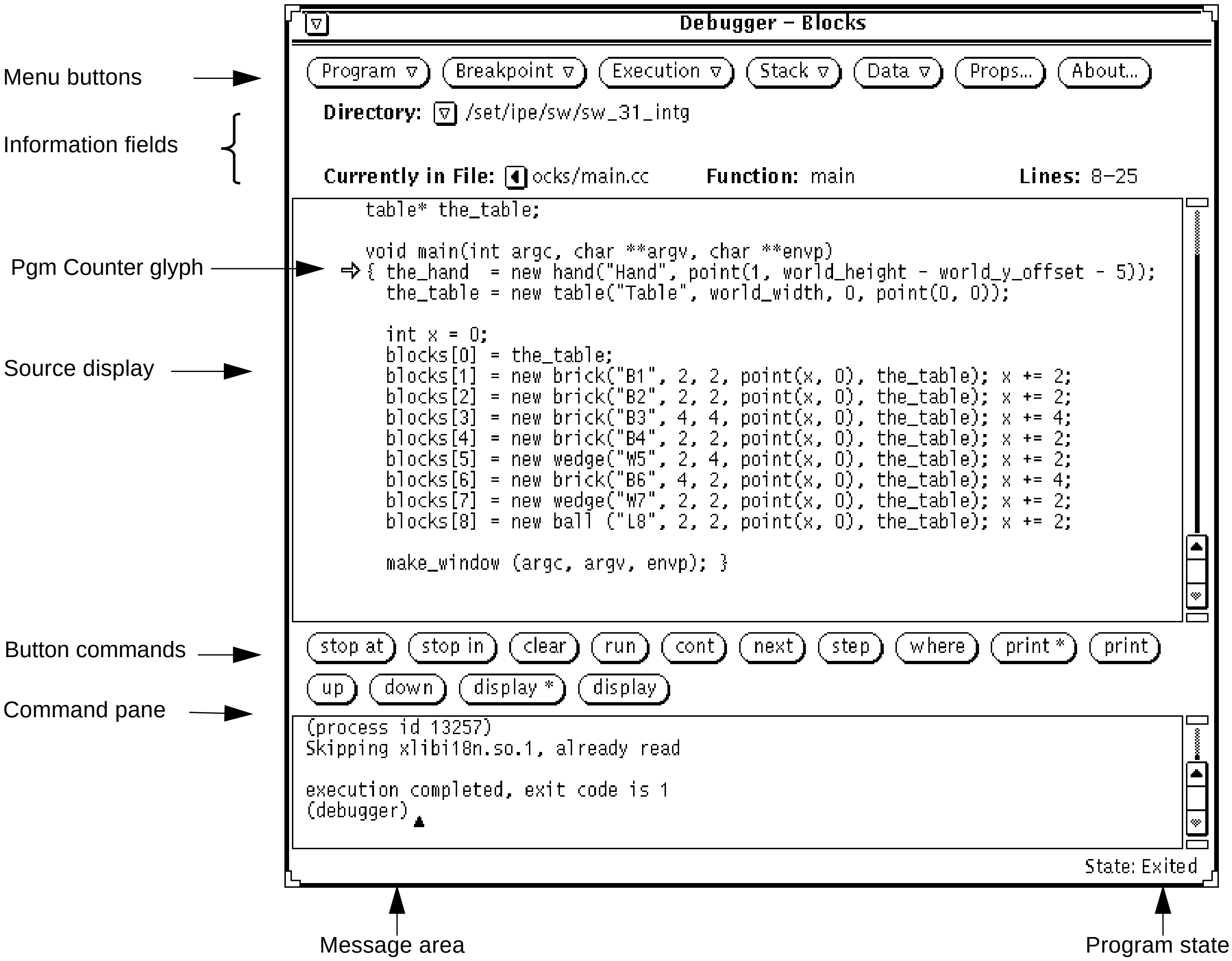
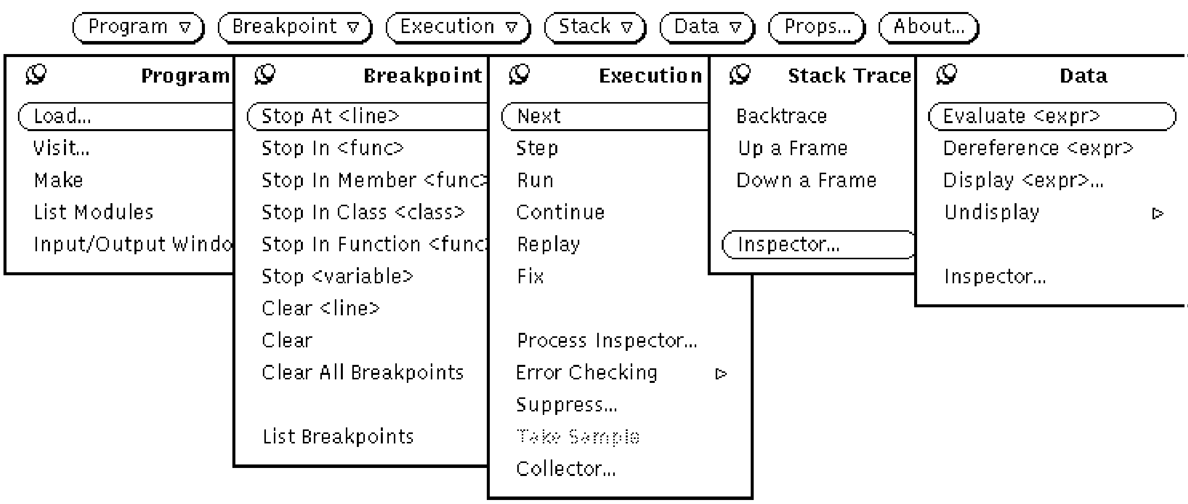


Figure 1-1 Debugger Base Window

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," on page 1-15. The About button displays a window that briefly describes the Debugger tool and contains a "Comments" mechanism that you can use to return your comments to SunSoft. 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

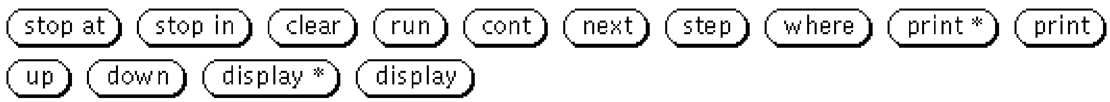
The information fields tell you where the program is stopped, what code is displayed in the source display, and the name of the current function. See Chapter 4, "Viewing and Visiting Code," for a detailed description of how to read the information presented in the base window.

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 4-47 for details.

Shell Commands for the Command Pane

The Debugger supports a number of primitive commands for debugging. If `set -o path` (the default) 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` Changes working directory.

`pwd` Prints working directory.

`use` Sets path for the Debugger to search for source files (debugging specific).

`search` Searches for a pattern matching *strings* in the current file (debugging specific).

`bsearch` Searches backwards for a pattern matching *strings* in the current file (debugging specific).

`dalias` Makes an alias for a debugging command.

`history` Prints a list of the most recent debugging commands (default 15), type `history` *n* to change default to *n*.

`kill` Kills the process loaded in the Debugger, but not the Debugger.

`quit` Terminates the session; detach the process if it was attached.

`shell_cmd` Issues 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*.

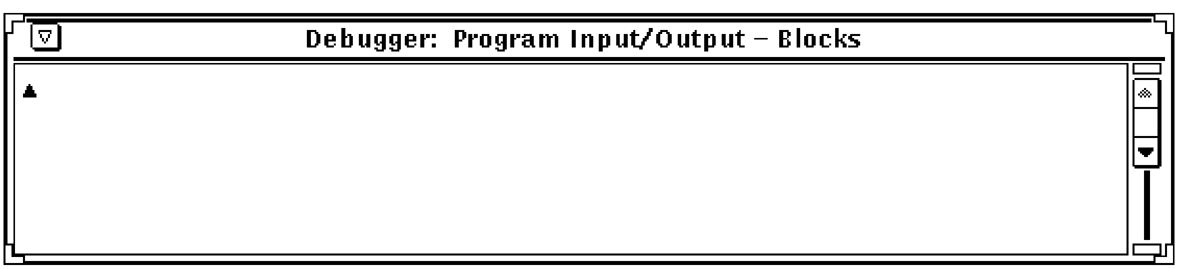
Note – If a program is running in the Debugger, Cntl-C (`^c`) interrupts the program, returning control to the Debugger.

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

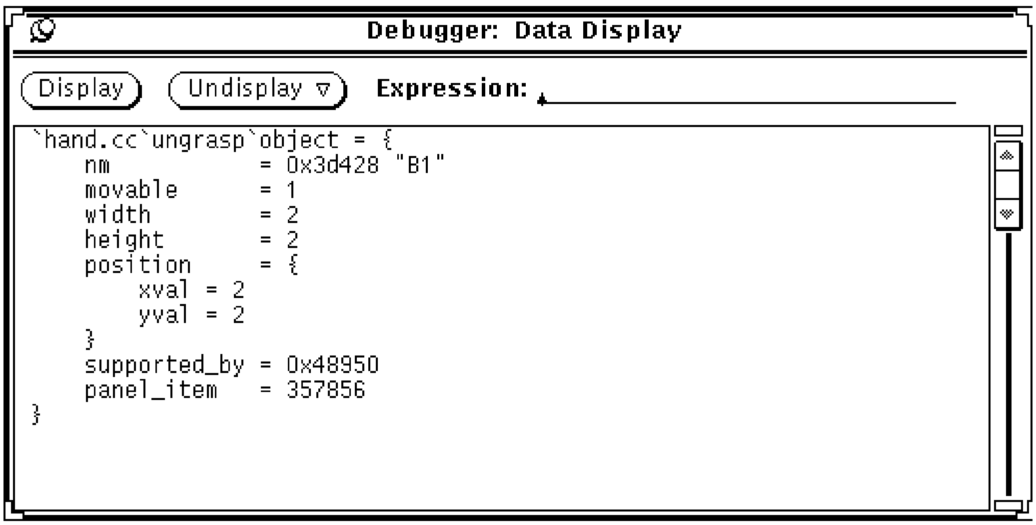
1.4.6 Program Input/Output Window (PIO)

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.

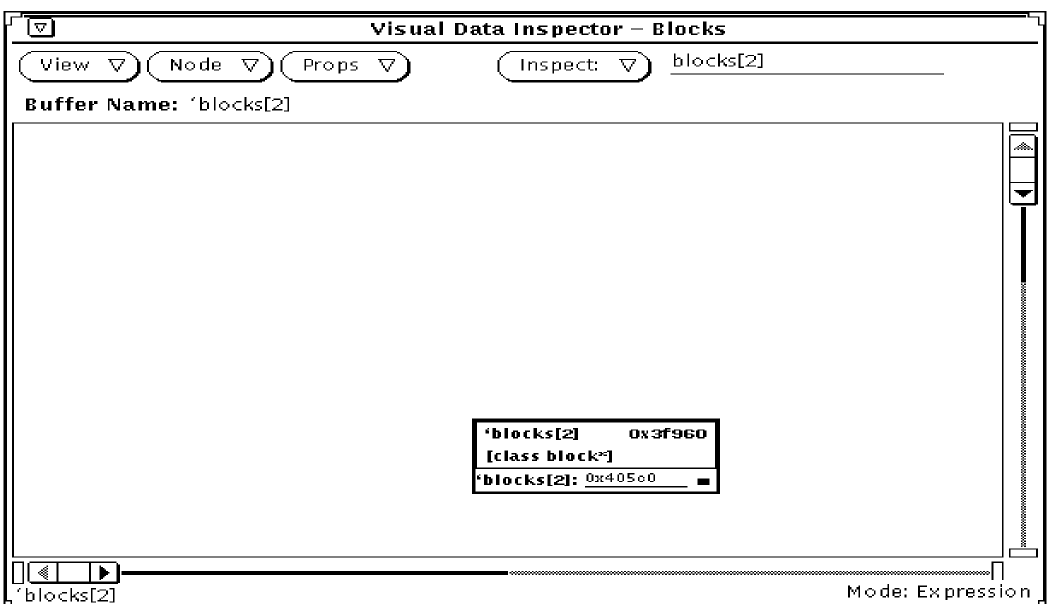
To move the PIO window to the front of the workspace, click on the Input/Output Window item on the Program menu.



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:

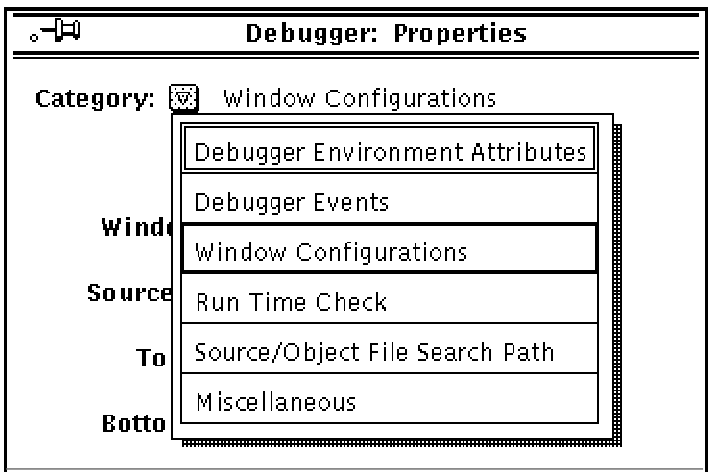


Here is the Visual Data Inspector window Solaris 2.x:.



1.4.7 Debugger Properties Window

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:

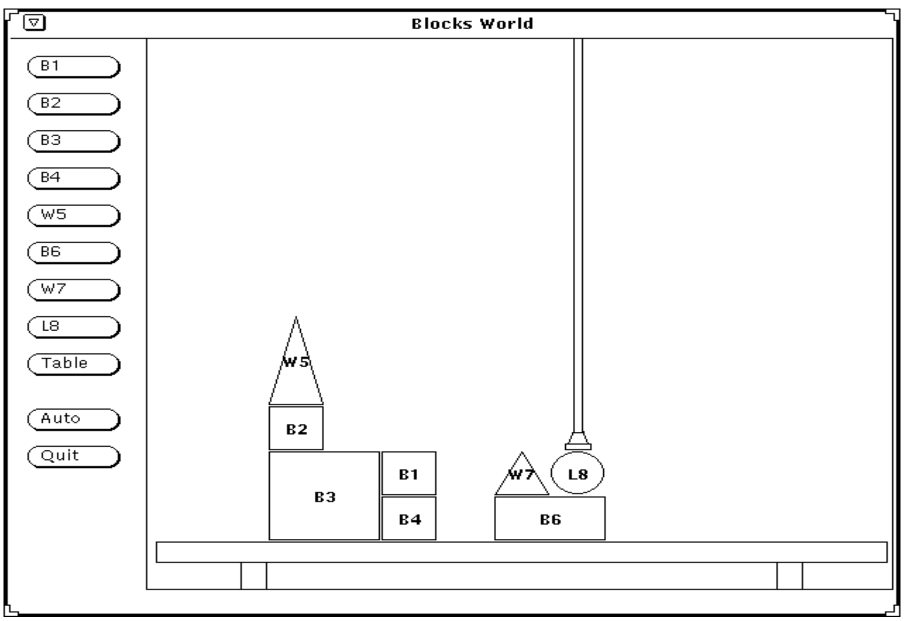


1.4.8 The Debugger Initialization Files

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 2-19.

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.

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 the* `-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 can debug programs that have been completely stripped on Solaris 2.x. These programs contain some information that can be used to debug your program, but only externally visible functions are available. Runtime Checking cannot work on stripped programs or load objects.

2.2 Using the Auto-Read Facility

Solaris

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 -c` 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 and HP-UX, 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 3-30, 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 -c` 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 , “Listing and Reading-in Program Modules,” on page 3-33 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` (uppercase 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:

Note – These start-up instructions assume that the two component Debugger programs, `dbx` and `debugger`, are correctly installed on your system.

3.1 Starting a Debugging Session

Start the Debugger from

- The 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` **or** `proworks` **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.**

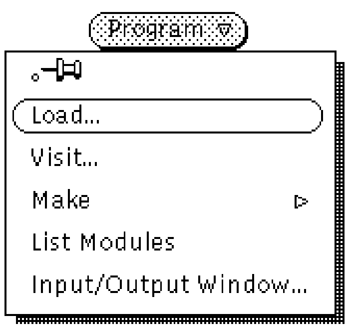
Program Loader

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

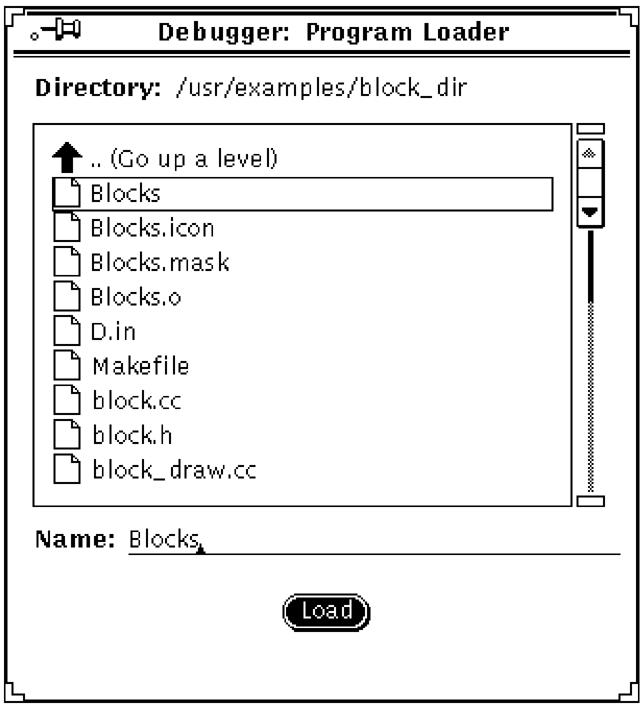
Use the Program Loader as a file chooser to select a program executable to load into the Debugger. Also, use the Program Loader to load a new program into the Debugger (replacing one already loaded).

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 3.3, “Options to the Debugger Start-up Command,” on page 3-35.

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.

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 3-29, 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, C++, or FORTRAN90: `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:

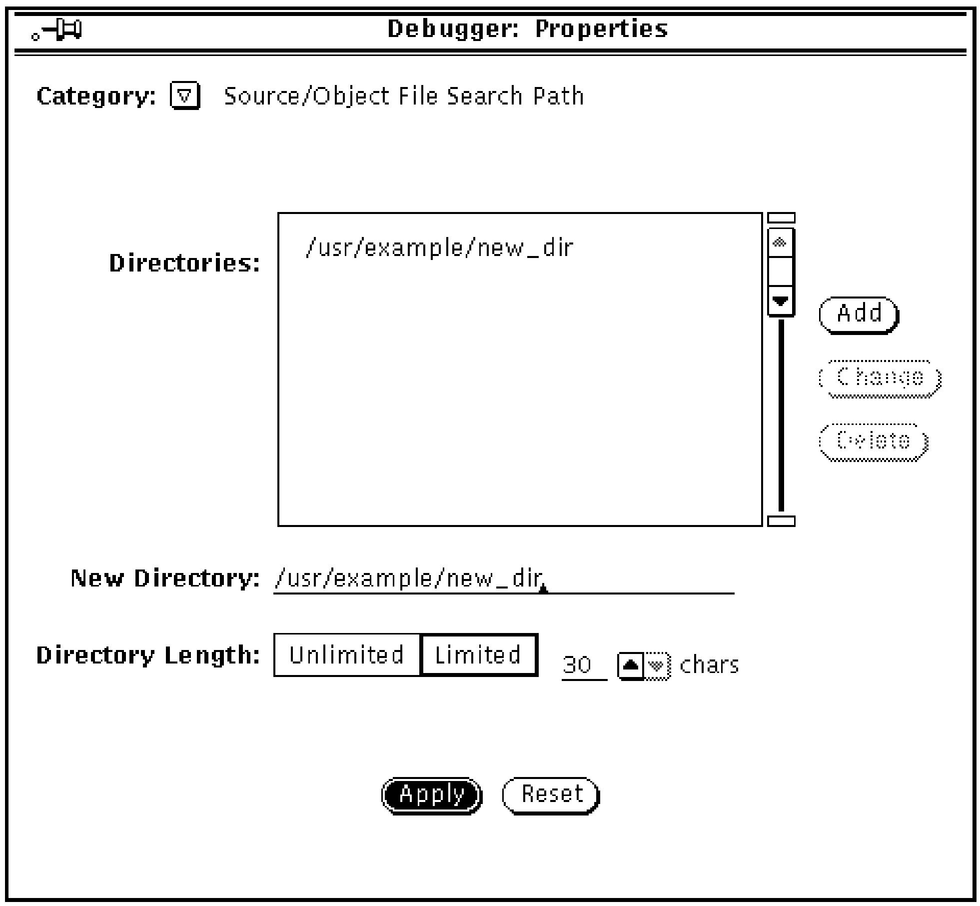
% **dbx - pid** // dash space *process_id*

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.**
2. **Click Add.**
   The directory name is displayed in the Directories display box.
3. **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.**
4. **Click Apply.**

To delete a path from the search path list:

1. **Select a pathname in the Directories box.**
2. **Click Delete.**
3. **Click Apply.**

3.2.1 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.

3.2.2 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.

3.2.3 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 directory *from* to directory *to* 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
```

Listing and Reading-in Program Modules

By default on Solaris 2.x, the compilers leave debugging information for each module compiled using the `-g -c` 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 2-20.)

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.

3.3 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 the Start-up Command

Option	Description
-help	Display a brief summary of these debugging options.
Remote Display	
-display *hostname*:0.0	Display the Debugger on a remote display; *hostname* is the name of the remote machine.
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	
`-c` *"dbx_cmd; ...;"*	Executes *dbx_commands* after initialization. Use quotation marks around debugging commands; for example, `%` **`debugger -c "stop in main" demo`**
-e	Echoes all input commands on standard out
`–I` *dir*	Adds *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 or Pathmap.
`d-kbd`	Debugs a program that sets the keyboard into up-down translation mode. This flag is necessary if a program uses up-down decoding.
-r	Loads and then runs 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.

Option	Description
-q	Silences the echoing of the "`Reading symbol table for …`" message and the "`Attached to…`" message.
`–s` *start-up*	Reads initialization commands from the file named *start-up*, not from the `.dbxrc` file.
`-sr` *start-up*	Reads and deletes "`start-up`".
`-wskip` *n*	Causes the `where` command to skip the first *n* stack frames.
-C	Enables Run Time Checking (Solaris only).

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:

4.1 Knowing the State of the Program

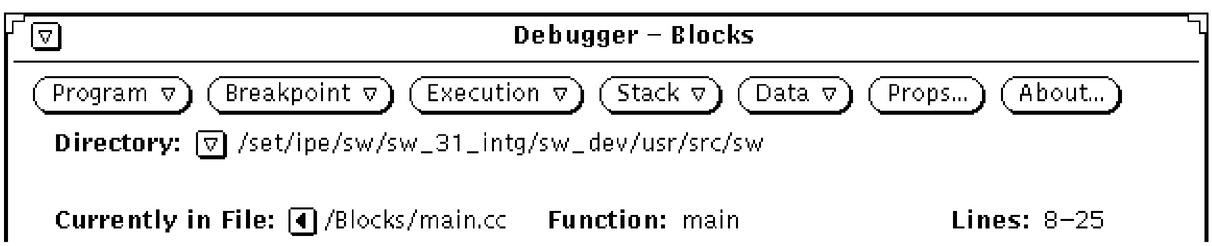
The Debugger reports the state of the program in the lower right message area of the base window.

The program may be in one of the following states:

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

4.2 Reading the Information Fields

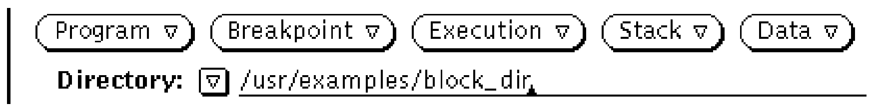
The Debugger displays three information fields beneath the menu buttons: Directory, Stopped in (File, Function, Line), and Currently in (File, Function, Lines).



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 3-30, and “Scope Resolution Search Path” on page 4-50.)

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.

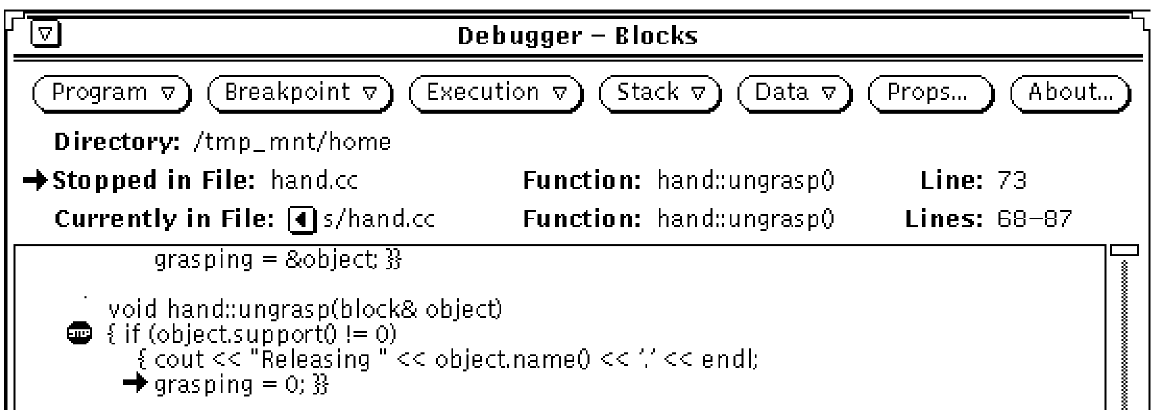


Figure 4-1 Stopped in File-Function-Line Information Field

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:

Stopped in File: hand.cc	**Function:** hand::animate	**Line:** 26
Currently in File: hand.cc	**Function:** hand::draw	**Lines:** 3–22

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 4-47, for details.

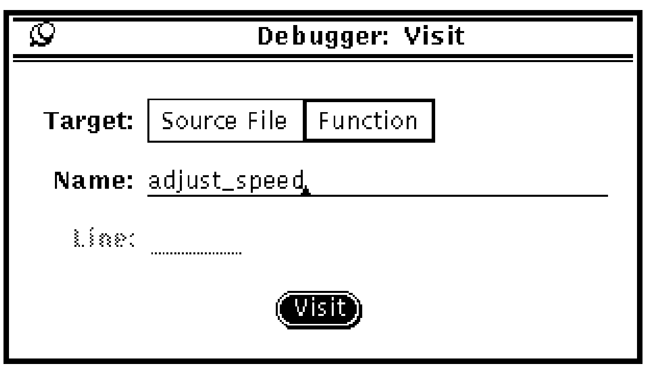
4.3 Visiting Code

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:

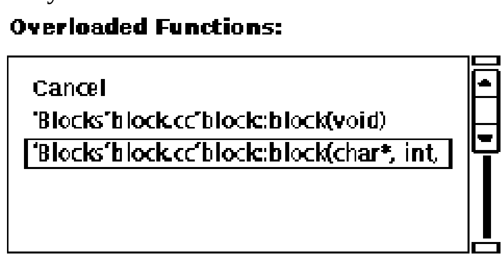
```
(debugger) func block::block
```

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.**

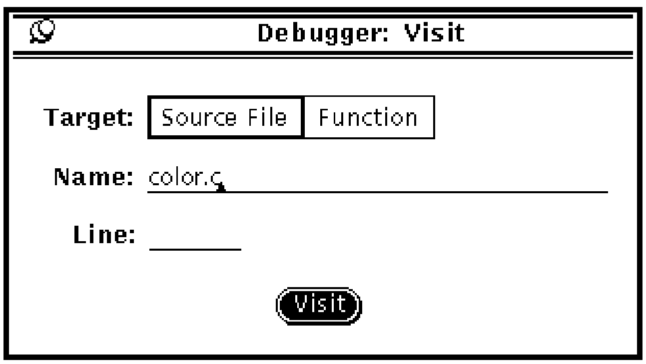
Note – Redisplaying the stop location by clicking on the arrow does *not* change the current function. The Debugger changes the current function to match the stop location only after the program executes lines of code and stops again.

4.3.4 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.

4.3.5 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 *line in the source pane*. 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 4-44.) 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:

Find a variable of global scope:

‘*global*

Find a function in a particulat loadobject:

‘*load.object_name*‘ *func*

Find a variable in a specific file and function

‘*source_file_name*‘ *function_name*‘*variable_name*

Search for a block local variable:

‘*source_lib_name*‘ *function_name:line_number*`*name*

Search for file static variable or function name:

‘*source_file_name*‘*name*

Search for file statics in all files

`` *name*

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.oanimate.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 4-44.)

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

♦ **Choose 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
forward:  `Blocks`block_draw.cc`table
function: `Blocks`block.cc`table::table(char*, int, int,
const point&)
class:  `Blocks`block.cc`table
class:  `Blocks`main.cc`table
variable:       `libc.so.1`hsearch.c`table
```

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 – (Solaris 2.x only) 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(char*, int, int, const point&,
load_bearing_block*)

(debugger) which draw

`block_draw.cc`wedge::draw(unsigned long)
```

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 4-50.) 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
`Blocks`block.cc`block::block(char*, int, int, const
point&,load_bearing_block*)
```

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 4.6.4, “Using whatis to See Inherited Members,” on page 4-56 for a description of *-r*.

4.6.1 Displaying Declarations of Variables and Members

Printing 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
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
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

Printing the Declaration of a Type or C++ Class

♦ **Type in the command pane**

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

4.6.3 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;
};
```

4.6.4 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.

4.6.5 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 {
public:
    table::table(char *name, int w, int h, const class point
&pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
(debugger)
```

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

```
(debugger) whatis -t -r class table
class table : public load_bearing_block {
public:
    /* from base class table::load_bearing_block::block */
    block::block();
   block::block(char *name, int w, int h, const class point &pos,
class
load_bearing_block *blk);
    virtual char *block::type();
    char *block::name();
    int block::is_movable();
//  deleted several members from exampleprotected:
    char *nm;
    int movable;
    int width;
    int height;
    class point  position;
    class load_bearing_block *supported_by;
    Panel_item panel_item;
    /* from base class table::load_bearing_block */
public:
    load_bearing_block::load_bearing_block();
   load_bearing_block::load_bearing_block(char *name, int w, int
h,
const class point &pos, class load_bearing_block *blk);
    virtual int load_bearing_block::is_load_bearing();
    virtual class list *load_bearing_block::supported_blocks();
   void load_bearing_block::add_supported_block(class block &b);
    void load_bearing_block::remove_supported_block(class block
&b);
    virtual void load_bearing_block::print_supported_blocks();
    virtual void load_bearing_block::clear_top();
 virtual void load_bearing_block::put_on(class block &object);
    class point load_bearing_block::get_space(class block
&object);
    class point load_bearing_block::find_space(class block
&object);
    class point load_bearing_block::make_space(class block
&object);
```

```
protected:
    class list *support_for;
    /* from class table */

public:
    table::table(char *name, int w, int h, const class point
&pos);
    virtual char *table::type();
    virtual 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.

This chapter describes common operations. For a fuller descriptions, see Chapter 6.

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:

5.1 Setting Breakpoints

There are three types of breakpoint action commands:

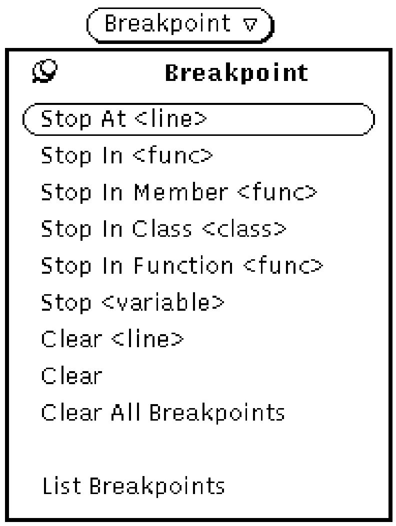
- `stop` **type breakpoints —** If the program arrives at a breakpoint created with a `stop` command, the program halts. The program cannot resume until you issue another debugging command.
- `when` **type breakpoints —** the program halts and the Debugger executes one or more debugging commands, then 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.

5.1.1 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> — sets breakpoint at selected line.

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

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

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

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

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

Clear <line> — clears breakpoint at selected line.

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

Clear All Breakpoints — clears breakpoints at all locations.

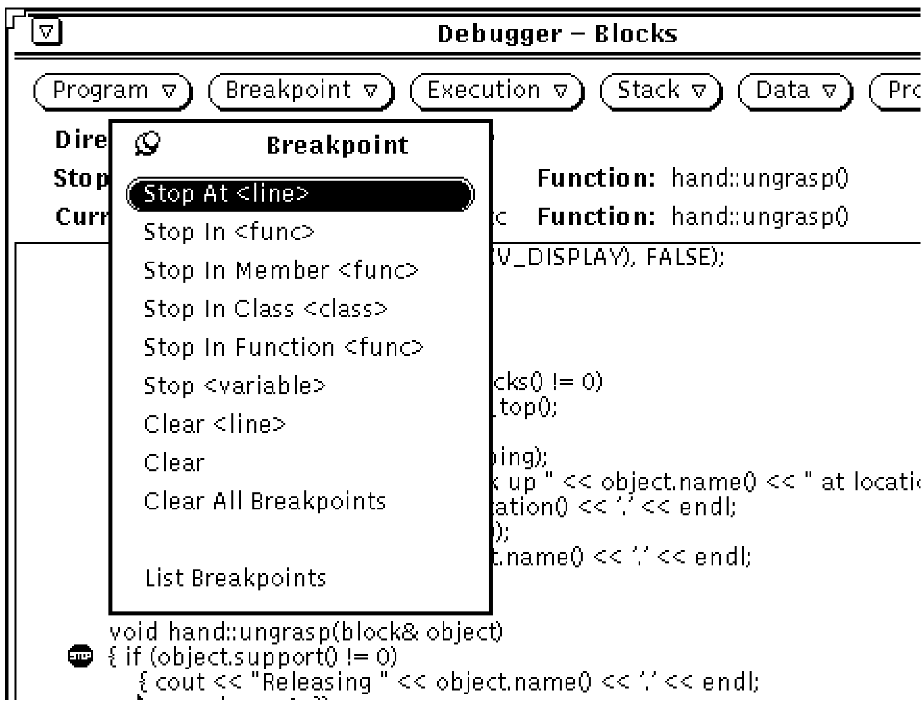
List Breakpoints — lists all currently active breakpoints and traces.

5.1.2 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,

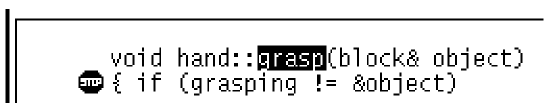
```
(debugger) stop at main.cc:3
```

To use the `stopi` command for setting a breakpoint at a machine instruction, see “Setting Breakpoints at Machine-Instruction Level” on page 15-198.

5.1.3 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.



Select either the function name or name plus arguments.

5.1.4 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. On HP-UX, `dlopen = shl_load` and `dlclose = shl_unload`.

Three exceptions:

- You cannot set a breakpoint in a `dlopen`’ed library before that library is loaded by `dlopen()`.
- Solaris 2.x — You cannot set a breakpoint in a `dlopen`’ed “filter” library until the first function in it is called.
- 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;}
```

Due to the large number of breakpoints that may be inserted by **Stop in Class**, and other Breakpoiont menu selections, you should be sure to set **dbxenv stepevents** to **on**, or select **props...**->**Debugger Events**->**Step Events** from the Debugger command pane. This will speed up `step` and `next`.

5.3 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 `cont`. In the example above, after listing the source code at the current line, the program continues executing.

5.4 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.

5.4.1 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`	Every line in the program as it is about to be executed.
`trace in` *function*	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 member_function in *class*; its line number, parameters passed in, and return value
`trace` *variable* `[in` *function*`]`	new value of *variable*, if it changes, and the line at which it changed

5.4.2 *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 click Apply.**

5.5 *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.

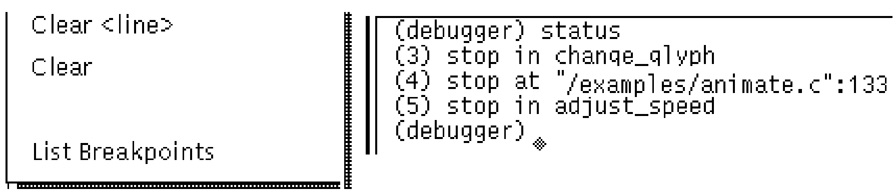
5.5.1 Listing Breakpoints and Traces

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.

5.5.2 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.**

5.5.3 Clearing a Breakpoint from the Current Focus Line

To clear a breakpoint that is set at the line where you are currently stopped:

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

This clears the last breakpoint you reach.

5.5.4 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
```

5.5.5 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 theAddress' Contents 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 *before* the instruction that wrote to the variable is executed, although the new contents of the memory are preset by the debugger by emulating the instrution. The program location arrow typically points to the line that does the modification.

- You cannot use addresses of stack variables, for example, autofunction 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 5-67.

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, which 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*

5.6 *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.

`Stop in foo if a>5` is not the same as `stop if a>5`. The former will breakpoint at `foo` and test the condition. The latter will automatically single-step and test for the condition..

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*`]`

A tracxe might seem slow because the interval between the execution of each line of code is set for too long a period of time. see Section 5.3.2 for a details about how to control the speed of a trace.

5.7 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

5.7.1 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 6-82.

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.

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:

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.

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:

6.1.1 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.

6.1.2 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.

6.1.3 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.

6.1.4 Filtering of Traces

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

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

6.1.5 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 5.7, "Event Efficiency," on page 5-74.

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), and `delete -all` delete all handlers including temporary handlers. `delete -temp` deletes only all temporary handlers.

clear

`clear`
`clear` *line*

`clear` with no arguments deletes all handlers based on breakpoints at the location where the process stopped. `clear` *line* deletes all handlers based on breakpoints on the given 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 6-90. 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 6-90.)

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 5.5.5, “Watchpoints,” on page 5-71 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

`$r3` on PPC

$r28 on PA-RISC

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)

FLTBOUNDSMemory bounds (invalid address)

FLTIOVFInteger overflow

FLTIZDIVInteger zero divide

FLTPEFloating-point exception

FLTSTACKIrrecoverable stack fault

FLTPAGERecoverable 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 Debugger has successfully attached to a process.

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. (On HP-UX `dlopen` is `shl_load` and `dlclose` is `shl_unload`.) 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 name 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* begins with a `/`, a full string match is performed. Otherwise, only the tails of the path are compared.

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

Solaris 2.x

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

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 usiSolaris 2.xng:

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

6.4.1 Limitations of `modify` event-spec

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 main 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
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.

Solaris 2.x

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. Modifiers consist of the following:

-if
`-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 condition is sometimes called a filter. A handler created with a filter is known as a filtered handler.

-in
`-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 6-96.

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, `the count` is incremented until it reaches *n*. Once that happens, the handler fires and the counter is reset to zero.

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"
22array[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 "$B1" ]
        thendelete "$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 5-61, and Chapter 6, "Event Management,"" on page 6-77, you can control the run-time behavior of a program executing under the Debugger.

This chapter is organized into the following sections:

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 Chapter 24, "dbx and the Solaris Dynamic Linker," 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. When ready, use the `run` command to start program execution.

7.1.1 Running a Program Without Arguments

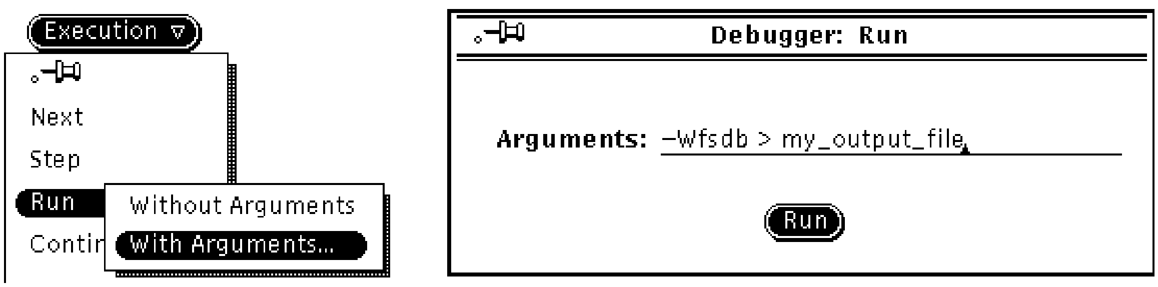
To run a program in the Debugger without arguments:

♦ **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 stop or kill 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. Refer to the example below, during the following discussion.

```
(dbx) stop in body
(3) stop in body


(dbx) call body()
stopped in body at line 33 in file "main.c"
   33        if (spin2)
dbx: Stopped within call to 'body'. Starting new command
interpreter
```

```
(dbx) cont
dbx: Call to 'body' completed. Going back to previous command
interpreter
stopped in main at line 58 in file "main.c"
   58       kill(0, 99);        /* artificial syscall fail */
```

Try again:

```
(dbx) call body()
stopped in body at line 33 in file "main.c"
   33        if (spin2)
dbx: Stopped within call to 'body'. Starting new command
interpreter
```

This time we kill the process:

```
(dbx) kill
dbx: 'kill' aborted. Recovering from nested call.
dbx: Call to 'body' terminated. Going back to previous command
interpreter
(dbx) kill
(dbx)
```

The second kill was successful.

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.

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.

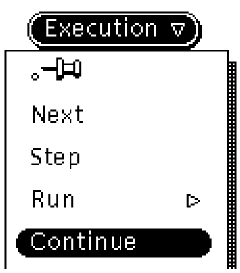
Unless the called function contains a breakpoint, when you use `call`, the Debugger behaves "next-like," returning from the called function. However, if the program hits a breakpoint in the called function, the Debugger stops the program at the breakpoint and emit the message . If you now issue a `Stack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

If you continue execution, the call will return normally. If you attempt to kill, run, rerun, or debug, the command will abort as dbx tries to recover from teh nested interpreters. You can then re-issue the command.

7.3 Continuing a Program

To resume program execution 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.**

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.

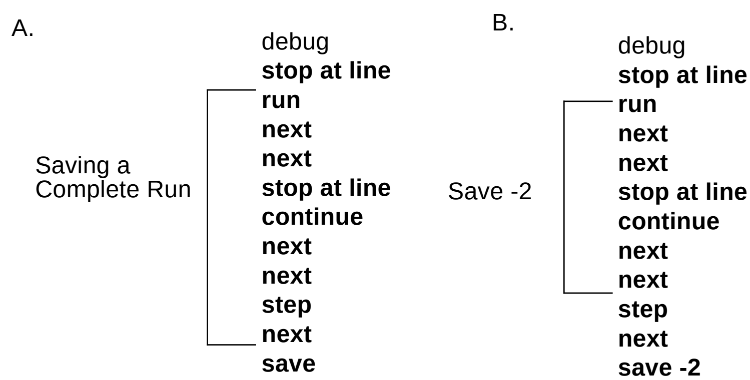
This chapter is organized into the following sections:

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.

8.1.1 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.

8.2.1 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*

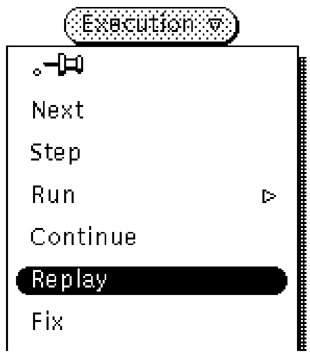
where *file_name* is the name of the file to which you saved the debugging run.

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.

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.

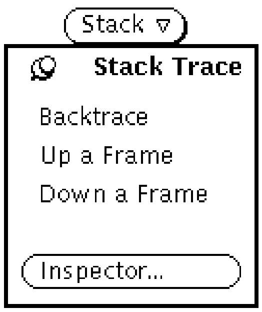
You can use the Stack Inspector in conjunction with dbx commands to manipulate the frame you are visiting. Use `dump` to show variables local to the frame. You use the `print` command or the Visual Data Inspector to examine the value of a variable, and you can use the `assign` command to change the value of a local variable.

This chapter is organized into the following sections:

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.

Type `help where` to see all options to the `where` command. Also, refer to the `hide` and `unhide` commands via `help hide` and `help unhide`.

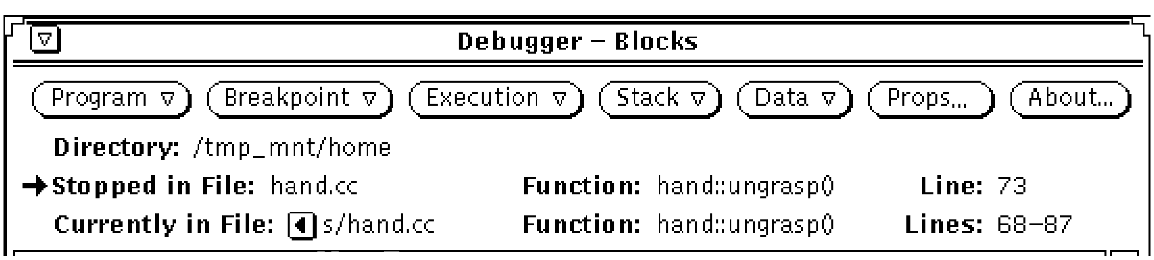
♦ **Type** `help where` **in the command pane.**

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.

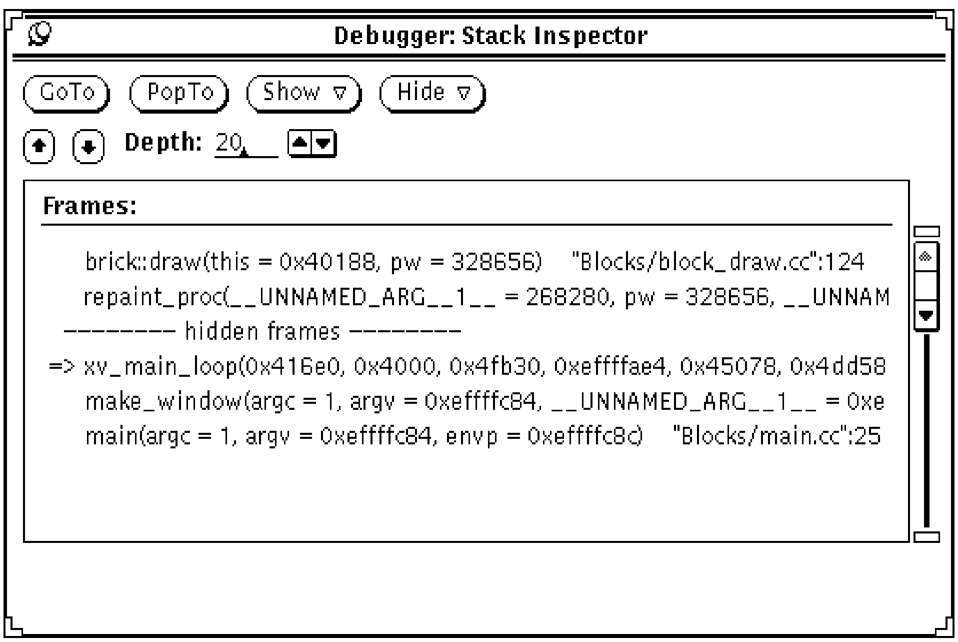
9.2 The Stack Inspector

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.

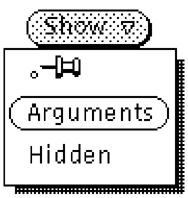
The window contains two command buttons, two menu buttons, up and down arrows that move the cursor in the Frames pane and the Depth field, which specifies the number of stack frames currently displayed in the Frames pane.

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 *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

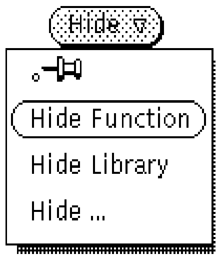
The command toggles to No Arguments.

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 Hide Menu



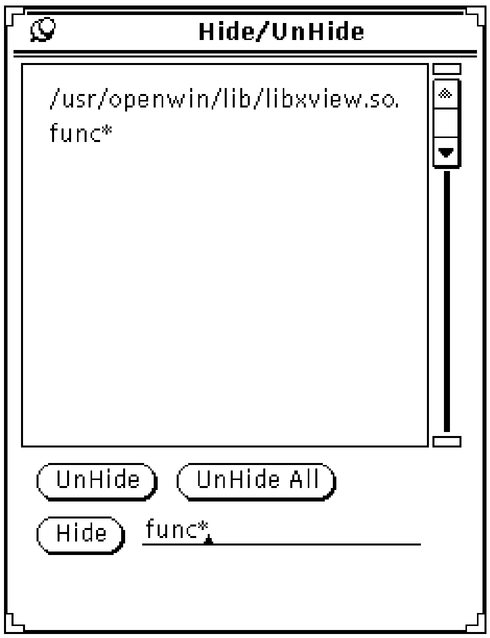
Hide Function — hides a selected function. This command hides functions by name. That is, all functions with the same name as the selected function are hidden.
Hide Library — hides all functions from the library that contains the selected function.
Hide — displays the Hide/Unhide window described in 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 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:

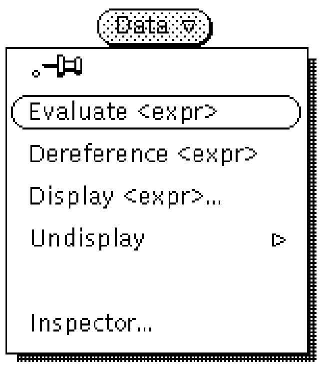
- Evaluate data (print) - spot-checks the value of an expression by using the Evaluate/Print commands
- Display data (display) - monitors the value of an expression each time the program stops by using the Display command
- Solaris 2.x — Inspect data (display plus relations) - examines program variables 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:

In addition to printing evaluations in the command pane, the Debugger features a separate Data Display window for monitoring data.

10.1 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.“

10.2 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.

10.2.1 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 4-52.)

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 4-51.

10.2.2 Variables Outside the Scope of the Current Function

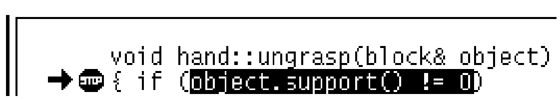
When you want to evaluate (or monitor) a variable that is outside the scope of the current function:

- Qualify the name of the function.
  See “Qualifying Symbols with Scope Resolution Operators” on page 4-48.
- Visit the Function; that is, change current function.
  See “Visiting Code” on page 4-43 for a description of how to change the current function.

10.2.3 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:

```
stopped in hand::ungrasp at line 71 in file "hand.cc"
(debugger) print object.support() != 0
object->support(object) != 0 = 1
(debugger)
```

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
```

10.2.4 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:

`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`:

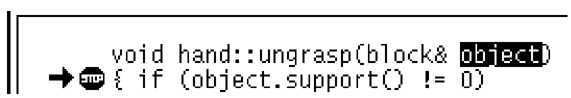
```
(debugger) print *t
*t = {
a = 4
}
(debugger)
```

10.3 Monitoring Expressions

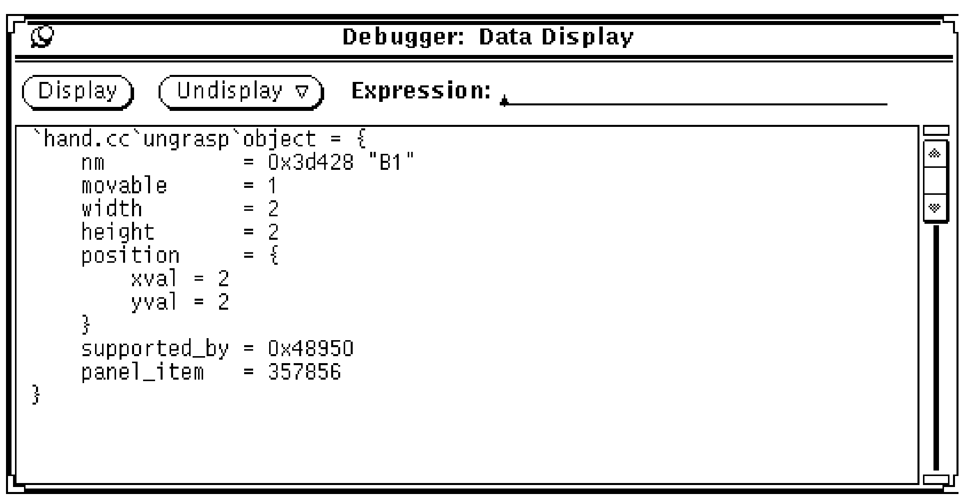
Monitoring the value of an expression each time the program stops is an effective technique for learning how and when a particular expression or variable changes. Display instructs the Debugger to monitor one or more specified expressions or variables. Monitoring continues until you turn it off with 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.)

10.3.1 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
```

10.4 Assigning a Value to an Variable

To assign a value to a variable:

♦ **Type in the command pane**

(debugger) **assign** *variable* **=** *expression*

10.5 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 an 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 an array:

```
(debugger) print arr(2,4)
```

10.5.1 Array Slicing for Arrays

(For the array slicing syntax for C and C++, refer to `helpfile.dbx` file.)

For arrays, the Debugger `print` command allows you to evaluate part of a large array. 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).

10.5.2 Syntax for Array Slicing and Striding

Array-slicing is supported in the `print`, `display`, and `inspect` commands for C, C++, and Fortran.

Array-slicing syntax for C and C++:

```
print <arr-exp>[<first-exp>..<last-exp>:<stride-exp>]
```

where:

`<arr-exp>` is an expression that should evaluate to an array or pointer type,

`<first-exp>` and `<last-exp>` are first and last elements to be printed,

`<stride-exp>` is the stride.

`<first-exp>`, `<last-exp>`, and `<stride-exp>` are (optional) expressions that should evaluate to integers. `<first-exp>` and `<stride-exp>` default to 0 and 1, respectively. `<last-exp>` is optional for array types and defaults to its upper bound.

Examples:

```
(dbx) print arr[2..4]
arr[2..4] =
[2] = 2
[3] = 3
[4] = 4
(dbx) print arr[..2]
arr[0..2] =
[0] = 0
[1] = 1
[2] = 2

(dbx) print arr[2..6:2]
arr[2..8:2] =
[2] = 2
[4] = 4
[6] = 6
```

Array-slicing syntax for Fortran:

```
print <arr-exp>(<first-exp>:<last-exp>:<stride-exp>)
```

where:

`<arr-exp>` is an expression that should evaluate to an array type,

`<first-exp>` and `<last-exp>` are first and last elements to be printed,

`<stride-exp>` is the stride.

`<first-exp>`, `<last-exp>`, and `<stride-exp>` are (optional) expressions hat should evaluate to integers. `<first-exp>` and `<last-exp>` default to lower and upper bound, respectively. `<stride-exp>` defaults to 1.

Examples:

```
(dbx) print arr(2:6)
arr(2:6) =
     (2) 2
     (3) 3
     (4) 4
     (5) 5
     (6) 6
(dbx) print arr(2:6:2)
arr(2:6:2) =
     (2) 2
     (4) 4
     (6) 6
```

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`)

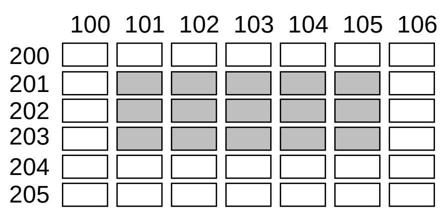
For an *n*-dimensional slice, separate the definition of each slice with a comma:

```
(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 omit 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.

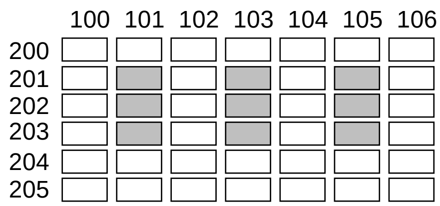
Strides

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.

Visual Data Inspector 11

Note – 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.

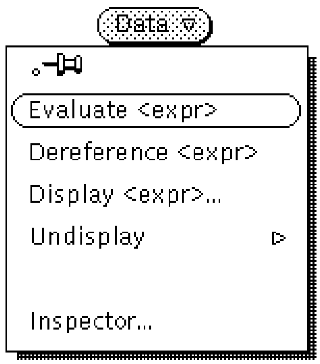
For information on evaluating or displaying the value of expressions, variables, and other data structures, see Chapter 10, "Evaluating and Displaying Data."

This chapter is organized into the following sections:

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.)

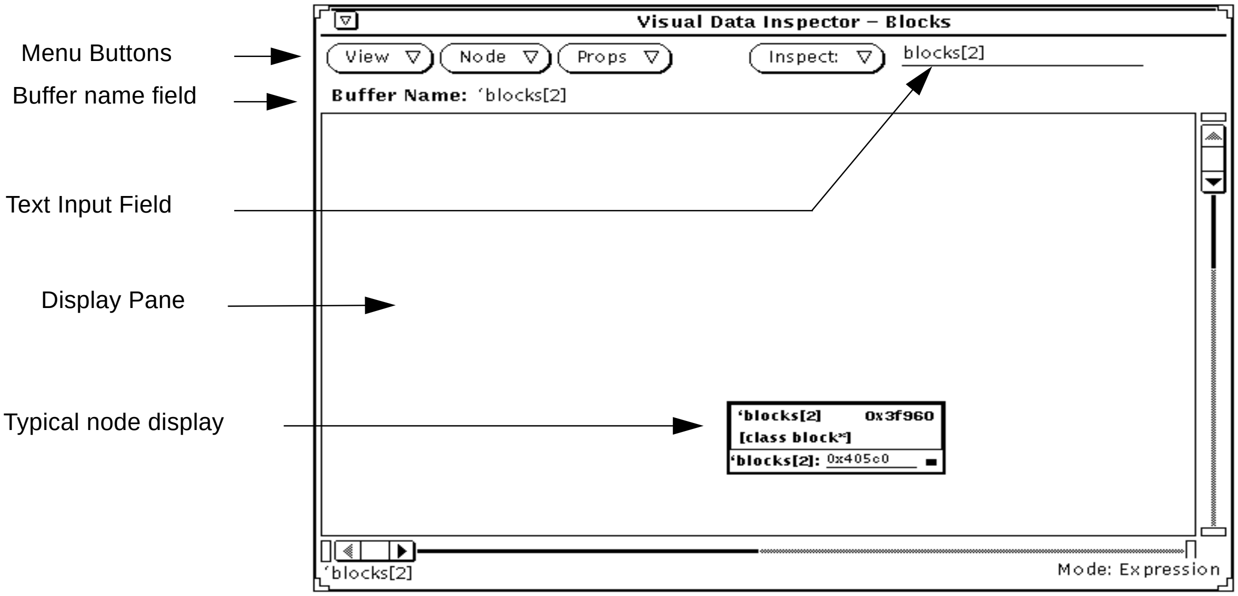
Display — displays the Data Display window, where the Debugger monitors the value of the selected expression.

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.

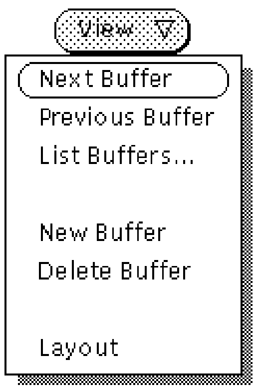


The window header shows the name of the executable program being inspected. The left footer shows the qualified symbol representing the node using scope resolution operators. The node name may have been truncated in the display node header (as indicated by a “<“ on the left of the name field). The right 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.

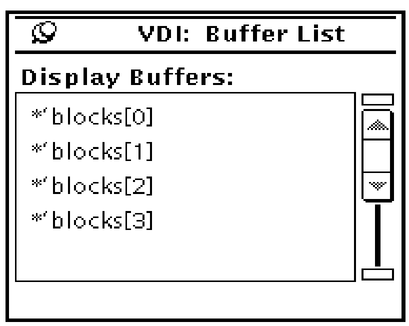
11.2.1 The View Menu

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.

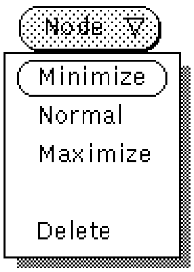


The VDI Buffer List window contains a list of open buffers.

♦ **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.

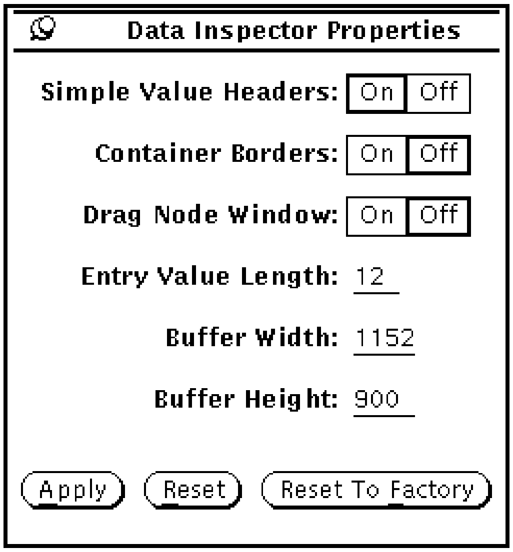
11.2.3 The VDI Props Menu

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 — selects On to display header information for each node. Select Off to display only a single value and name.

Container Borders — selects On to display borders around structures that are nested. Select Off to remove borders around the containers.

Drag Node Window — selects On to drag nodes as solid objects. Select Off to drag node outlines.

Entry Value Length — specifies 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 — allows 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.

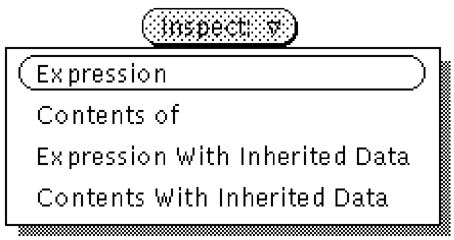
Apply — applies the changes, if any.

Reset — restores 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 — inspects the expression exactly as requested.

Contents of — inspects 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 — inspecs 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.

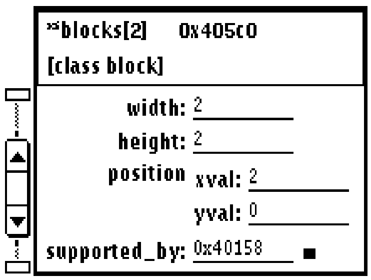
♦ **You then click on the Inspect button or use carriage return to start the inspection. A stopwatch is displayed while inspection is in progress.**

♦ **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.

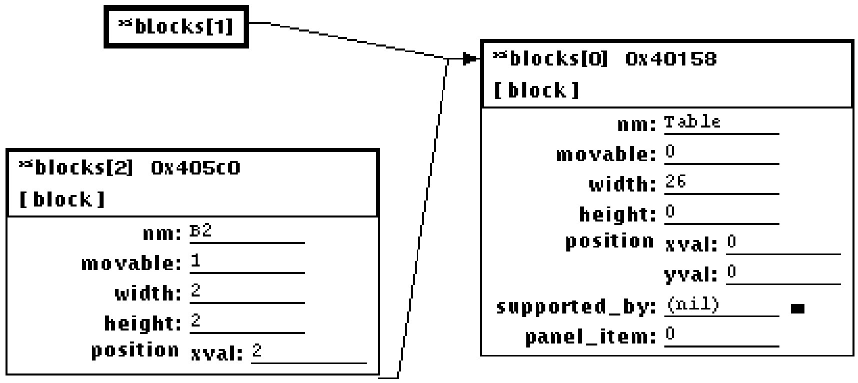


Figure 11-1 Display Pane Node Formats

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.

Process/Thread Inspector 12

Note – 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:

12.1 Introduction to the Multithreaded Debugger

The Debugger can debug mulithreaded applications that use either Solaris threads or POSIX threads

The Debugger recognizes a multithreaded program by detecting whether it utilizes `libthreads.so.` The program will use `libthread.so` either by explicitly being compied with `-lthread`, or implicitly by being compiled with `-lpthread.`

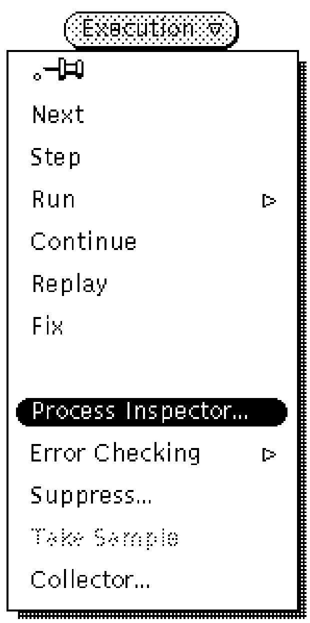
When it detects a multithreaded program, the Debugger tries to `dlopen libthread_db.so`, a special library for thread debugging. The standard path for this library is `/usr/lib`; if it is located elsewhere, use the environment variable `THREAD_DB_DIR` to specify the correct path to `libthread_db.so`.

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 “*Multithreaded Programming Guide*” 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 `lwpid_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.

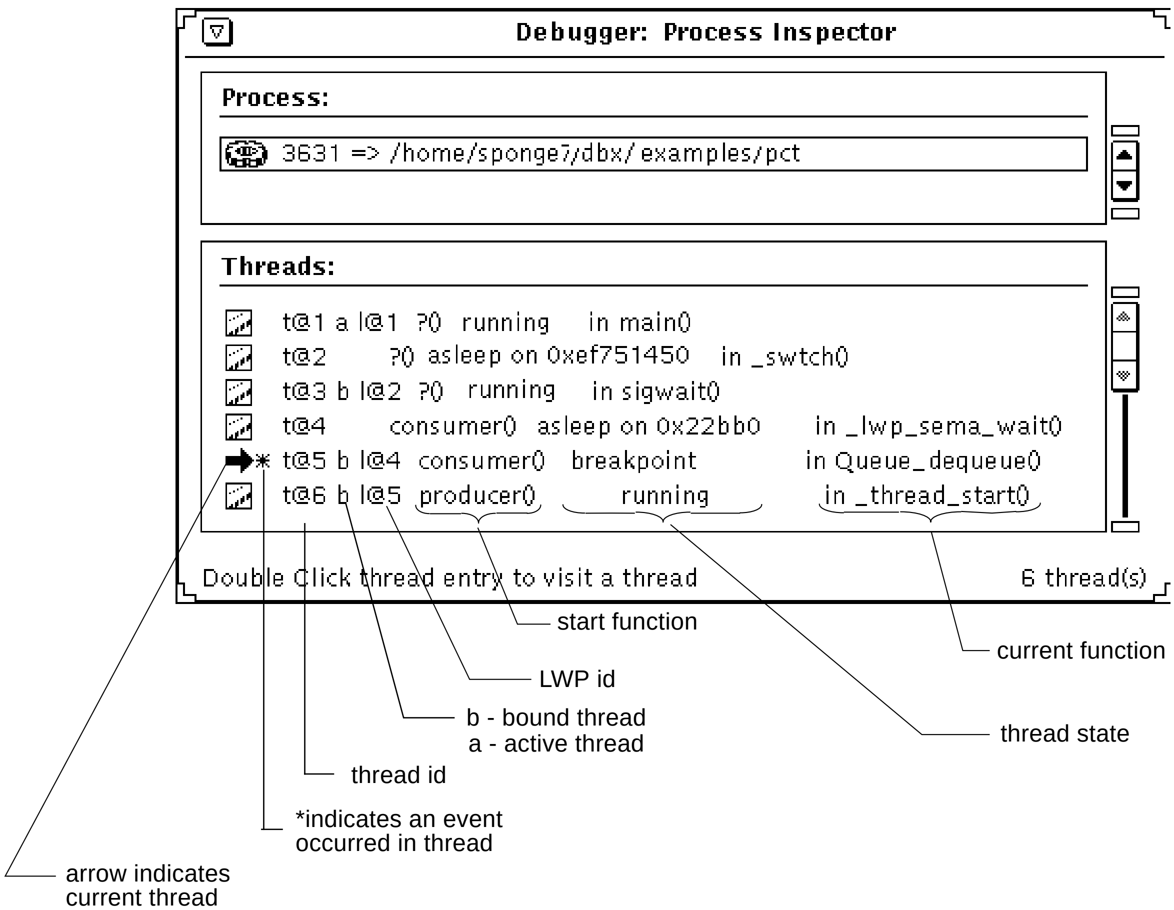


Figure 12-1 The Process/Thread Inspector Window

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	When a detached thread exits (`thr_exit()`), it is in a zombie state until it has rendezvoused through the use of `thr_join()`. `THR_DETACHED` is a flag specified at thread creation time (`thr_create()`). A non-detached thread that exits is in a zombie state until it has been reaped.
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`.

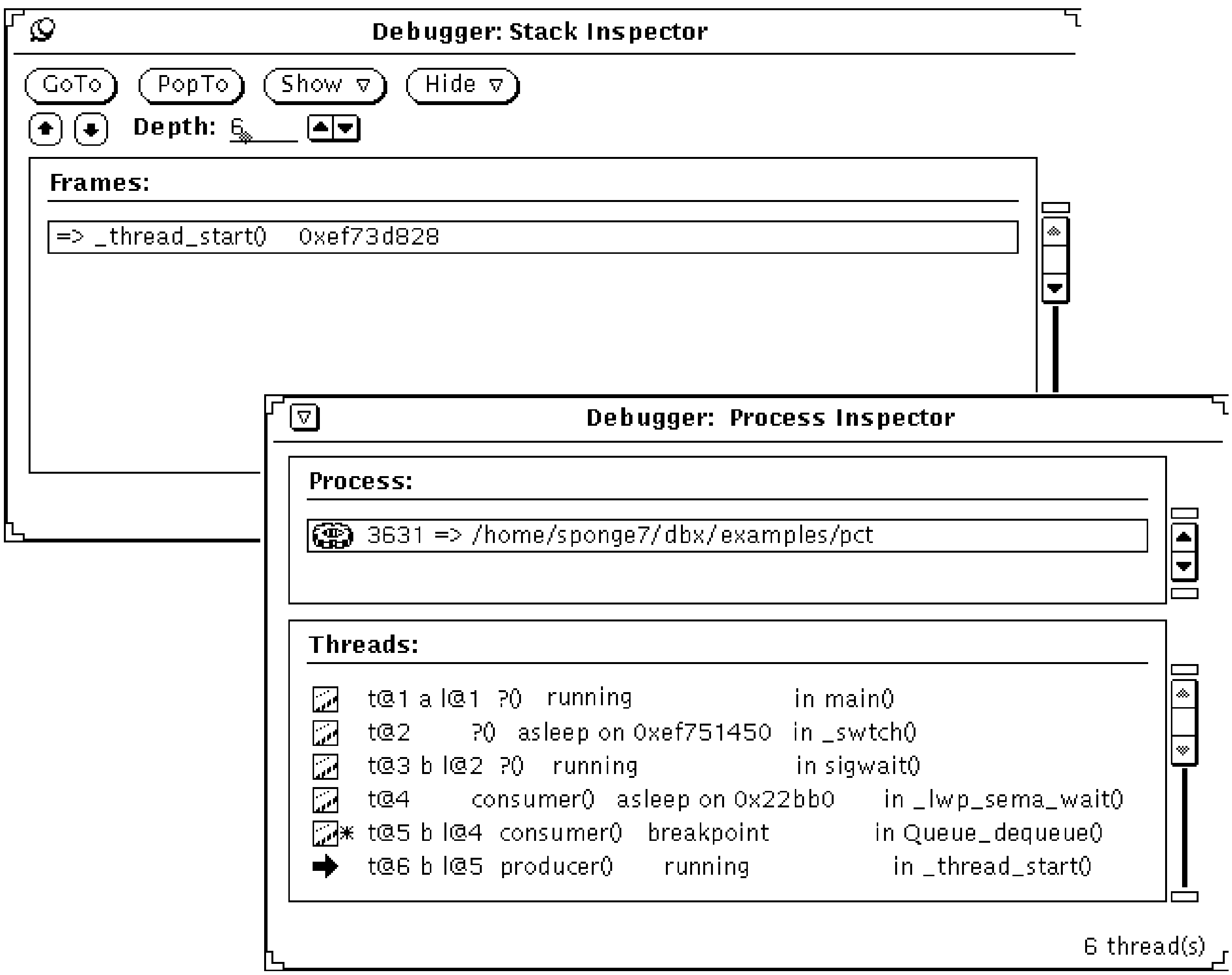
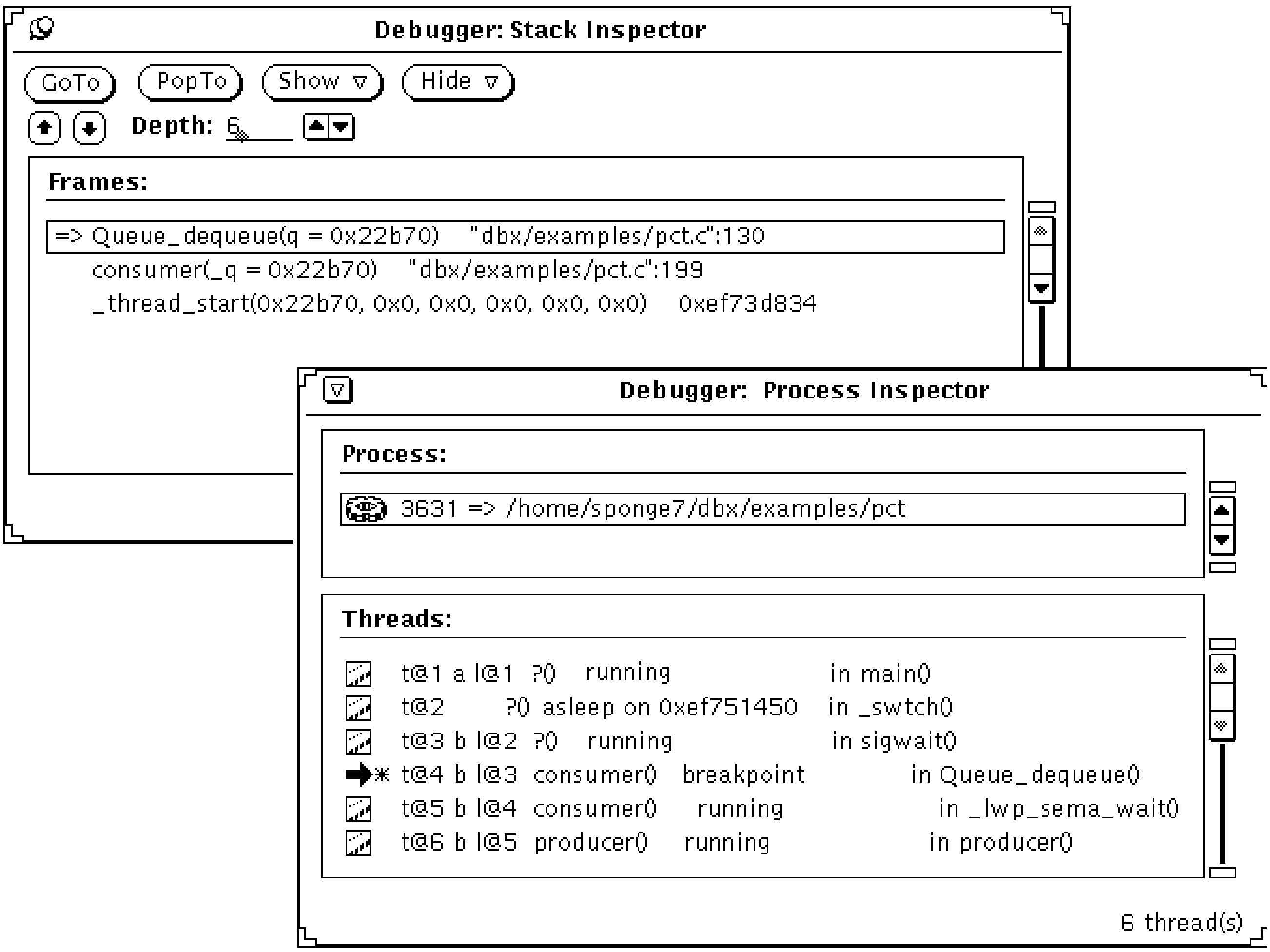


Figure 12-2 Viewing the Context of Another Thread



12.4.1 *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.

12.4.2 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 #.

Table 12-2 LWP State

LWP State	Description
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.

Customizing the Debugger 13

This chapter describes how to use the controls on the Properties window and how to use the initialization file, `.dbxrc`, 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:

13.1 The Props Button

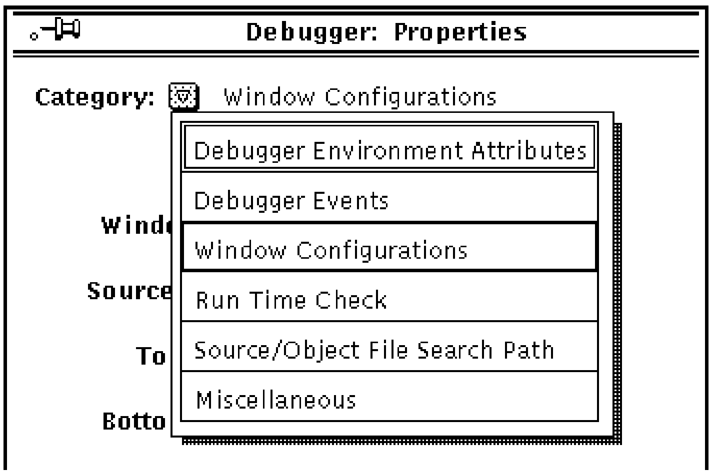
Clicking on the Props button displays the Properties window. The Properties window controls are arranged on six separate control panels:

- **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, Step Events, 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)**
  (SPARC and Solaris only) 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, and change the Log Filename.

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.**



13.2 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.

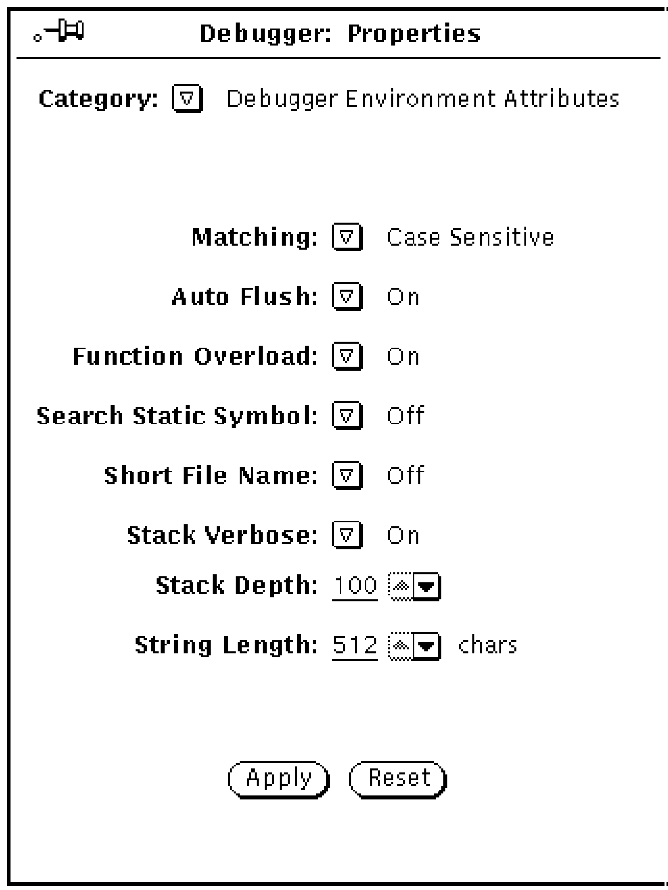


Figure 13-1 Debugger Environment Attributes Control Panel

Matching:

Set whether the Debugger distinguishes between upper- and lower-case characters when interpreting variables and function names.

`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 input_case_sensitive [true|false}`

Note – The Debugger detects when you load a FORTRAN module; it switches the setting automatically to `false` (with the exception that `MAIN()` must always be uppercase.) However, if during a session (or by setting `dbxenv input_case_sensitive true` 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 `true`.

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 output_auto_flush {on | off}`

Function Overload:

Set to allow the Debugger to do automatic overload resolution of C++ functions.

Default: On
Debugger command: `dbxenv overload_function {on | off}`

Search Static Symbol:

Set to find file static symbols, even when they are not in the scope.

Default: Off
Debugger command: `dbxenv scope_look_aside {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: On
Debugger command: `dbxenv output_short_file_name {on | off}`

Stack Verbose:

Set to allow printing of arguments and line information in `where`.

Default: On
Debugger command: `dbxenv stack_verbose {on | off}`

Stack Depth:

Set the default size for the `where` command.

Default: 100
Debugger command: `dbxenv stack_max_size` *number*

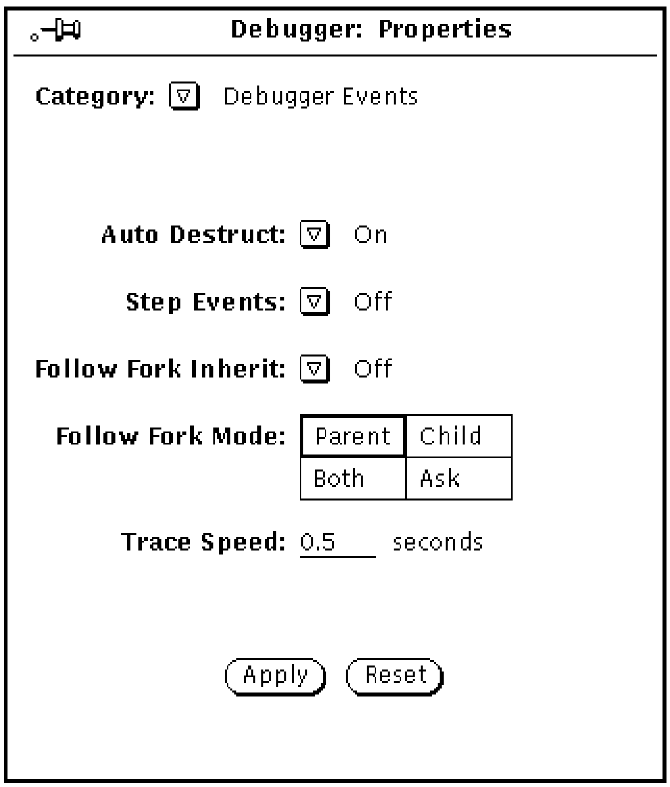
String Length:

Set maximum number of characters printed for character pointers, 'char *'.

Default: 512 characters.
Debugger command: `dbxenv output_max_string_length` *number*

13.3 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_auto_destruct {on | off}`

Step Events

Set `on` to allow breakpoints while using the `step` and `next` commands.

Default: Off
Debugger command: `dbxenv step_events {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 | both | 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 trace_speed number_seconds`

13.4 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.

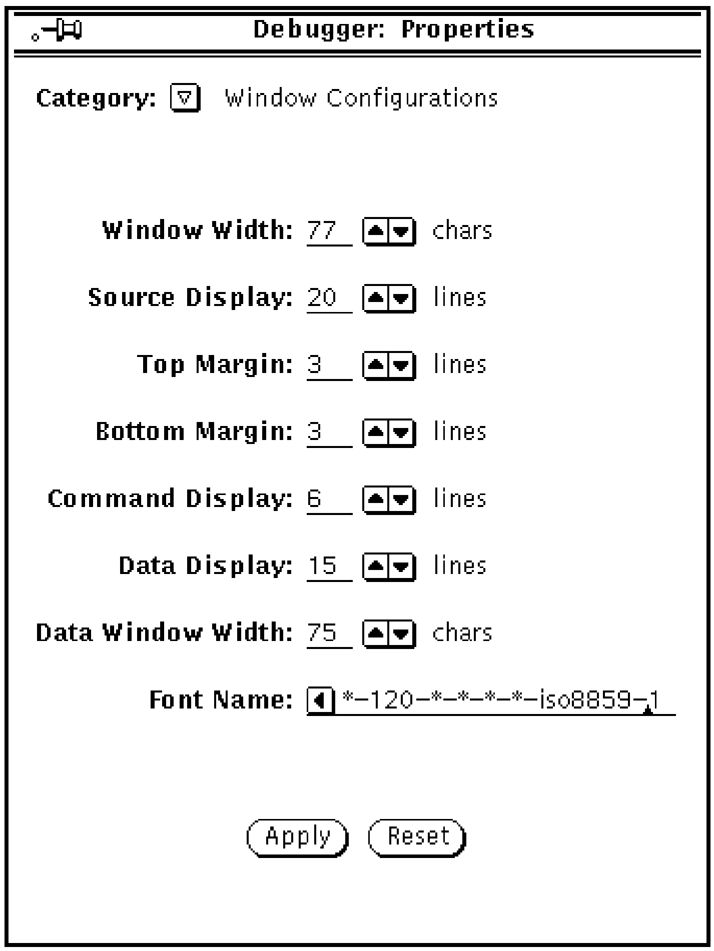


Figure 13-2 Window Configuration Control Panel

Window Width:

Set the width, in characters, of the source display or command pane.

Default: 77 characters
Debugger command: `toolenv width` *num*

Source Display:

Set the number of lines of code displayed in the source display.

Default: 20 lines
Debugger command: `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
Debugger command: `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
Debugger command: `toolenv botmargin` *num*

Command Display:

Set the number of lines displayed in the command pane.

Default: 6 lines
Debugger command: `toolenv cmdlines` *num*

Data Display:

Set the number of lines displayed in the Data Display Window.

Default: 15 lines
Debugger command: `toolenv displines` *num*

Data Window Width:

Set the width, in characters, of the Data Display Window.

Default: 75
Debugger command: `toolenv dispwidth` *num*

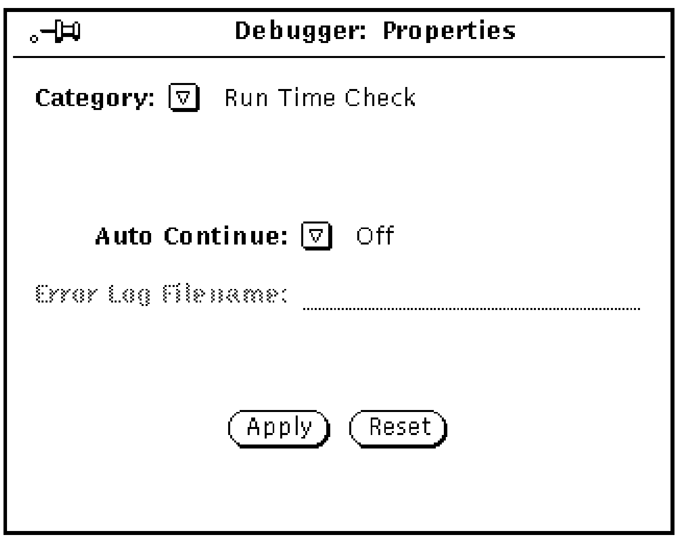
Font Name:

Set the name of the font file the Debugger uses to display code in the Debugger windows and panes.

Default: *system_default_file*
Debugger command: `toolenv font` *file*

13.5 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
Debugger command: `dbxenv autocontinue {on | off}`
Debugger command: `dbxenv errlogfile` *filename*

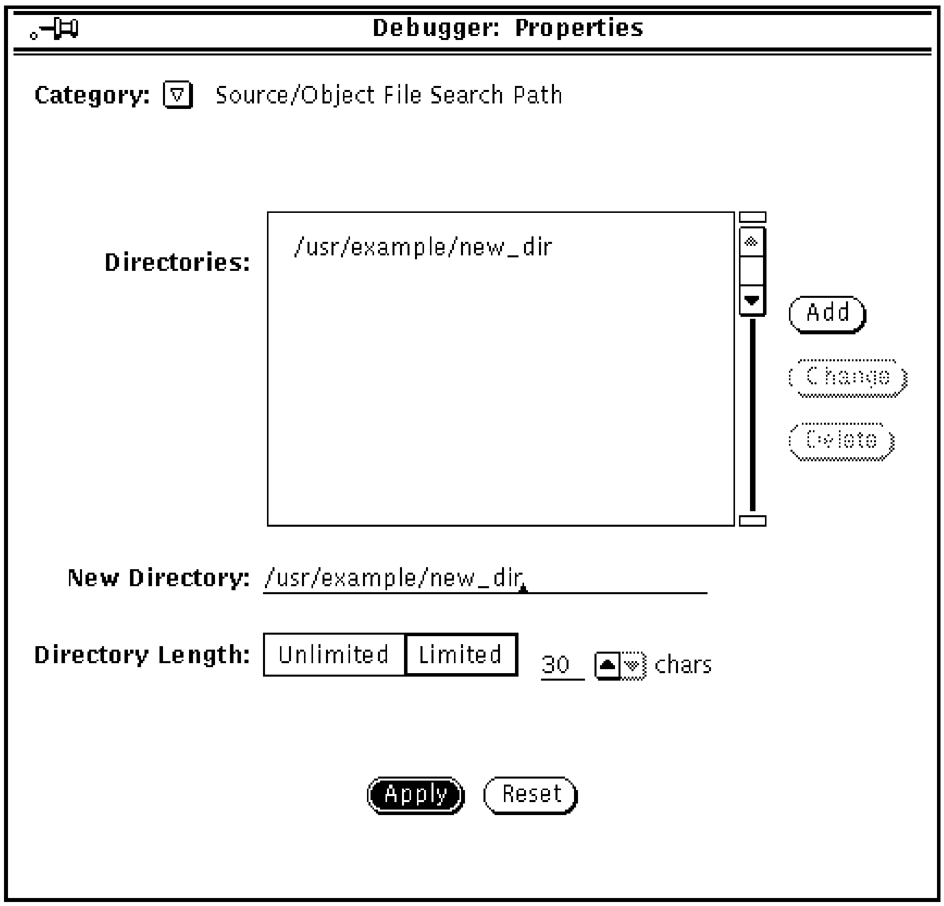
13.6 Setting the Source/Object File Search Path

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.**
2. **Click Add.**
   The directory name appears in the Directories display box.
3. **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.**
4. **Click the Change button.**
5. **Click Apply.**

To delete a path from the search path list:

1. **Select a pathname in the Directories box.**
2. **Click Delete.**
3. **Click Apply.**

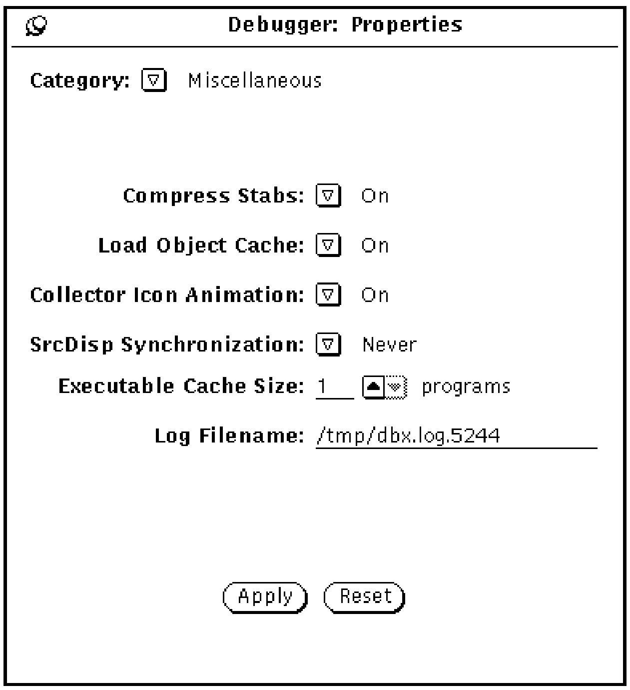
13.6.1 Adjusting the Display of Pathnames in the Directories Box

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.

13.7 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 symbol_info_compression {on | off}`

Load Object Cache

Enable or disable the load object cache.

Default: On
Debugger command: `dbxenv locache_enable {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. You should set it 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`*>

13.8 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.

13.8.1 A Sample Initialization File

Here is a sample `.dbxrc` file:

```
dbxenv case input_case_sensitive false
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.
input_case_sensitive refers to the matching control.
true 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.

13.8.2 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>
```

13.9 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.

13.9.1 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.

Editing a Program 14

This chapter describes how to edit source code in a separate command tool or from within the Debugger. To edit the currently displayed file in the source display, choose Enable Edit from the source display floating pop-up menu.You can edit code in the source display after converting the initially read-only pane to an editable pane.

This chapter is organized into the following sections:

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.

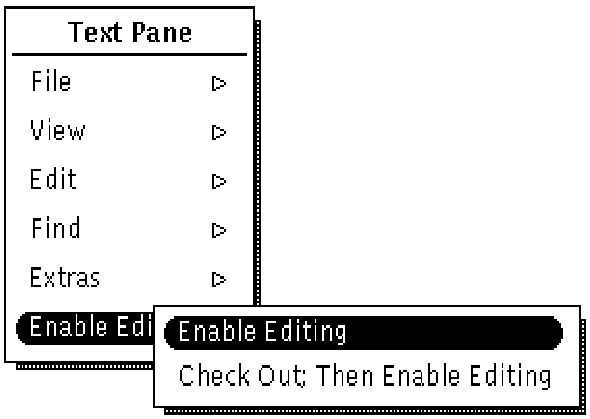
Invoking dbx with the `-editor` flag causes dbx to send and receive ToolTalk messages. If there is an XEmacs editor on the same $DISPLAY as dbx, and that XEmacs has had Eos enabled (via the SPARCworks=>Select for Debugger menu item), then dbx will use XEmacs for source display and will respond to debugger commands issued from XEmacs. See the XEmacs help topic `SPARCworks` for more information.

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.**



After enabling editing, the source display window behaves just like a normal textedit window.

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

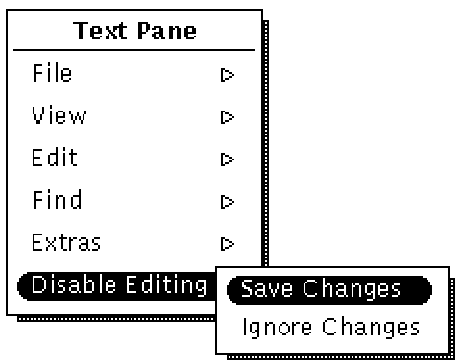
Notice that the Enable Edit item on the floating pop-up menu is a pull-right menu. The second item, Check Out; Then Enable Editing, automatically checks the file out of SCCS, if it is under this version control software. You cannot check it back into SCCS from within the Debugger.

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.**

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:

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.

(PowerPC) Table 15-4 lists the PowerPC registers.

(PA-RISC) DebuggerTable 15-5 lists the PA-RISC register.

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 15-190.)

The following examples show 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 Dx117c8
(debugger) x0x17c8/5i
support_5blockFv+8:            save  %sp, %g1, %sp
support_5blockFv+0xc:          st    %io, [%fp68]
support_5blockFv+10:           ld    [%fp, + 68%], %i3
support_5blockFv+14:           ld    [%l3,  + 24], %o0
support_5blockFv+18:           ba    support_5blcokFv+0x
                                     20
```

(x86) This is an x86 example:

```
(dbx) x &support/5i
support:            push1       %ebp
support+1:          mov1        %esp, %ebp
support+3:          sub1         $4,%esp
support+6:          mov1        8(%ebp),%ecx
support+0x9:        mov1        24(%ecx),%ecx
```

(PowerPC) This is a PowerPC example:

```
(dbx) x &`Block`hand.cc`block::support/5i
support:                                    mflr  %r0
support+0x4:                                stw   r0,0x0004(%r1)
support+0x8:                                mr    %r11,%r1
support+0xc:                                stwu  %r1,0xffd0(%r1)
support+0x10:                               stw   %r31,0x002c(%r1)
```

(PA-RISC) This is an PA-RISC example:

```
(hp_dbx) x &`Block`hand.cc`block::support/5i
support:                                       STW   %rp,-20(%sr0,%sp)
support+0x4:                                   OR    %sp,%r0,%t3
support+0x8:                                   STWM  %r3,4(%sr0,%sp)
support+0xc:                                   STWM  %r4,4(%sr0,%sp)
support+0x10:                                  LD0   248(%sp),%sp
```

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

i	Display as an assembly instruction
d	Display as a halfword in decimal
D	Display as a word in decimal
o	Display as a half-word 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 as a wide character
w	Display as a wide character.
s	Display as a string of characters terminated by a null byte.
W	Display as a wide character.
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], %l0
main+0x54:     add   %l0, 4, %l0
main+0x58:     ld    [%l0], %l0
main+0x5c:     mov   %l0, %o0
main+0x60:     call  0x24e18    [unresolved PLT 12: atoi]
main+0x64:     nop
main+0x68:     mov   %o0, %l0
main+0x6c:     st    %l0, [%fp + -8]
   99                 foo(i);
main+0x70:     ld    [%fp + -8], %l0
main+0x74:     mov   %l0, %o0
main+0x78:     call  foo
```

```
main+0x7c:      nop
   100                exit(0);
main+0x80:      mov    0, %l0
main+0x84:      mov    %l0, %o0
main+0x88:      call   0x24218 [unresolved PLT 16: exit]]
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`.

(PowerPC) Here is a PowerPC example:

```
(dbx) listi 98, 100
  98                              int i = atoi(argv[1]);
main+0x50:          lwz           %r12,0xffe8(%r29)
main+0x54:          lwz           %r3,0x0004(%r12)
main+0x58:          crclr         cr1[eq]
main+0x5c:          bl            0x02010814  [atoi  [PLT]]
main+0x60:          stw           %r3,0xffe0(%r29)
```

```
  99                                   foo(i);
main+0x64:          lwz                %r3,0xffe0(%r29)
main+0x68:          bl                 0x020005f0       [foo]
  100                                    exit(0);
main+0x6c:          li                 %r3,0x0000
main+0x70:          crclr              cr1[eq]
main+0x74:          bl                 0x02010804   [exit [PLT]]
main+0x78:          mr                 %r11,%r29
main+0x7c:          lwz                %r29,0x0024(%r1)
main+0x80:          lwz                %r30,0x0028(%r1)
main+0x84:          lwz                %r31,0x002c(%r1)
main+0x88:          lwz                %r0,0x0034(%r1)
main+0x8c           mtlr               %r0
main+0x90           mr                 %r1, %r11
main+0x94:          blr-
(dbx)
```

(PA-RISC) Here is a PA-RISC example:

```
(hp_dbx) listi 8, 10
  8                                    int i = atoi(argv[1]);
main:               STW                %rp,-20(%sr0,%sp)
main+0x4:           OR                 %sp,%r0,%t3
main+0x8:           STWM               %r3,4(Z%sr0,%sp)
main+0xc:           STWM               %r4,4 (%sr0,%sp)
main+0x10:          LD0                248 (%sp),%sp
main+0x14:          OR                 %t3,%r0,%r3
main+0x18:          STW                %t3,-4(%sr0,%sp)
main+0x1c:          STW                %arg0,-36(%sr0,%r3)
```

```
main+0x20:     STW      %arg1,-40(%sr0,%r3)
main+0x24:     BL n     244, %rp
main+0x28      OR       %r0,%r0,%r0
main+0x2c:     LDW      -40(%sr0,%r3),%r4
main+0x30:     ADDI     4,%r4,%r4
main+0x34:     LDW      0(%sr0,%r4)%r4
main+0x38:     OR       %r4,%r0,%arg0
main+0x3c      BL n     32636,%rp
main+0x40      OR       %r0,%r0,%r0
main+0x44:     OR       %ret0,%r0,%r4
main+0x48:     STW      %r4,148,(%sr0,%r3)
     9         foo      (i);
main+0x4c:     LDW      148(%sr0,%r3)%r4
main+0x50:     OR       %r4,%r0,%arg0
main+0x54:     BL n     76,%rp
main+0x58:     OR       %r0,%r0,%r0
    10         exit     (0);
main+0x5c:     LDO      0(%r0),%r4
main+0x60:     OR       %r4,%r0,%arg0
main+0x64:     BL n     3260,%rp
main+0x68:     OR       %r0,%r0,%r0
```

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 The `adb` Command

The `adb` command allows you to enter commands in an `adb(1)` syntax. You may also enter adb mode which interprets every command as adb syntax. Most adb commands are supported. Refer to the online help topic "adb" for details.

15.5 Sun-4 Register Information

The following register information is for SPARC systems.

Table 15-2 Table 15-2Sun-4 Registers

`$g0-$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

The $f0f1 $f2f3 ... $f30f31 pairs of floating point registers are treated as having C "double" type (normally $fN registers are treated as C "float" type). See the *SPARC Architecture Reference Manual* and the *Sun-4 Assembly Language Reference Manual* for more information on Sun-4 registers and addressing.

15.6 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

15.7 PowerPC Register Information

The following register information is for PowerPC systems.

Table 15-4 PowerPC Register Names

Register	Description
`$r0-$r31`	General Purpose Registers
`$crr`	Condition Register
`$msr`	Machine State Register
`$ctr`	Count Register
`$xer`	Integer Exception Register
`$mq`	Multiply Quotient / Divide Dividend Register
`$lr`	Link Register
`$f0-$f31`	Floating Point (Double Precision) Registers
`$fpscr`	FP Status and Control Register
`$sp`	Equivalent to $r1
`$ps`	Equivalent to $msr

15.8 PA-RISC Register Information

The following information is for PA-RISC (HP) systems.

Table 15-5 PA-RISC Register Names

Register	Description
`$r0-$r31`	General Purpose Registers
`$rp`	Return Pointer
`$fp`	Frame Pointer
`$sp`	Stack Pointer
`$sr0-$sr7`	Space Registers
`$cr0-$cr31`	Control Registers (0 through 7 are reserved)

Register	Description
`$f0-$f31`	Floating Point Registers (Double Precision)
`$fr0l-$fr31l`	Floating Point Registers (single precision Left)
`$fr0R-$fr31R`	Floating Point Registers (single precision Right)
`$pc`	Program Counter

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:

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>
```

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)` or one of its variations) 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 and HP/UX, 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

Solaris 2.x

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:

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 displayed. However, the *sig* is still forwarded to the process.

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.

To see the list, type `catch` or `ignore` in dbx or the debugger.

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 *signal_name*.

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 7-110). 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

Note – This feature is available only on Solaris 2.x.

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 manual,** *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:

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.

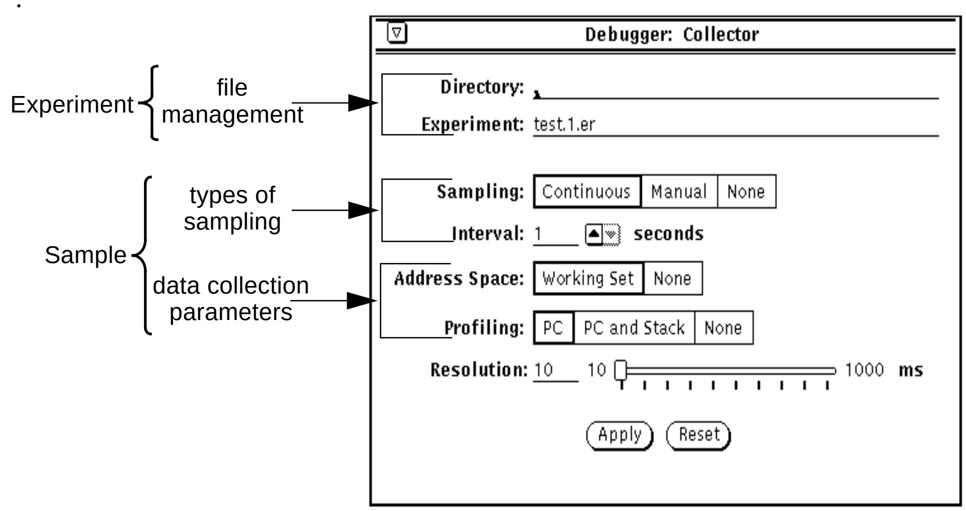


Figure 18-1 Collector Window

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.

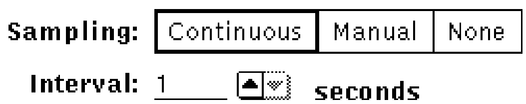
  This area controls the samples that are taken. Each sample contains information on the target application over a specified period of time

- 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.

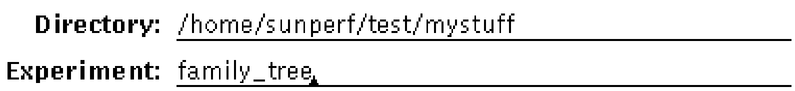


Figure 18-2 Experiment Area of the Collector Window

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`.

Runtime Checking 19

Note – This feature is available only on Solaris and Access Checking is only available on SPARC.

The Debugger's Runtime Checking feature (RTC) enables you to automatically detect runtime errors that occur in an application during the development phase. Runtime errors such as memory access errors and memory leak errors can be detected. Additionally, memory usage can be monitored. This chapter is organized into the following sections:

19.1 Introduction

The following subsections present a general overview of Runtime Checking major features.

19.1.1 Memory Access Error Detection (SPARC only)

Programs may incorrectly read or write memory in a variety of ways. These are called memory access errors. For example, the program may reference a block of memory which has been de-allocated perhaps through a `free()` call for a heap block, or because a function returned a pointer to a local variable. Using a variable before initializing it is also a kind of access error. Access errors may result in wild pointers in the program and can cause incorrect program behavior, including wrong outputs and segmentation violations. Some kinds of memory access errors can be very hard to track down.

RTC checks whether your program accesses memory correctly by monitoring each read, write, and memory free operation.

RTC inserts checkpoints in front of each memory access 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 memory access instruction only after you continue the program.

RTC keeps track of 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 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.

19.1.2 Memory Leak Detection

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. Memory leaks result in virtual memory consumption and cause performance degradation. 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.

19.1.3 Memory Usage Profiler

The memory profiler provides information about where the memory is allocated in the program. It is useful, for example, during performance tuning or to control virtual memory use. When the program exits, a memory use report can be generated. Memory usage information can also be obtained at any time during program execution by a command that causes memory usage to be displayed.

19.1.4 When to Use RTC

A way to avoid seeing a large number of errors at once is to use RTC early 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.

19.1.5 Features

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. There are some special considerations with programs not compiled with the -g option. See Default Suppressions on page 244, for more information.

No recompiling, relinking, or Makefile changes are required to take advantage of RTC.

RTC works with all languages supported by the Debugger.

RTC works on code for which you do not have the source (for example, libraries)

RTC works with multithreaded code.

19.1.6 Requirements

Runtime Checking:

- 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.

19.1.7 Limitations

Runtime Checking:

- Does not support attaching to a running process
- Does limited checking on child processes of the debugged program
- Memory access checking may be affected by 8Mbyte limit while running RTC on large programs. See What to do if you run into this 8Mb limit: on page 263 for more information because the 8Mbyte limit is flexible in some situations. Also refer to, rtc8m on page 262.

19.2 Getting Started

This section describes how to use RTC from dbx, the Debugger, and in batch mode.

19.2.1 Operation from dbx

To use Runtime Checking with a command-line interface:

Start dbx with the `-C` option:

```
dbx -C yourprog
```

The -C flag is a convenience option to cause early loading of the RTC library.

Note – The -C option. This starting flag is not required for RTC. If you start dbx or the Debugger without the -C option and issue a `check` *type-of-checking* command, the RTC library will be loaded when you issue the next run command; that may cause reloading of the shared libraries needed by the program. Using the -C flag initially allows you to avoid reloading.

You must turn on the type of checking you want before you run the program.

To turn on memory access checking only, type

```
(dbx) check -access
```

To turn on memory leaks checking only, type

```
(dbx) check -leaks
```

To turn on memory use checking (which also includes leaks checking), type

```
(dbx) check -memuse
```

To turn on both memory leak checking *and* memory access checking, type

```
(dbx) check -all
```

To turn off memory leak checking and memory access checking, type

```
(dbx) uncheck -all
```

Run the program being tested, with or without breakpoints. Here is a simple example showing Runtime Checking being applied to a program called `hello`.

```
yourmach410: cat -n hello.c
     1 #include stdio.h
     2 #include stdlib.h
     3 #include string.h
     4
     5 char *hello1, *hello2;
     6
     7 void
     8 memory_use()
     9 {
    10    hello1 = (char *)malloc(32);
    11    strcpy(hello1, "hello world");
    12    hello2 = (char *)malloc(strlen(hello1)+1);
    13    strcpy(hello2, hello1);
    14 }
    15
    16 void
    17 memory_leak()
    18 {
    19    char *local;
    20    local = (char *)malloc(32);
    21    strcpy(local, "hello world");
    22 }
    23
    24 void
    25 access_error()
    26 {
    27    int i,j;
    28
    29    i = j;
    30 }
    31
    32 int
    33 main()
    34 {
    35    memory_use();
    36    access_error();
    37    memory_leak();
    38    printf("%s\n", hello2);
    39    return 0;
    40 }
```

```
yourmach411:
yourmach411: cc -g -o hello hello.c
yourmach412:
yourmach412: dbx -C hello
Reading symbolic information for hello
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for librtc.so
Reading symbolic information for libc.so.1
Reading symbolic information for libintl.so.1
Reading symbolic information for libdl.so.1
Reading symbolic information for libw.so.1
dbx check -access
access checking - ON
dbx check -memuse
memuse checking - ON
dbx run
Running: hello
(process id 24584)
Enabling Error Checking... done
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff6e8
    which is 96 bytes above the current stack pointer
Variable is 'j'
Current function is access_error
   29        i = j;
dbx cont
hello world
Checking for memory leaks...
Memory Leak (mel):
Found leaked block of size 32 bytes at address 0x23748
At time of allocation, the call stack was:
    [1] memory_leak() at line 20 in "hello.c"
    [2] main() at line 37 in "hello.c"

Leak Summary:
    actual leaks:          1  total size:       32 bytes
    possible leaks:        0  total size:        0 bytes

Checking for memory use...
```

```
Blocks in use (biu) report:

  Total   % of   Num of    Avg     Allocation trace
  Size     All   Blocks    Block
                           Size
==============================================================
    32     72.72%   1        32     memory_use < main
    12     27.27%   1        12     memory_use < main

Blocks in use Summary:
             blocks in use:     2  total size:      44 bytes


execution completed, exit code is 0
dbx quit
```

The function access_error() reads variable j before it is initialized. RTC reports this access error as a Read from uninitialized (rui).

Function memory_leak() does not free() the variable *local* before it returns. When memory_leak() returns this variable goes out of scope and the block allocated at line 20 becomes a leak.

The program uses global variables *hello1* and *hello2* which are in scope all the time. They both point to dynamically allocated memory which is reported as Blocks in use (biu).

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.

You can selectively suppress reporting of RTC errors using the suppress command. See Error Suppression on page 243 for details. The program continues to the next error or breakpoint, whichever is detected first.

19.2.2 Operation from the Debugger

To use Runtime Checking with a graphical user interface:

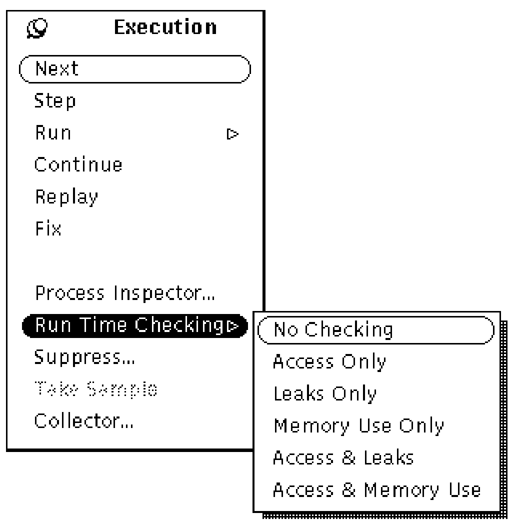
Start the Debugger with the -C option:

```
debugger -C yourprog &
```

Refer to the description of the -C option in the **Note** on Page 223.

To turn on the desired checking mode from the Debugger window:

- ♦ **Choose Run Time 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 To Use Runtime Checking in Batch Mode

```
bcheck [ -access | -all | -leaks | -memuse ] [ -o
logfile ] [ -q ] [ -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. For more information see Doing More with Runtime Checking on page 257. Also refer to the `bcheck` (1) man page for details on its use.

19.3 Using Memory Access Checking

The simplest way to find access errors is to turn on access checking (`check -access`) 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.

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`)

19.3.1 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.

add - rAddress 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.

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 0xefffff67c
    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];
```

When RTC detects an access error, it reports the type and location of the error and returns control to the user unless the dbxenv variable `rtc_auto_continue` is set to **on**. If the Debugger GUI is present, RTC shows the location of the error in the source code display. Setting `rtc_auto_continue` **on** causes RTC not to stop upon finding an error, but to continue running, automatically. It also causes all errors to be redirected to the value of the dbxenv `rtc_error_log_file_name.` See dbxenv Variables on page 255 for more details about dbxenv's related to RTC.

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. The dbx `cont` command runs the program until another error or a breakpoint is encountered, or until the program terminates.

The following sections provide a more detailed description of each type of access error reported by RTC

19.3.2 .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)
*/
        }
```

19.3.3 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) */
```

19.3.4 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) */
```

19.3.5 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) */
        }
```

19.3.6 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) */
```

19.3.7 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) */
```

19.3.8 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) */
```

19.3.9 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) */
```

19.3.10 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 */

19.3.11 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 */

19.4 Memory Leak Errors

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 pointing to the blocks, programs cannot even reference them, much less free them.

Memory leaks result in increased virtual memory consumption and generally result in memory fragmentation. This may slow down the performance of your program and the whole system.

Typically, memory leaks occur because allocated memory is not freed and you loose a pointer to the allocated block. Here are some examples of memory leaks:

```
Example 1:

void
foo()
{
    char *s;
    s = (char *) malloc(32);

    strcpy(s, "hello world");

    return; /* no free of s.  Once foo returns, there is no pointer pointing */
            /* to the malloc'ed block, so that bloc
k is leaked. */
}

Example 2:

Sometimes memory leak may result from incorrect use of the API.

void
printcwd()
{

    printf("cwd = %s\n", getcwd(NULL, MAXPATHLEN));

    return; /* libc function getcwd() returns a pointer to malloc'ed */
            /* area when the first argument is NULL, program should */
            /* remember to free this.  In this case the block is not freed */
            /* and results in leak. */
}
```

Memory leaks can be avoided by following a good programming practice of always freeing memory when when it is no longer needed and paying close attention to library functions which return allocated memory. If you use such functions, remember to free up the memory approproiately.

19.4.1 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 was 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.

19.4.2 Using Memory Leak Checking

Note – RTC leak checking requires use of the standard `libc malloc/free/realloc` or allocators based on those functions

---

Memory leaks checking can be turned on via `check -leaks` or with `check -memuse` commands. RTC detects the following memory leak errors:

- Memory Leak (`mel`)
- Possible leak — Address in Register (`air`)
- Possible leak — Address in Block (`aib`)

The following discussion describes the commands for leaks checking:

`check -leaks [-frames` *n*`] [-match` *m* `]`

Turns on leak checking.

`-frames` *n* implies that up to *n* distinct stack frames will be displayed when reporting leaks.

`-match` *m* is used for combining leaks; if the call stack at the time of allocation for two or more leaks matches *m* frames, then these leaks are reported in a single combined leak report, as explained below.

The default value of *n* is 8 or the value of *m* (whichever is larger). Maximum value of *n* is 16. The default value of *m* is 2.

With leak checking turned on, you get an automatic leak report when the program exits. All leaks (including possible leaks) will be reported at that time, provided the program has not been killed using the `kill` command. However you can ask for leak report at any time by using the `showleaks` command.

`showleaks [-a] [-m` *m*`]`

Reports new memory leaks that occurred since the last `showleaks` command. `showleaks` also reports a summary of all the blocks in use.

`-a` Show all the leaks generated so far (not just the leaks since the last `showleaks` command).

`-m` *m* This is used for combining leaks; if the call stack at the time of allocation for two or more leaks matches *m* frames, then these leaks are reported in a single combined leak report. If the -m option is given, it overrides the global value of *m* (specified with the `check -match` command). The default value of *m* is 2 or (if specified) the global value last given with the `check` command.

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 controlled by the *number-of-frames-to-match* parameter specified by the `-match` *m* option on a `check -leaks` or the `-m` option of the `showleaks` command. If the call stack at the time of allocation for two or more leaks matches to *m* frames to the exact program counter level, these leaks are reported in a single combined leak report.

Consider the following three call sequences:

Block 1	**Block 2**	**Block 3**
[1] malloc	[1] malloc	[1] malloc
[2] d() at 0x20000	[2] d() at 0x20000	[2] d() at 0x20000
[2] c() at 0x30000	[2] c() at 0x30000	[2] c() at 0x31000
[2] b() at 0x40000	[2] b() at 0x41000	[2] b() at 0x40000
[2] a() at 0x50000	[2] a() at 0x50000	[2] a() at 0x50000

If all of these blocks lead to memory leaks, the value of *m* determines whether the leaks are reported as separate leaks or as one repeated leak. If *m* is 2, Blocks 1 and 2 are reported as one repeated leak because the 2 stack frames above `malloc()` are common to both call sequences. Block 3 will be reported as a separate leak because the trace for `c()` does not match the other blocks. For *m* greater than 2, RTC reports all of them as separate leaks. (The `malloc` is not shown on the leak report.)

In general, the smaller the value of *m*, the fewer individual leak reports and the more combined leak reports are generated. The greater the value of *m*, the fewer combined leak reports and the more individual leak reports are generated.

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.

19.4.3 Understanding the Memory Leaks Report

The following is a typical leaks report:

```
Memory Leak (mel):
Found 2 leaked blocks with total size 64 bytes
At time of each allocation, the call stack was:
         [1] true_leak() at line 220 in "leaks.c"
         [2] foo() at line 224 in "leaks.c"

Possible memory leak -- address in block (aib):
```

```
Found 2 leaked blocks with total size 16 bytes
At time of each allocation, the call stack was:
        [1] in_block() at line 177 in "leaks.c"
        [2] bar() at line 181 in "leaks.c"

Memory Leak (mel):
Found leaked block of size 8 bytes at address 0x26ea8
At time of allocation, the call stack was:
        [1] true_leak() at line 220 in "leaks.c"
        [2] main() at line 87 in "leaks.c"
```

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

RTC prints the following information for combined memory leak reports:

number	Number of leaked blocks whose call stack matched to m frames.
size	Total combined size of all the leaked blocks.
stack	The stack frames that were common to all blocks at the time of allocation.

19.4.4 Fixing Memory Leaks

Once you have obtained a memory leak report from RTC, here are some general guidelines as to how to go about fixing the memory leaks. The most important thing in fixing a leak is to determine where the leak is happening. The leak report generated will tell you the allocation trace of the leaked block.

This is the place where the leaked block was allocated from. You can then look at execution flow of your program and see how the block was used. If it is obvious where the pointer was lost the job is easy, otherwise you can use `showleaks` to narrow your leak window. `showleaks` by default gives you only the new leaks created since the last showleaks command. You can run `showleaks` repeatedly to to narrow the window where the block was leaked.

The following discussion provides a more detailed description of each type memory leak error in an RTC report.

19.4.5 *Memory Leak* (`mel`)

Problem: An allocated block has not been freed, and no reference to the block exists anywhere in the program.
Possible causes: Program failed to free a block that is no longer used.

Example:
```
char *ptr;
    ptr = (char *)malloc(1);
    ptr = 0;
/* Memory leak (mel) */
```

19.4.6 Address in Register (`air`)

Problem: A possible memory leak. An allocated block has not been freed, and no reference to the block exists anywhere in program memory.
Possible causes: The only reference(s) to an allocated block are contained in registers. This can occur legitimately if the compiler keeps a program variable only in a register instead of in memory. The compiler often does this for local variables and function parameters when optimization is turned on. If this error occurs when optimization has not been turned on, it is likely to be an actual memory leak. This can occur, for example, if the only pointer to an allocated block goes out of scope before the block is freed.
Example:

```
        if (i == 0) {
                char *ptr = (char *)malloc(4);
                /* ptr is going out of scope */
        }
     /* Memory Leak or Address in Register */
```

19.4.7 Address in Block (`aib`)

Problem: A possible memory leak. There is no reference to the start of an allocated block, but there is at least one reference to an address within the block.
Possible causes: The only pointer to the start of the block is incremented.
Example:

```
char *ptr;
main()
{
        char *ptr = (char *)malloc(4);
        ptr++;            /* Address in Block */
}
```

19.5 Showing Memory Usage

Note – RTC leaks checking requires use of the standard `libc` `malloc/free/realloc` or allocators based on those functions

---

RTC lets you see all the heap memory in use. You can use this information to get a sense of where memory gets allocated in your program or which program sections are using the most dynamic memory. This information can also be useful in reducing the dynamic memory consumption of your program and may help in performance tuning

19.5.1 Using Memory Use Checking

The following describes commands related to memory use checking.

`check -memuse [-frames` *n*`] [-match` *m*`]`

turns on memory use checking.

`-frames` *n*

n frames are listed in showing the allocation trace of the block in use.

`-match` *m*

Because there may be many of blocks in use in the program, RTC automatically combines the blocks allocated from the same execution trace into one report. The decision to combine the report is controlled by value of m. If the call stack at the allocation of two or more blocks matches m frames (to the exact program counter level), then these blocks are reported in a combined block in use report. The way blocks are combined is similar to leaks report. See the example under *Understanding the Memory Leaks Report,* earlier in this chapter.

Turning on memory use checkoing also turns on leaks checking. In addition to a leak report at the program exit, you also get a blocks in use (biu) report.

You can get a memory use report at any time during program execution by using the `showmemuse` command. Executing `showmemuse` generates a report showing all blocks of memory in use.

```
showmemuse [-a] [-m m] [-n num] [-v]
```

Report the top *num* blocks-in-use records sorted by size. Only the new blocks since the last showmemuse command are shown.

`-a` (all) specifies that all blocks in use be shown.

`-v` (verbose) is not specified, only one line per record is printed.

-m is used for combining the blocks-in-use report. If the call stack at the time of allocation for two or more blocks matches *m* frames, these blocks are reported in a single combined report. If the -*m* option is given, it overrides the global value of m.

`-n` *num* show up to the *num* records in the output. The default is 20.

Here is an example of `showmemuse:`

```
dbx showmemuse -n 5
Checking for memory use...

Blocks in use (biu) report:

  Total   % of   Num of    Avg     Allocation trace
  Size     All   Blocks    Block
                           Size
========= ====== ======  ======== =======================================
    65536 21.50%      4     16384  calloc < Heap::get_block < Heap::alloc < operator new
    25116  8.24%     91       276  allfeat < init_file < flush_config < init_license
    16488  5.40%      1     16488  clnt_dg_create < clnt_tli_create < load_dom_binding < __yp_dobind
    16384  5.37%      1     16384  clnt_vc_create < clnt_tli_create < getclnt < __clnt_create_loopback
    16384  5.37%      1     16384  calloc < Heap::get_block < Heap::alloc < new_block

Blocks in use Summary:
        blocks in use:     3321   total size:  304797 bytes

dbx  cont
.....
.....
dbx  showmemuse -v -n 2
Checking for memory use...
Block in use (biu):
Found 27 blocks totaling 442368 bytes (41.47% of total; avg block size 16384)
At time of each allocation, the call stack was:
        [1] calloc() at 0xef4adacc
        [2] Heap::get_block() at line 127 in "heap.cc"
        [3] Heap::alloc() at line 259 in "heap.cc"
        [4] operator new() at line 222 in "heap.cc"

Block in use (biu):
Found 2 blocks totaling 311296 bytes (29.18% of total; avg block size 155648)
At time of each allocation, the call stack was:
        [1] calloc() at 0xef4adacc
        [2] Heap::get_block() at line 127 in "heap.cc"
        [3] Heap::heap_configure() at line 157 in "heap.cc"
        [4] readnames() at line 1773 in "object.cc"

Blocks in use Summary:
        blocks in use:      426   total size: 1066574 bytes
```

When memory use checking is on, at program exit an implicit `showmemuse -a -n 20` is performed. You can get a verbose output at the program exit time by using `rtc_biu_at_exit` dbxenv variable.

Possible values for `rtc_biu_at_exit` are follows:

`on` - When memuse checking is `on` print non-verbose report at exit

`verbose` - When memuse checking is `on` print verbose report at exit

`off` - No memory use report is generated at exit. (default value)

`check -memuse` sets the value of `rtc_biu_at_exit` to `on.`

With `check -all`, the value of `rtc_biu_at_exit` is unchanged, so when memory use checking is turned on with `check -all`, by default no memory use report is generated at program exit.

19.5.2 Understanding the Memory Use Report

The RTC Memory Use Report contains the following information in the non-verbose mode. It prints one line per record.:

```
Total size of the record

% of all - Record size as a % of all the blocks-in-
use size.

Number of Blocks in this record

Average Block Size for this allocation trace

Simple allocation trace giving only function names.
```

The `verbose` mode prints all of this information in more detail..

19.6 Error Suppression

RTC provides a powerful error suppression facility that allows great flexibility in limiting the number and types of errors reported. If an error occurs that is suppressed, then no report is given, and the program continues as if no error had occurred.

The following kinds of suppression are available:

- Suppression by scope and type

  You must specify which type of errors to suppress. You can specify to which parts of the program the suppression should apply. The options are:

  Global — if no scope is specified, it is taken to be global scope and applies to the whole program.

  Load Object — applies to an entire loadobject, such as a shared library.

  File — applies to all functions in a particular file.

  Function — applies to a particular function.

  Line — applies to a particular source line.

  Address — applies to a particular instruction at an address.

- Suppression of last error

  You can tell RTC to suppress the last (most recent) error to prevent repeated reports of the same error. You can also tell RTC to do this automatically, for all errors, by using the dbx environment variable `rtc_auto_suppress`. See dbxenv Variables on page 255 for more information on `rtc_auto_suppress`.

- Others

  You can use the dbx environment variable `rtc_error_limit` to control the suppression. See dbxenv Variables on page 255 for more information on `rtc_error_limit`.

Error suppression is done by using the `suppress` command. Suppression can be undone by using the `unsuppress` command. Suppression is persistent across `run` commands within the same debug session, but not across `debug` commands.

19.6.1 Default Suppressions

To detect all errors RTC does not require the program be compiled -g. However, symbolic (`-g`) information is sometimes needed to guarantee the correctness of certain errors (mostly read from uninitialized memory). For this reason certain errors (`rui` for `a.out` and `rui aib air` for shared libraries) are suppressed by default if no symbolic information is available. This behavior can be changed by using the `-d` option of the suppress/sunsuppress commands.

For example, executing:

```
unsuppress -d rui
```

causes RTC to no longer suppress `read from uninitialized memory` (rui), in the code which does not have symbolic information (that was compiled non `-g`).

19.6.2 Error Suppression from dbx

From dbx, error suppression is set by typing the `suppress` command directly. See Command Syntax on page 249 for details on the `suppress` and `unsuppress` commands.

Examples:

In the following examples `main.cc` is a file name, `foo` and `bar` are functions and `a.out` is name of an executable.

`suppress mel in foo`

Do not report memory leaks originating in function `foo`

`suppress biu in libc.so.1`

Suppress reporting blocks in use allocated from `libc.so.1`

`suppress rui in a.out`

Suppress read from uninitialized in `a.out`

`suppress rua in main.cc`

Do not report read from unallocated in file `main.cc`

`suppress duf at main.cc:10`

Suppress duplicate free resulting at line 10 of `main.cc`

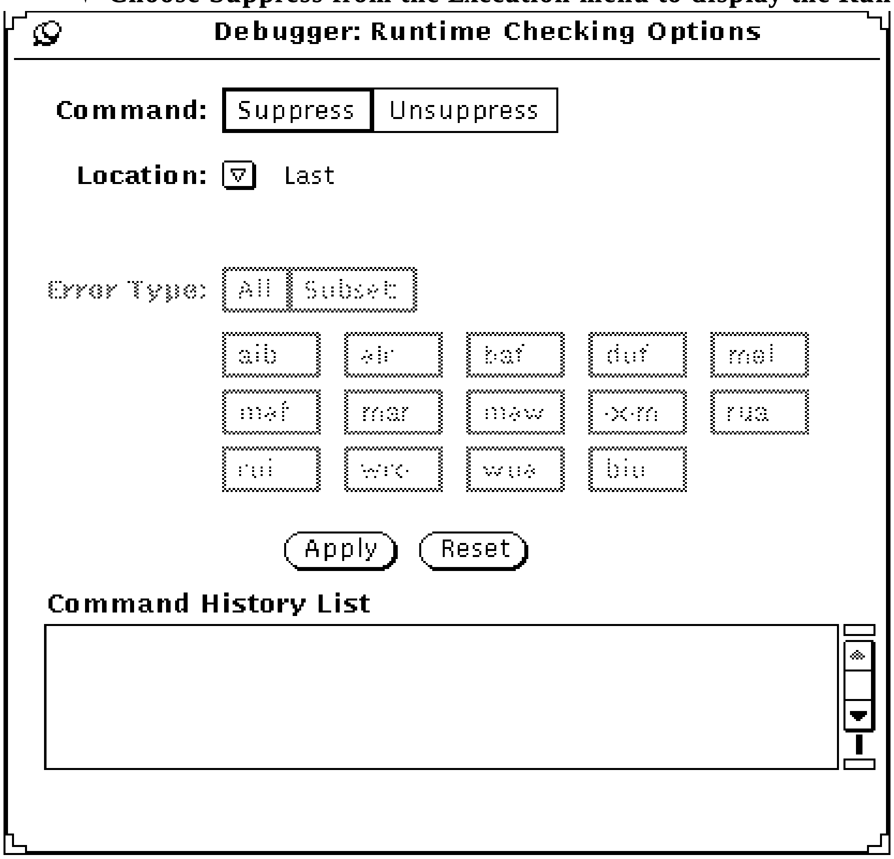
`suppress all in bar`

Suppress reporting of all errors in function `bar`

19.6.3 Error Suppression from the Debugger

You can select suppress from the Debugger window.

♦ **Choose Suppress from the Execution menu to display the Run Time**

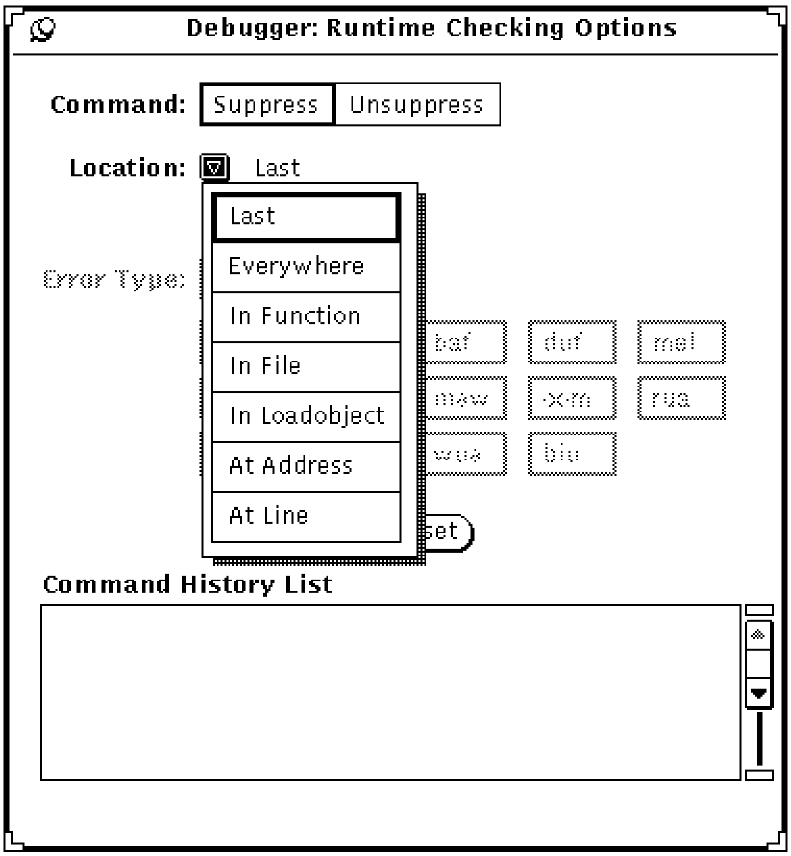


Checking Options window

You can select the location specifier from the Run Time Checking Options window.

♦ **Select the command you want, Suppress or Unsuppress, from the Command choice item.**

♦ **Press the Location menu button.**



The Location specifier options tell which area of the program to suppress or unsuppress the error:

Last

Suppress/Unsuppress the last (most recent) error.

Everywhere

Suppress/Unsuppress the selected Error Types throughout the program.

In Function

Suppress/Unsuppress the selected Error Types in the named function.

In File

Suppress/Unsuppress the selected Error Types in the named file.

In Loadobject

Suppress/Unsuppress the selected Error Types in loadobjects. The Loadobject name given must be valid. The name can be either the complete path or just the basename of the loadobject. To view a list of valid load objects, type

```
(dbx/debugger) loadobjects
```

At Address

Suppress/Unsuppress the selected Error Types at the given address.

At Line

Suppress/Unsuppress the selected Error Types at the given address.

Selecting the Error Type

You can select all error types or a particular error type:

- ♦ **Choose** *All* **from the Error Type choice setting to include all errors for suppression or unsuppression. -OR-**
- ♦ **Choose** *Subset:* **from the Error Type choice setting to activate the list of error types from which to make individual selections. Select as many types of errors as you want to have suppressed or unsuppressed.**

Press the Apply button to activate the selection. Press Reset to change the settings back to the factory defaults.

The Command History list shows the command list you issued in sequential order. Because RTC commands are cumulative, this list helps you see which suppressions are in effect.

19.6.4 Using Suppression to Manage Errors

Dealing with Large Programs

For the initial run on a large program, 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 rtc_auto_suppress 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 rtc_error_limit` *n* instead.

19.7 Command Syntax

19.7.1 Command `check|uncheck`

All forms of the check and uncheck forms are described below.

- `check`

  Print the current status of Runtime Checking.

- `check -access`

  Turns on access checking.

- `check -leaks [ -frames` *n* `] [ -match` *m* `]`

  Turns on leak checking.

  `-frames` *n* implies that up to *n* distinct stack frames will be displayed when reporting leaks.

  `-match` *m* is used for combining leaks. if the call stack at the time of allocation for two or more leaks matches *m* frames, these leaks are reported in a single leak report. However, you can ask for a leak report at any time (refer to the `showleaks` command).

- `check -memuse [ -frames` *n* `] [ -match` *m* `]`

  Turns on memory use checking. `check -memuse` also implies `check -leaks.` In addition to a leak report at the program exit, you also get a memory use report. At any time during program execution, you can see where all the memory in the program has been allocated (see `showmemuse`).

  `-frames` *n* implies thet up to *n* distinct stack frames will be displayed while reporting memuse/leaks.

  `-match` *m* is used for combining these reports. If the call stack at the time of allocation for two or more leaks, matches *m* frames, these are reported in a single combined memuse/leaks report.

  The default value of *n* is 8, or the value of *m*, whichever is greater.

  The maximum value of *n* is 16.

  The default value of *m* is 2.

  `check -memuse` sets the value of `rtc_biu_at_exit` to **on**.

  I f you have entered `check -leaks` and `rtc_biu_at exit = on`, no `biu` report will be shown at exit because the value of `rtc_biu_at_exit` is used only if `-memuse` is `on`.

- `check -all [ -frames n ] [ -match m ]`

Equivalent to `check -access; check -memuse [-frames n] [-match m]` The value of dbxenv `rtc_biu_at_exit` is not changed with `check -all`. By default no memory use report will be generated at exit. See dbxenv Variables on page 255 for a the description of the `rtc_biu_at_exit` variable.

- `uncheck`

  Disable checking of memory access, leaks, or usage (RTC)

- `uncheck -access`

  Turn off access checking

- `uncheck -leaks`

  Turn off leak checking

- `uncheck -memuse`

  Turn off memuse checking (leaks checking is also turned off).

- `uncheck -all`

  Equivalent to `uncheck -access; uncheck -memuse`

- `uncheck [`*funcs*`] [`*files*`] [` *loadobjects* `]`

  Equivalent to `suppress all in` *funcs files loadobjects*`;`

- `check` *function* file* loadobject**

This format of the `check` command allows you to turn on checking in specific functions, modules, and loadobjects while leaving it turned off for the rest of the program.

This command is equivalent to:

```
suppress all
unsuppress all in function* file* loadobject*
```

The command operates cumulatively. For example, the three commands

```
check main
check foo
check f.c
```

are equivalent to:

```
suppress all
unsuppress all in main
unsuppress all in foo
unsuppress all in f.c
```

Notice that the `suppress all` command is only applied once, leaving checking turned on for `main`, `foo`, and f.c.

- `uncheck` *function** *file** *loadobject**

  is equivalent to:

  `suppress all in` *function** *file** *loadobject**

19.7.2 *Command* `showleaks`

`showleaks [-a] [-m` *m*`]`

Reports new memory leaks that occurred since the last `showleaks` command. `showleaks` also reports a summary of all the blocks in use.

`-a` Show all the leaks generated so far (not just the leaks since the last `showleaks' command).

`-m` *m* This is used for combining leaks; if the call stack at the time of allocation for two or more leaks matches *m* frames, then these leaks are reported in a single combined leak report. If the `-m` option is given, it overrides the global value of *m* (specified with the `check` command). The default value of *m* is 2 or (if specified) the global value last given with the check command.

19.7.3 *Command* `showmemuse`

Executing this command wiil generate a report showing all blocks of memory in use.

- `showmemuse [-a] [-m m] [-n` *num*`] [-v]`

  Report the top *num* blocks-in-use records sorted by size. Only the new blocks since the last showmemuse command are shown.

`-a` (all) specifies that all blocks in use be shown.

`-v` (verbose) is not specified, only one line per record is printed.

`-m` is used for combining the blocks-in-use report. If the call stack at the time of allocation for two or more blocks matches *m* frames, these blocks are reported in a single combined report. If the *-m* option is given, it overrides the global value of m.

`-n` *num* show up to the *num* records in the output The default is 20.

At exit, a `showmemuse -a -n 10 -v` command is implicitly performed if `check -memuse` is on.

19.7.4 *Command* `suppress|unsuppress`

`{ suppress | unsuppress } -d [` *error type*`...[ in` *`loadobject`* `]`
`{ suppress | unsuppress } -reset`
`{ suppress | unsuppress } -last`
`{ suppress | unsuppress }`

{ `suppress` | `unsuppress` } [*error type*... [`l`*ocation specifier*]]

`suppress -r` *`id`* `...`

`suppress -r 0 | all | -all`

Some, or all files in a loadobject may not be compiled with the `-g` switch. This implies that there is no debugging information available for functions that belong in these files. RTC uses some default suppression in these cases.

To get a list of these defaults:

`{ suppress | unsuppress } -d`

To change the defaults for one loadobject:

`{ suppress | unsuppress } -d [error type] [ in` *`loadobject`*
`]`

To change the defaults for all loadobjects:

`{ suppress | unsuppress } -d [`*error type*`]`

To reset these defaults to the original settings:

```
suspress -reset
```

To suppress/unsuppress the most recent error:

```
{ suppress | unsuppress } -last
```

This command applies only to access errors and not to leaks errors.

To display the history of the suppress commands.

```
{ suppress | unsuppress }
```

Turn error reports on or off for the specified error types for the specified location.

{ suppress | unsuppress } [*error type*... [*location specifier*]]

Remove the suppress or unsuppress events as given by the id(s). (id(s) can be obtained by doing `suppress`.

{ suppress } -r <*id*>...

```
{ suppress} -r 0 | all | -all
```

Remove all the (un)suppress events as given by `suppress`.

error type:

`aib` - Address in block

`air` - Address in register

`all` - all errors

`baf` - Bad free

`duf` - Already freed

`maf` - Misaligned free

`mar` - Misaligned read

`maw` - Misaligned write

`mel` - Memory leak

`oom` - Out of memory

`rua` - Read from unallocated memory

`rui` - Read from write-only memory

`wro` - Write to read-only memory

`wua` - Write to unallocated memory

`biu` - Block in use (allocated memory). Though this is not an error, you can use `` `biu' `` just like *errors* in the suppress commands.

Table 19-3 shows the error type location specifier.

Table 19-1 Location Specifier

in loadobject	all functions in the designated program or library (*)
in *file*	all functions in *file*
in *function*	named *function*
at *line specifier*	at source *line*
addr *address*	at hex *address*

To see a list of the `loadobjects,` type the `loadobjects` command in the Debugger command pane or at the dbx prompt. Either the full pathname or the basename of the loadobject can be used.

If the *location specifier* is blank, the command applies globally to the program.

Only one *location specifier* may be given per command.

line specifier:

lin # - *line* number

file:line # - particular *line* in designated *file*

19.7.5 `dbxenv` *Variables*

The following `dbxenv` variables control the operation of RTC:

`dbxenv rtc_auto_continue` {on | off}

The default is: `off`

`rtc_auto_continue on` causes RTC not to stop upon finding an error, but to continue running, automatically. It also causes all errors to be redirected to the `rtc_error_log_file_name.`

`dbxenv rtc_auto_suppress` {on | off }

The default is: `off`

`rtc_auto_suppress on` causes a particular error at a particular location to be reported only the first time it is encountered, and suppressed thereafter. This is useful, for example, for preventing multiple copies of the same error report when an error occurs in a loop which is executed many times.

Note – If you are using the `bcheck` utility, the default setting for `dbxenv rtc_auto_suppress` is `on`. You turn off `rtc_auto_suppress` by using the `-s` *script* option (see Batch Mode Operation (bcheck) on page 257).

`dbxenv rtc_biu_at_exit` *`on | off | verbose`*

This variable is used when `check -memuse` is **on** (either implicitly or via `check -all`). If the value of the variable is **on**, a non-verbose memory use (blocks in use) report will be produced at program exit.

If the value is `verbose,` a verbose memory use report will be produced at program exit. The value **off** causes no output.

This variable has no effect on the interactive `showmemuse` command.

Default: `off`

`dbxenv rtc_error_log_file_name` {`filename`}

The default is: /`tmp/dbx.errlog.`*`pid`*

`rtc_error_log_file_name` redirects RTC error messages to the designated file instead of to the standard output of dbx.

The program does not automatically stop when run time errors are detected in batch mode. All error output is directed to your `rtc_error_log_file_name filename.` 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 `rtc_error_log_file_name.`

To redirect all errors to the terminal, set the `rtc_error_log_file_name` to `/dev/tty`.

Note – If the `rtc_error_log_file_name`, *filename*, already exists, the contents of that file are erased before the batch output is redirected to that file.

`dbxenv rtc_error_limit` *n*

The default is: 1000.

n is the maximum number of errors that RTC reports. The error limit is used separately for access errors and leak errors. For example, if the error limit is set to 5, then a maximum of 5 access errors and 5 memory leaks are shown in both the leaks report at the end of the run and for each `showleaks` command you issue.

If you want to permanently change any of these variables from their default values, place the `dbxenv` commands in the `$HOME/.dbxrc` file. Then, your preferred values are used whenever you use RTC.

19.8 Doing More with Runtime Checking

19.8.1 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`.

The `-q` option makes `bcheck` completely quiet. `bcheck` will return with the same status as the ‘program’. This is useful when you want to use `bcheck` in scripts or makefiles.

`bcheck` can perform memory leaks checking, memory access checking, or memory use checking, or all. Its default action is to perform leaks checking only.

Note – When using the `bcheck` utility, the default setting for `dbxenv rtc_auto_suppress` 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 memuse checking on `cc` quietly and exit with normal exit status:

```
bcheck -all -q cc -c prog.c
```

You can also enable batch mode directly from within dbx by setting the following dbx variable:

```
(dbx) dbxenv rtc_auto_continue on
(dbx) dbxenv rtc_error_log_file_name filename
```

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 error log file. The number of stack frames can be controlled using the `dbxenv` variable `stack_max_size`.

Note – If the file *your_logfile_name* already exists, `bcheck` erases the contents of that file before it redirects the batch output to it.

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.**

19.8.2 Using Fix & Continue with Runtime Checking

You can use RTC along with Fix and Continue (see Chapter 20, “Fix and Continue” for a description) to rapidly isolate and fix programming errors.

RTC in conjuction with fix and continue provides a really powerful combination and may save you a lot of debugging time. Here is an example which illustrates this.

```
yourmachine: cat -n bug.c
     1  #include stdio.h
     2  char *s = NULL;
     3
     4  void
     5  problem()
     6  {
     7          *s = 'c';
     8  }
     9
    10  main()
    11  {
    12          problem();
    13          return 0;
    14  }
yourmachine: cat -n bug-fixed.c
     1  #include stdio.h
     2  char *s = NULL;
     3
     4  void
```

```
problem()
{

    s = (char *)malloc(1);
    *s = 'c';
}

main()
{
    problem();
    return 0;
}
```

```
yourmachine46: cc -g bug.c
yourmachine47: dbx -C a.out
Reading symbolic information for a.out
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for librtc.so
Reading symbolic information for libc.so.1
Reading symbolic information for libintl.so.1
Reading symbolic information for libdl.so.1
Reading symbolic information for libw.so.1
dbx check -access
access checking - ON
dbx run
Running: a.out
(process id 15052)
Enabling Error Checking... done
```

```
Write to unallocated (wua):
Attempting to write 1 byte through NULL pointer
Current function is problem
     7        *s = 'c';
dbx pop
stopped in main at line 12 in file "bug.c"
    12        problem();
dbx #at this time we will edit the file, in this example just
copy the correct version
dbx cp bug-fixed.c bug.c
dbx fix
fixing "bug.c" ......
pc moved to "bug.c":14
stopped in main at line 14 in file "bug.c"
    14        problem();
dbx cont

execution completed, exit code is 0
dbx quit
The following modules in `a.out' have been changed (fixed):
bug.c
```

Remember to remake program.

19.9 Troubleshooting Tips

After error checking has been enabled for a program and the program is run, one of the following errors may be detected:

`system error: cannot recover;  Access checking disabled`

This error generally means that a fatal error occurred and error checking has been disabled. This happens if some system process level information is unavailable. In this case, the problem might be fixed by rebooting the system. Otherwise, call your Software Support representative.

`out of memory;  Access checking disabled`

RTC was unable to obtain memory it needed for correct operation. This indicates there was insufficient memory or swap space available on the system. Try one or more or the following corrective actions; then, rerun RTC.

Have the SysAdmin increase the amount of swap space available on the system.

Kill some unused or hung processes, or try again when the system is less heavily loaded.

Upgrade the amount of real memory in the system.

Move to another system that has more memory available.

`patch area too far;  Access checking disabled`

RTC was unable to find patch space close enough to a load object for error checking to be enabled.

See the following section for more information.

`librtc.so and dbx version mismatch;  Error checking disabled`

Have the SysAdmin reinstall the software.

19.9.1 rtc8m

Note – This applies to SPARC only.

To do access checking dbx/RTC replaces each load and store instruction with a branch instruction that branches to a patch area. This branch instruction has an 8Mb range. This means that if the debugged program has used up all the address space within 8Mb of the particular load/store instruction being replaced, there is no place to put the patch area.

If RTC can't intercept ALL loads and stores to memory it cannot provide accurate information and so disables access checking completely. Leakschecking is unaffected by this.

Dbx internally applies some strategies when it runs into this limitation and continues if it can rectify this problem. In some cases dbx cannot proceed; when this happens it will turn off access checking after printing an error message.

What to do if you run into this 8Mb limit:

Dbx provides some possible workarounds to users who have run into this limit. These workarounds require the use of a utility called `rtc_patch_area` (documented in the man page `rtc_patch_area(1)`).

This utility will create object files or shared object files which can be linked into the user's program and create patch areas for RTC to use.

There are two typical situations that can prevent dbx from finding patch areas within 8Mb of all the loads and stores in an executable image:

Case 1: The statically linked a.out file is too large.

Case 2: One or more dynamically linked shared libraries is too large.

When dbx runs into this limitation, it prints a message telling how much patch space it needs and directs you to the appropriate case (Case 1 or Case 2).

In Case 1, if the `a.out` patch space requested by dbx is around 8MB or less, you can use **rtc_patch_area** to make an object file to serve as a patch area and link it into the `a.out`.

Example:

After you have seen a message like the following:

```
Enabling Error Checking... dbx: warning: rtc: cannot find
patch space within 8Mb (need 6490432 bytes for ./a.out)
```

```
dbx: patch area too far (8Mb limitation); Access checking
disabled
```

(See `help rtc8M', case 1)

1. **Create an object file** `patch.o` **for a patch area:**

```
rtc_patch_area -o patch.o -size 6490432
```

(Note: the "-size" flag is optional; the default value is 8000000.)

2. Relink the `a.out`**, adding** `patch.o` **to the link line.**

If dbx has requested more than 8Mb of patch space, the `rtc_patch_area` workaround is unlikely to work; in that case the recommended workaround is to divide the a.out into smaller shared libraries.

In Case 2, the only possible workaround here is to rebuild the shared library with extra patch space.

Example:

After you have seen a message like the following:

```
Enabling Error Checking... dbx: warning: rtc: cannot find
patch space within 8Mb (need 563332 bytes for ./sh1.so)
```

```
dbx: patch area too far (8Mb limitation); Access checking
disabled
```

(See `help rtc8M', case 2)

1) Create an object file `patch.o` for a patch area:

```
rtc_patch_area -o patch.o -size 6490432
```

(Note: the "-size" flag is optional; the default value is 8000000.)

2) Relink `sh1.so`, adding `patch.o` to the link line.

3) Try RTC again with the new binary; if dbx requests patch space for another shared library, repeat steps 1-2 for that library.

If the patch space requested by dbx is more than 8Mb for a given shared library, the `rtc_patch_area` workaround is unlikely to work; in that case the recommended workaround is to split this shared library into smaller shared libraries.

19.9.2 rtc Patch Area

`rtc_patch_area` is a shell script that creates object files or shared library files that can be linked into the user's program to add patch area space to programs with large text, data, or bss images.

The object file (or shared library) created contains one RTC patch area of the specified size or 8000000 if size is not supplied.

The name of the resulting object file (or shared library) is written to the standard output.

Either the `-o` or `-so` options must be used.

`-so sharedlibname`

Specify the name of the shared library to be created. This name is then written to the standard output.

`-o objectname`

Specify the name of the object file to be created. This name is then written to the standard output. If the `-so` option was used and `-o` was not used the default will be a name in `/tmp` based on the LOGNAME environment variable.

`-size size`

Create a patch area of size bytes (default and reasonable maximum is 8000000).

`-cc compiler`

Use compiler instead of cc or acc to build the object file.

`rtc_patch_area -so sharedlibname`

`Examples:`

Generate a standard 8Mb patch area object file:

```
rtc_patch_area -o patch.o
```

Generate an object file containing a 100,000 byte patch:

```
rtc_patch_area -size 100000 -o patch.o
```

Generate a 1Mb patch area shared library:

```
rtc_patch_area -so rtc1M.so -size 1000000
```

Fix and Continue 20

Note – This feature is available only on Solaris.

This chapter is organized into the following sections:

20.1 Basic Concepts

The Fix and Continue feature of the Debugger allows you to modify a source file, and without leaving the Debugger, recompile the file and continue execution of the program. The advantages of using Fix and Continue are

- You do not have to relink the program.
- You do not have to reload the program into the Debugger.
- You can resume running the program from the fix location.

20.2 How Fix and Continue Operates

Before applying the `fix` command you need to edit the source. You can modify the source file either by enabling editing in the source display or by editing the file in your system editor. See Chapter 14, "Editing a Program" for an explanation of how to edit in the source display.

After you save your changes to the source, select **Fix** from the Execution menu. Once `fix` has been invoked, the Debugger calls the Compiler with the appropriate compiler options. The modified files are compiled and temporary shared object (`.so`) files are created. Semantic tests are done to check the safety of the fix by comparing the old and new files (the old file is the file before the latest edits were made; the new file is the file containing the latest edits).

The new object file is linked to your running process using the runtime linker. The program counter is moved from the old function to the beginning of the same line in the new function (if the function is on top of the stack being fixed). All the breakpoints in the old file are moved to the new file. The modified source is then displayed in the source display and you can resume debugging from the exact point in the source that you stopped at.

You can use Fix and Continue on files that have been compiled with or without debugging information, but there are some limitations in the functionality of Fix and Continue for files originally compiled *without* debugging information. Specifically, see the Note on static variables under, *Precautions*, in this Chapter. Also, you can use the `-g` option to do fixes that recompile with debugging information. See *Command Summary*, in this Chapter.

Solaris 2.x

You can fix shared objects (`.so`) files, but they have to be opened in a special mode. You must use either `RTLD_NOW|RTLD_GLOBAL` or `RTLD_LAZY|RTLD_GLOBAL` in the call to `dlopen`. This procedure works only for Solaris 2.x.

20.3 Source Modification Using Fix and Continue

You can modify sources in the following ways when using Fix and Continue:

- Add, delete, or change lines of code in functions.
- Add or delete functions.
- Add or delete global and static variables.

20.4 Restrictions

The Debugger might have problems when functions are mapped from the old file to the new file. To minimize such problems when editing a source file, you should be careful not to:

- Change the name of a function.
- Add, delete, or change the type of arguments to a function.
- Add, delete, or change the type of local variables in functions currently active on the stack (see the `pop` command for more information).
- Only the body of a C++ template function definition can be modified. Any change to the declaration of a template or to template instances requires a complete remake of the program.

20.5 Precautions

Resuming program execution after performing a Fix and Continue can cause errors that may not be directly related to the actual modifications to the source. Before resuming program execution after you have modified the source, you should be aware of the following conditions:

- If a function being modified is already instantiated on the stack (not the top frame), the program executes the old code (that is, the unmodified code) when returning to that frame.
- Breakpoints are moved (not copied) from old files to new files. Therefore, breakpoints in already instantiated functions on the stack in the old code are lost.
- Changes made to global variables are not undone by the `pop` or `fix` command. Use the `assign` command to manually reassign correct values to global variables.
- Values of static variables are normally retained across fixes. However, if the file was not originally compiled with debugging information, static variables in that file are reinitialized after each fix. You can use the `assign` command to manually reassign correct values to static variables.

- If the number of lines of source code has been changed by your edits, the placement of the breakpoints might be incorrect. The Debugger displays a warning message telling you that the breakpoints have moved to new lines. You should check the locations of the breakpoints with the `status` command in the new code before running the program.
- The program counter is moved to the beginning of the same line in the new (fixed) function. If the function has been modified, the program counter can end up at a wrong line. In that case, do one of the following:
  - Use the `pop` command. It pops one (or more) frames from the stack and a `cont` re-enters the function.
  - Use the `cont at` *linenum* command to continue from another line (effectively moving the program counter before executing).
  - Manually repair data structures (use the `assign` command to make them consistent with the program counter) before continuing.
  - Rerun the program.
- If you change a C++ template function definition, you must issue the `fix` command on all files that define instances of that template. In such cases you may need to use the `-f` option of `fix` to force fixing of files whose source is unchanged. See *Command Summary*, in this chapter.
- (Solaris 1.x) New global functions and variables can only be referenced from within the same module in which they are defined.

20.6 Example

The following example shows how a simple bug can be fixed with Fix and Continue. The application will get a segmentation violation in line 6 when trying to dereference a NULL pointer:

```
dbx[1] list 1,$
    1   #include <stdio.h>
    2
    3   char *from = "ships";
    4   void copy(char *to)
    5   {
    6       while ((*to++ = *from++) != '\0');
    7       *to = '\0';
    8   }
    9
    10  main()
    11  {
```

```
    12      char buf[100];
    13
    14      copy(0);
    15      printf("%s\n", buf);
    16      return 0;
    17  }
(debugger) run
Running: testfix
(process id 4842)
signal SEGV (no mapping at the fault address) in copy at line 6 in
    file "testfix.cc"
    6       while ((*to++ = *from++) != '\0');
```

Change line 14 to `copy` to `buf` instead of `0` and save the file, then do a fix:

```
    14      copy(buf);  <===  modified line
```

```
(debugger) fix
fixing "testfix.cc" .....
pc moved to "testfix.cc":6
stopped in copy at line 6 in file "testfix.cc"
    6       while ((*to++ = *from++) != '\0');
```

If the program is continued from here, it still gets a `SEGV` because the zero-pointer is still pushed on the stack. Use the `pop` command to pop one frame of the stack:

```
(debugger) pop
stopped in main at line 14 in file "testfix.cc"
    14      copy(buf);
```

If the program is continued from here, it will run, but it will not print the correct value because the global variable `from` has already been incremented by one. The program would print `hips` and not `ships`. Use the `assign` command to restore the global variable and then continue. Now the program prints the correct string:

```
(debugger) assign from = from-1
(debugger) cont
ships
```

20.7 Command Summary

The `fix` command, with no arguments, fixes just the current source file. More generally, `fix` commands have the following form:

`fix` [*options*] [*file*1, *file*2,...]

where

`-a` fixes all modified files.

`-f` tells the Debugger to force fixing the file, even if the source was not changed. This is useful, for example, if only a header file is modified.

`-c` tells the Debugger to print the compilation line (which may include some options added internally for use by dbx).

`-g` strips `-O` flags from the compilation line and adds the `-g` flag to it. This allows you to use `fix` to bring debugging information into a file originally compiled without it.

`-n` sets a no execution mode in the Debugger. Use this option when you want to list the source files to be fixed without actually fixing them.

`-v` sets verbose mode, in which the compilation and link lines are printed during a fix (this overrides the setting of the dbxenv variable `fix_verbose).`

`+v` sets a non-verbose mode (overrides the setting of the dbxenv variable `fix_verbose).`

file1, file2,... specifies a list of modified source files to fix.

When options `-v` or `+v` are not used, the value of the dbxenv variable `fix_verbose` determines the verbose mode (**on** or **off**). Its default value is **off**.

Beyond the `-g` option, the Debugger provides the following `ksh` facilities for customizing the compilation and link steps used by `fix.`

`_cb_fix_comp`

`_cb_fix_comp` <*compilation arguments*>

This ksh function, if defined, is invoked by dbx instead of calling the compiler directly during a `fix.` This is a good place to customize the compilation line by adding or changing options, modifying paths, etc. Use with caution.

For example:

```
_cb_fix_comp() {
    echo $*
    eval $*
}
```

This will just echo and execute the compilation line without modifications, whereas:

```
_cb_fix_comp() {
        eval "$(echo $* | sed -e 's! -g0 ! -g !')"
}
```

will replace `-g0` with `-g` (to turn off inlining) and execute the compiler.

The arguments of this function are from the original, unmodified compilation line, plus some options added internally for use by dbx.

_cb_fix_link

`_cb_fix_link` <*linker arguments*>

This ksh function, if defined, is invoked by dbx instead of calling the linker directly during a `fix`. This is a good place to customize the link line. For example, by adding or changing options or modifying paths. Use with caution.

This link creates a shared object which is opened with dlopen into the target process. It does *not* modify the a.out.

For example:

```
_cb_fix_link() {
        echo $*
        eval $*
    }
```

will just echo and execute the link line.

If fix is invoked with an option other than -a and without a filename argument, then it applies only to the current source file.

Note – Sometimes it may be necessary to modify a header (.h) file as well as a source file. To be sure that the modified header file is accessed by all source files in the program that include it, you must give as an argument to the fix command a list of all the source files that include that header file.

If you do not include the list of source files, only the primary source file is recompiled and only it includes the modified version of the header file. Other source files in the program continue to include the original version of that header file.

When fix is invoked, the Debugger looks for the source file's current working directory as of the time of compilation, before executing the compilation line. The Debugger might have problems locating the correct directory due to a change in the file system structure from compilation time to debugging time. To avoid this problem, use the command pathmap, which creates a mapping from one pathname to another. The mapping is applied to source paths and object file paths. For detailed information on pathmap, see Section 3.2.3, "Setting the Search Path with pathmap," on page 3-33. You can also t ype help pathmap in the command pane or see the dbx man pages.

Handling C++ Exceptions 21

This chapter describes the commands that are available for handling C++ exceptions and how the Debugger handles exceptions.

This chapter is organized into the following sections:

One reason a program stops running is if an exception occurs. Exceptions signal programming anomalies, such as division by zero or array overflow. To deal with these exceptions, you can set up `try` blocks to catch exceptions that have been raised by `throw` expressions elsewhere in the code.

The Debugger can also help you with exception handling. While debugging a program, the Debugger enables you to

- Catch unhandled exceptions before stack unwinding
- Catch unexpected exceptions.
- Catch specific exceptions whether handled or not before stack unwindin.
- Determine where a specific `throw` would be caught if it occurred at a particular point in the program

If you do a `step` after stopping at a throw point, control is returned at the start of the first destructor that gets executed during stack unwinding. If you do a `step` out of a destructor that is being executed during stack unwinding, control is returned at the start of the next destructor. When all the destructors have been executed, doing a `step` brings you to the catch block that is handling the throw.

21.1 Commands for Handling Exceptions

This section lists the exception handling commands that you can invoke when debugging a program.

`exception`

Use this command to display the type of an exception (the command is an alias for `print *$exception`). When the runtime is processing an exception (for example, during stack unwinding), the ksh variable `$exception` is set to point to that exception. This variable can be used at any time during debugging.

`intercept [-a |` *exception-type*`]`

You can intercept (“catch”) exceptions of a specific type before the stack has been unwound. Use this command to list all the types that are being intercepted. With the `-a` option, the command intercepts all types. Invoking the `intercept` command with a type adds the specified type to the intercept list. With no arguments, `intercept` lists the types being intercepted.

`unintercept [-a |` *exception-type*`]`

Use this command to remove exception types from the intercept list. With the `-a` option, the command removes all types from the list. Invoking the command with a specific type removes that type from the list. With no arguments, it lists the types being intercepted (same as `intercept`).

`intercept -x` <*exception-type*>

Use this form to exclude a particular type from being intercepted. For example, to intercept all types except `_int_`, you could enter:

```
intercept -a
intercept -x int
```

`unintercept -x` <*exception-type*>

Use this form to stop excluding a particular type from being intercepted (to allow it to be intercepted if `intercept -a` has been used.

`whocatches` *exception-type*

This command reports the location in the source that would catch the exception or informs you that the exception has no catch location. This command helps you to find out what would happen if an exception were thrown from the top frame of the stack.

`stop throw` <*type*>

This command is the same as `intercept` <*type*>.

`stop throw`

This command is the same as `intercept -a`.

`when throw` <*type*> {<*cmds*>;}

Execute <*cmds*> whenever <*type*> is thrown.

`when throw` {<*cmds*>;}

Execute <*cmds*> whenever any type is thrown.

The next section demonstrates how exception handling is done in the Debugger using a sample program containing exceptions.

21.2 Exception Handling in the Debugger

In the following example, an exception of type `int` is thrown in the function `bar` and is caught in the following catch block.

```
#include <stdio.h>

class c {
    int x;
  public:
    c(int i) { x = i; }
    ~c() {
        printf("destructor for c(%d)\n", x);
    }
};
```

```
11
12  void bar() {
13      c c1(3);
14      throw(99);
15  }
16
17  int main() {
18      try {
19          c c2(5);
20          bar();
21          return 0;
22      }
23      catch (int i) {
24          printf("caught exception %d\n", i);
25      }
26  }
```

The following transcript from the example program shows the exception handling features in the Debugger.

```
(dbx) intercept int
(dbx) intercept
int
(dbx) stop in bar
(2) stop in bar(void)
(dbx) run
Running: a.out
(process id 304)
stopped in bar at line 13 in file "foo.cc"
   13         c c1(3);
(dbx) whocatches int
int is caught at line 24, in function main (frame number 2)
(dbx) whocatches c
dbx: no runtime type info for class c (never thrown or
caught)
(dbx) cont
Exception of type int is caught at line 24, in function
main (frame number 4)
stopped in _ex_dbg_will_throw at 0xef76dac8
_ex_dbg_will_throw:    save    %sp, -96, %sp
Current function is bar
  14      throw(99);
(dbx) step
```

```
stopped in c::~c at line 8 in file "foo.cc"
   8      printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(3)
stopped in c::~c at line 9 in file "foo.cc"
   9    }
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
   8     printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(5)
stopped in c::~c at line 9 in file "foo.cc"
   9     }
(dbx) step
stopped in main at line 24 in file "foo.cc"
  24      printf("caught exception %d\n", i);
(dbx) step
caught exception 99
stopped in main at line 26 in file "foo.cc"
  26   }
```

Debugging with C++ Templates 22

This chapter is organized into the following sections:

The Debugger supports C++ templates. You can 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, such as

- Setting breakpoints at class or function template instantiations
- Printing a list of all class and function template instantiations
- Displaying the definitions of templates and instances
- Calling member template functions and function template instantiations
- Printing values of function template instantiations
- Displaying the source code for function template instantiations

This chapter describes a sample program containing function and class templates along with examples of commands called on the templates and the template instantiations.

22.1 Template Example

The following code example shows the class template `Array` and its instantiations and the function template `square` and its instantiations:

```
template<class C> void square(C num, C *result)
{
   *result = num * num;
}

template<class T> class Array
{
public:
   int getlength(void)
   {
      return length;
   }

   T & operator[](int i)
   {
      return array[i];
   }

   Array(int l)
   {
      length = l;
      array = new T[length];
   }

   ~Array(void)
   {
      delete [] array;
   }

private:
   int length;
   T *array;
};

int main(void)
{
   int i, j = 3;
   square(j, &i);

   double d, e = 4.1;
```

```
        square(e, &d);

        Array<int> iarray(5);
        for (i = 0; i < iarray.getlength(); ++i)
        {
           iarray[i] = i;
        }

        Array<double> darray(5);
        for (i = 0; i < darray.getlength(); ++i)
        {
           darray[i] = i * 2.1;
        }

        return 0;
    }
```

where

Array is a class template

square is a function template

Array<int> is a class template instantiation (template class)

Array<int>::getlength is a member function of a template class

square(int, int*) and square(double, double*) are function template instantiations (template functions)

22.2 Debugger Commands on Templates

This section shows examples of Debugger commands you can invoke on templates and template instantiations. Use the whereis command to find template instantiations and then use whatis to get the definitions of the templates and template instances. Once you know the class or type definitions, you can print values, display source listings, or set breakpoints. For detailed information on these commands, see their respective discussions in this manual, the dbx man page, or the Debugger help command.

22.2.1 Displaying Instantiations of Function and Class Templates

Use `whereis` to print a list of all occurrences of function or class instantiations of the specified function or class template:

Class template:

```
(debugger) whereis Array
in loadobject “a.out”
class template instance: ‘Array<int>
class template instance: ‘Array<double>
class template:‘template_doc_2.cc‘Array
```

Function template:

```
(debugger) whereis square
in loadobject "a.out"
function template instance: ‘square(double, double*)
function template instance: ‘square(int, int*)
function template:          ‘square
```

22.2.2 Displaying Template Definitions

Use `whatis` to print the definitions of function and class templates and instantiated functions and classes:

Class template:

```
(debugger) whatis Array
template<class T> class Array
To get the full template declaration, try ‘whatis -t
Array<int>’;
(debugger)
```

Function template:

```
(debugger) whatis square
```

Select the template definition via the Overload Display.

```
template<class C> void square(C num, C *result);
```

Class template instantiation:

```
(debugger) whatis -t Array<double>
class Array<double> {
public:
      int Array<double>::getlength();
      double &Array<double>::operator [](int i);
      Array<double>::Array<double>(int l);
      Array<double>::~Array<double>();
private:
      int length;
      double *array;
};
```

Function template instantiation:

```
(debugger) whatis square(int, int*)
void square(int num, int *result);
```

22.2.3 Setting Breakpoints in Function and Class Templates

You can set breakpoints at function template instantiations, function templates, and template classes.

Use `stop inclass` to set breakpoints at all member functions of a template class:

```
(debugger) stop inclass Array<int>
(2) stop inclass Array<int>
```

Use `stop infunction` to set breakpoints at all instances of the specified function template:

```
(debugger) stop infunction square
(9) stop infunction square
```

Use `stop in` to set a breakpoint at a member function of a template class or at a template function:

Class template instantiation:

```
(debugger) stop in Array<int>::Array<int>(int l)
(2) stop in Array<int>::Array<int>(int)
```

Function instantiation:

```
(debugger) stop in square(double, double*)
(6) stop in square(double, double*)
```

22.2.4 Calling a Function Instantiation

Use `call` to explicitly call a function instantiation or a member function of a class template, provided you are stopped in a context where the call is appropriate. If the Debugger is unable to choose the correct instance, a pop-up menu is displayed to allow you to choose the specific instance.

Function instantiation:

Stopped for example, at the `return` statement in `main`:

```
(debugger) call square(8, &i)
(debugger) print i
i = 64
```

22.2.5 Printing the Value of a Function Instantiation

Use `print` to evaluate a function instantiation or a member function of a class template:

```
(debugger) print iarray.getlength()
iarray.getlength() = 5
```

22.2.6 Printing the `this` pointer

Use `print` to evaluate the `this` pointer:

```
(debugger) whatis this
class Array<int> *this;
(debugger) print *this
```

```
*this = {
      length   = 5
      array    = 0x21608
}
```

22.2.7 Listing a Function Instantiation

Use `list` to print the source listing for the specified function instantiation:

```
(debugger) list square(int, int*)
```

The source display in the Debugger window shows the source for `square`.

Korn Shell 23

The Debugger command language is based on the syntax of the Korn Shell[1] (ksh 88), including I/O redirection, loops, built-in arithmetic, history, and command-line editing (only in command-line mode; not available from the Debugger).

This chapter lists the differences between ksh-88 and the Debugger command language.

This chapter is organized into the following sections:

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

23.1 Features of ksh-88 not Implemented

The following features of ksh-88 are not implemented in the Debugger:

- `set -A` *name* for assigning values to array *name*
- `set -o` particular options: **allexport bgnice gmacs markdirs noclobber nolog privileged protected viraw**
- `typeset -l -u -L -R -H` attributes
- backquote (**`…`**) for command substitution (use **$(...)** instead).
- [[*expr*]] compound command for expression evaluation
- **@**(pattern[| pattern] **…)** extended pattern matching
- co-processes (command or pipeline running in the background that communicates with your program)

23.2 Extensions to ksh-88

The Debugger adds the following features as extensions:

- **$** [*p* –> *flags*] language expression.
- `typeset -q` enables special quoting for user-defined functions
- csh-like `history` and `alias` arguments
- `set +o path` disables path searching
- **0xabcd** C syntax for octal and hexadecimal numbers
- `bind` to change Emacs-mode bindings
- `set -o hashall`
- `set  -o  ignoresuspend`
- `print -e` and `read -e` (opposite of `-r`, “raw”)
- dbx commands are built in

23.3 Renamed Commands

Particular dbx commands have been renamed to avoid conflicts with ksh commands.

- The dbx `print` command retains the name `print`. The ksh `print` command has been renamed `kprint`.
- The ksh `kill` command has been merged with the dbx `kill` command.
- The `alias` command is the ksh `alias`, unless in dbx compatibility mode.

These dbx commands have been renamed:

- *addr/fmt* is now `examine` *addr/fmt*.
- /*pattern* is now `search` *pattern*.
- ?*pattern* is now `bsearch` *pattern*.

23.4 The Debugger Startup Mode

During startup, the Debugger searches for `.dbxrc` first (ksh mode). The search order is:

- current directory `./.dbxrc`
- home directory `$HOME/.dbxrc`

If `.dbxrc` is not found, the Debugger searches for `.dbxinit` (dbx compatibility mode). The search order is:

- current directory `./.dbxinit`
- home directory `$HOME/.dbxinit`

If neither `.dbxrc` nor `.dbxinit` is found, the Debugger assumes ksh mode.

dbx and the Solaris Dynamic Linker 24

Note – This feature is available only on Solaris.

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.

This chapter is organized into the follwing sections:

24.1 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.

24.2 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.

24.2.1 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 "readins. . ." messages and 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.

24.2.2 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.

24.2.3 `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 6-82), can be used with the `when` command, and some shell programming, to help ease the burden of managing breakpoints in `dlopen` type shared objects.

24.2.4 `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.

24.2.5 *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.

24.3 *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.

Three exceptions:

- You cannot set a breakpoint in a `dlopen`’ed library before that library is loaded by `dlopen()`.
- You cannot set a breakpoint in a `dlopen`’ed “filter” library until the first function in it is called.

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`.

Part 4 — Appendixes

Debugger Commands A

This appendix is a reference to the Debugger commands that you can enter in the command pane.

It contains a table of the full set of commands and other topics arranged into functional groups.

A.1 On-Line Help in the Command Pane

On-line help for each Debugger command is available in the command pane:

To see the full list of commands by functional group, type:

```
(debugger) help
```

To see the syntax and a brief description of a particular command, type:

(debugger) **help** ***command_name***

The dbx man page contains a description of each command.

A.2 Debugger Commands By Functional Groups

The complete set of Debugger commands are listed below. Note that the groupings are the same as the groups that appear when you type `help` in the command pane with no argument.

In this list, * = Solaris 2.x only, ** = SPARC only.

A.2.1 Execution and Tracing

cancel	catch	clear	cont	delete	fix
fixed	handler	ignore	intercept	next	pop
replay	rerun	restore	run	runargs	save
status	step	stop	trace	unintercecept	when
whocatches					

A.2.2 Displaying and Naming Data

assign	call	demangle	dis	display	down
dump	examine	exists	frame	hide	inspect*
print	undisplay	unhide	up	whatis	where
whereami	whereis	which			

A.2.3 Accessing Source File

bsearch	cd	edit	file	files	func
funcs	line	list	loadobject	loadobjects	module
modules	pathmap	pwd	search	use	

A.2.4 Debugging Multiple Threads *

lwpp	lwps	thread	threads

A.2.5 Run Time Checking *

check	showleaks	showmemuse	suppress
uncheck	unsuppress		

A.2.6 Miscellaneous Commands

collector *	dalias	dbxbugreport	dbxenv	debug	detach
document	help	history	import	kalias	kill
language	quit	setenv	sh	source	!
!!					

A.2.7 Debugger

button	menu	toolenv	unbutton	unmenu

A.2.8 Machine Level

adb	examine	listi	nexti
stepi	stopi	tracei	wheni

A.2.9 Language Specific Information

C++	ObjC	fortran	fortran90

A.2.10 Other Topics

alias	array-sizing	callbacks	changes	changes301
.dbxrc	editing	-editor	events	FAQ
fix-pitfalls	follow-fork*	format	forwardref	invocation
ksh	lwpid	MT*	path	prettyprint
redirection	registers	rtc	rtcmt*	scope
signals	tid*			

The command `help` *cmdname* provides additional help for each command or topic. See `help changes` for new and changed features.

Use the `commands` command to see a one line summary of each command.

Operators Recognized by the Debugger

B

Operator	Description	C	C++	Pascal	Fortran
+	add	x	x	x	x
-	subtract	x	x	x	x
*	multiply	x	x	x	x
/	divide	x	x	x	x
div	divide			x	
mod				x	
%	integer remainder			x	
<<	left shift	x	x	x	x
>>	right shift	x	x	x	x
&	bitwise and	x	x	x	x
|	bitwise or	x	x	x	x
^	exclusive or	x	x	x	x
~	bitwise complement	x	x	x	x
&	address of	x	x	x	
*	contents of	x	x	x	x
<	less than	x	x	x	x

<table>
<tr><th>Operator</th><th>Description</th><th>C</th><th>C++</th><th>Pascal</th><th>Fortran</th></tr>
<tr><td>></td><td>greater than</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td><=</td><td>less than or equal to</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>>=</td><td>greater than or equal to</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>==</td><td>equal to</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>=</td><td>equal to</td><td></td><td></td><td>x</td><td></td></tr>
<tr><td>!=</td><td>not equal to</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td><></td><td>not equal to</td><td></td><td></td><td>x</td><td></td></tr>
<tr><td>!</td><td>not</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>&&</td><td>logical and</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>| |</td><td>logical or</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>sizeof</td><td>size of variable type</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>(type)</td><td>type cast</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>.</td><td>structure field reference</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>-></td><td>pointer to structure field reference</td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>::</td><td>C++ double colon scope resolution operator</td><td></td><td>x</td><td></td><td></td></tr>
<tr><td>?:</td><td>embedded “if-then-else” (C and C++ only)</td><td>x</td><td>x</td><td></td><td></td></tr>
<tr><td>+</td><td>unary plus</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>-</td><td>unary minus</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>and</td><td></td><td></td><td></td><td>x</td><td></td></tr>
<tr><td>or</td><td></td><td></td><td></td><td>x</td><td></td></tr>
<tr><td>^</td><td>pointer dereference</td><td></td><td></td><td>x</td><td></td></tr>
<tr><td>/=</td><td>fortran</td><td></td><td></td><td></td><td>x</td></tr>
<tr><td>//</td><td>fortran concut</td><td></td><td></td><td></td><td>x</td></tr>
<tr><td>**</td><td>fortran power</td><td></td><td></td><td></td><td>x</td></tr>
<tr><td>.and.</td><td></td><td></td><td></td><td></td><td>x</td></tr>
</table>

Operator	Description	C	C++	Pascal	Fortran
.or.					x
.negv.					x
.xor.					x

B.1 Precedence and Associativity

Precedence and associativity of operators are the same as the respective languages. Parentheses can be used for grouping.

If the program being debugged is not active and there is no `core` file, you may only use expressions containing constants. Procedure calls also require that the program be active.

The Debugger attempts to evaluate expressions the same way as the compiler.

Quick Start: Debugging Fortran 77 C

C.1 The Debugger (dbx)

This section introduces some dbx features likely to be used with FORTRAN. Use it as a quick start for debugging.

Note – Before you use the debugger, you must install the appropriate Tools package—read *Installing SunSoft Developer Products (SPARC/Solaris)* for details.

C.1.1 Sample Program for Debugging

Here is a program that includes the files, `a1.f`, `a2.f`, and `a3.f`, that contain bugs, and is used in several examples of debugging.

Example: Main for debugging:

a1.f

```
      PARAMETER ( n=2 )
      REAL twobytwo(2,2) / 4 *-1 /
      CALL mkidentity( twobytwo, n )
      PRINT *, determinant( twobytwo )
      END
```

Example: Subroutine for debugging:

a2.f

```
      SUBROUTINE mkidentity ( array, m )
      REAL array(m,m)
      DO 90 i = 1, m
          DO 20 j = 1, m
              IF ( i .EQ. j ) THEN
             array(i,j) = 1.
              ELSE
             array(i,j) = 0.
              END IF
20        CONTINUE
90    CONTINUE
      RETURN
      END
```

Example: Function for debugging:

a3.f

```
      REAL FUNCTION determinant ( a )
      REAL a(2,2)
      determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
      RETURN
      END
```

C.1.2 Sample `dbx` Session

The following examples use the sample program.

1. **Compile and link with the** `-g` **flag. You can do this in one or two steps.**

   Example: Compile and link *in one step*, with `-g`:

```
demo% f77 -o silly -g a1.f a2.f a3.f
```

   Example: Compile and link *in separate steps*:

```
demo% f77 -c -g a1.f a2.f a3.f
demo% f77 -g -o silly a1.o a2.o a3.o (Use -g in Solaris 1.x, but not in 2.x)
```

2. **To start** `dbx`**, type** `dbx` **and the name of your executable file.**
   The prompt becomes: `(dbx)`.

   Example: Start `dbx` on the executable named `silly`:

```
demo% dbx silly
Reading symbolic information…
(dbx)
```

3. **To quit** `dbx`**, enter the** `quit` **command.**

   Example: Quit `dbx`:

```
(dbx)quit                    (Skip this for now so you can do the next steps.)
demo%
```

4. **To set a breakpoint, wait for the** `dbx` **prompt, then type:** `stop in` *subnam***, where** *subnam* **names a subroutine, function, or block data subprogram.**

   Example: A way to stop at the first executable statement in a main program:

MAIN must be in uppercase.

```
(dbx) stop in MAIN
(2) stop in MAIN
(dbx)
```

   Although `MAIN` must be in uppercase, in general, *subnam* can be uppercase or lowercase.

Case-Sensitive Variable Recognition.

5. **To run the program from** `dbx`**, enter the** `run` **command, which runs the program in the executable files that were named when you started** `dbx`**.**

   Example: Run the program from within `dbx`:

```
(dbx) run
Running: silly
stopped in MAIN at line 3 in file "a1.f"
   3       call mkidentity( twobytwo, n )
(dbx)
```

   When the breakpoint is reached, `dbx` displays a message showing where it stopped, in this case, at line 3 of the `a1.f` file.

6. **To print a value, enter the** `print` **command.**

   Example: Print value of `n`:

```
(dbx) print n
n = 2
(dbx)
```

   Example: Print the matrix `twobytwo`; the format may vary with the release:

```
(dbx) print twobytwo
twobytwo =
   (1,1)       -1.0
   (2,1)       -1.0
   (1,2)       -1.0
   (2,2)       -1.0
(dbx)
```

Example: Print the matrix `array`:

The error message details may vary with the release, and, of course, with any translation.

```
(dbx) print array
dbx: "array" is not defined in the current scope
(dbx)
```

The print fails because `array` is not defined here—only in `mkidentity`.

7. To advance execution to the next line, enter the `next` command.

Example: Advance execution to the next line:

```
(dbx) next
stopped in MAIN at line 4 in file "a1.f"
    4      print *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo =
    (1,1)      1.0
    (2,1)      0.0
    (1,2)      0.0
    (2,2)      1.0
(dbx) quit
demo%
```

The `next` command executes the current source line, then stops at the next line. It counts subprogram calls as single statements.

Compare `next` with `step`. The `step` command executes the next source line, or the next step into a subprogram, and so forth. In general, if the next executable source statement is a subroutine or function call, then:

- `step` sets a breakpoint at the first source statement of the subprogram.
- `next` sets the breakpoint at the first source statement after the call, but still in the calling program.

C.2 Segmentation Fault

If a program gets a segmentation fault (`SIGSEGV`), it references a memory address outside of the memory available to it.

C.2.1 Some Causes

The most frequent causes for a segmentation fault are:

- An array index is outside the declared range.
- The name of an array index is misspelled.
- The calling routine has a `REAL` argument, which the called routine has as `INTEGER`.
- An array index is miscalculated.
- The calling routine calls has fewer arguments than required.
- A pointer is used before it is defined

C.2.2 Some Ways to Locate the Problems

To locate the offending source line, use `dbx` to find the source code line where a segmentation fault occurred.

Example: Use a program to generate a segmentation fault:

```
demo 4% cat WhereSEGV.f
       INTEGER a(5)
       j = 2000000
       DO 9 i = 1,5
          a(j) = (i * 10)
9      CONTINUE
       PRINT *, a
       END
demo 5%
```

Example: Use dbx to find the line number of a segmentation fault:

```
demo 5% f77 -g -silent WhereSEGV.f
demo 6% a.out
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation fault (core dumped)
demo 7% dbx a.out
Reading symbolic information for a.out
program terminated by signal SEGV (segmentation violation)
(dbx) run
Running: a.out
signal SEGV (no mapping at the fault address)
    in MAIN at line 4 in file "WhereSEGV.f"
    4                   a(j) = (i * 10)
(dbx)
```

C.3 Exceptions

If a program gets an exception, there are many possible causes. One approach to locate the problem is to find the line number in the source program where the exception occurred, then look for clues there.

Example: Find where an exception occurred:

`WhereExcept.f`

You can find the source code line number where a floating-point exception occurred by using the ieee_handler routine with either dbx or debugger.

The catch FPE dbx command

```
      EXTERNAL myhandler                              ! Main
      INTEGER ieeer, ieee_handler, myhandler
      REAL r/14.2/, s/0.0/
      ieeer = ieee_handler('set', 'all', myhandler)
      PRINT *, r/s
      END
      INTEGER FUNCTION myhandler(sig, code, context) ! Handler
*     {This handler is OK in SunOS 4.X/5.0 since it just aborts.}
      INTEGER sig, code, context(5)
      CALL abort()
      END
demo% f77 -g -silent WhereExcept.f
demo% dbx a.out
Reading symbolic information for a.out
(dbx) catch FPE
(dbx) run
Running: a.out
signal FPE (floating point divide by zero)
     in MAIN at line 5 in file "WhereExcept.f"
    5      PRINT *, r/s
(dbx)
```

C.4 Trace of Calls

Sometimes a program stops with a core dump, and you need to know the sequence of calls that brought it there. This sequence is called a *stack trace*.

Example: Show the sequence of calls, starting at where the execution stopped:

```
ShowTrace.f is a program
contrived to get a core
dump a few levels deep in
the call sequence—to show
a stack trace.

Note the reverse order:

    MAIN called calc
    calc called calcb.

Execution stopped, line
23  Æ
calcB called from calc,
line 9 Æ
calc called from MAIN,
line 3  Æ
```

```
demo% f77 -silent -g ShowTrace.f
demo% a.out
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quil 174% dbx a.out
Reading symbolic information for a.out
...
(dbx) run
Running: a.out
(process id 1089)
signal SEGV (no mapping at the fault address) in calcb at line 23
in file "ShowTrace.f"
   23                    v(j) = (i * 10)
(dbx) where -V
=>[1] calcb(v =  ARRAY , m = 2), line 23 in "ShowTrace.f"
  [2] calc(a =  ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f"
  [3] MAIN(), line 3 in "ShowTrace.f"
(dbx)
```

The `where` command shows where in the program flow execution stopped—how execution reached this point—that is, a *stack trace* of the called routines. This information can be helpful, since you no longer get an *automatic* traceback.

C.5 Traceback

A traceback shows the sequence of calls at the time of an exception.

By default, if your program generates an exception that would normally cause a core dump, you get a simple stack traceback in addition to the system provided core dump. Such exceptions include segmentation violation, bus errors, etc.

If you put `-notrace` on the f77 command line, you get no traceback.

C.5.1 Reasons to Use Traceback

- You need only the call sequence.
- The core file is so big that it is slow to load into the debugger.
- The core would be so big that you do not get a core file.
- Your system is configured to not get a core file.

C.5.2 Reasons to not Use Traceback

- In the current implementation, traceback runs in the same sick process it is examining.
- Traceback adds extra, confusing information to the core file.
- In certain rare circumstances, the traceback can corrupt your core file.
- Trying to create a traceback can result in no traceback and no core file.
- If a core file is created, a `where` command under `dbx` tells you as much information with less effort.

Note that traceback usually does not corrupt the core file; it usually works ok.

C.5.3 How Traceback Works

The traceback information is written to the *executable*`.trace` file. If your executable file is `a.out`, then your traceback file is `a.out.trace`. It is saved on a file because the traceback might be long and scroll off the screen. All of these `.traceback` files are cumulative, that is, they are *appended* to, not written from scratch.

C.5.4 Traceback Output

Details vary slightly, but each line of the output has a form something like this:

```
Called from [func: name] at [lineno], address, args=
```

The level of information the traceback reports depends on how the file was compiled. If the file was stripped, then it reports only hex values for function addresses, line numbers, and parameters. If the file is a normal executable, then it reports most function names; if the file was compiled with `-g`, then it reports line number information also.

It shows four arguments, no matter how many there are. It shows extra arguments as zeros.

Example: Run the executable file, in this case named `sigs`, which causes a segmentation fault. It should look something like this:

```
demo % sigs
*** Segmentation Violation = signal 11 code 3
demo % ■
```

Then display the traceback file. It should look something like this:

The traceback shows the function that catches the signal, __f77_sigdie.

Your function will be listed just below that.

```
demo % cat sigs.trace
Begin traceback…
Called from [func: __f77_sigdie], at 0x25c2, args=0x21ec8 0x2ab88
0x28c00 0xf
Called from [func: test42], at 0xecc17e4, args=0xb 0x3 0xefffb90
0x0
Called from [func: _MAIN_], at 0x22b4, args=0x0 0x0 0xefffbf4
0x2460
Called from [func: _main], at 0x2460, args=0x0 0x10 0x0 0x0
End traceback…
demo % ■
```

The actual addresses will vary depending on your installation and architecture.

C.6 Arrays

Example: dbx recognizes arrays and can print them:

Arraysdbx.f

```
demo% dbx a.out
Reading symbolic information…
(dbx) list 1,25
    1           DIMENSION IARR(4,4)
    2           DO 90 I = 1,4
    3                   DO 20 J = 1,4
    4                           IARR(I,J) = (I*10) + J
    5   20              CONTINUE
    6   90      CONTINUE
    7           END
(dbx) stop at 7
(1) stop at "Arraysdbx.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "Arraysdbx.f"
    7           END
(dbx) print IARR
iarr =
    (1,1) 11
    (2,1) 21
    (3,1) 31
    (4,1) 41
    (1,2) 12
    (2,2) 22
    (3,2) 32
    (4,2) 42
    (1,3) 13
    (2,3) 23
    (3,3) 33
    (4,3) 43
    (1,4) 14
    (2,4) 24
    (3,4) 34
    (4,4) 44
(dbx) print IARR(2,3)
    iarr(2, 3) = 23  ← Order of user-specified subscripts ok
(dbx) quit
demo%
```

C.7 Array Slices

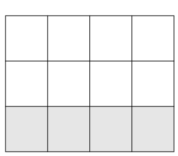
Example: dbx prints array *slices* if you specify which rows and columns:

ShoSli.f

This is one way of printing portions of large arrays.

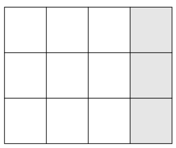
```
demo% f77 -g -silent ShoSli.f
demo% dbx a.out
Reading symbolic information for a.out
(dbx) list 1,12
    1       INTEGER*4  a(3,4), col, row
    2       DO row = 1,3
    3           DO col = 1,4
    4             a(row,col) = (row*10) + col
    5           END DO
    6       END DO
    7       DO row = 1, 3
    8            WRITE(*,'(4I3)') (a(row,col),col=1,4)
    9       END DO
    10      END
(dbx) stop at 7
(1) stop at "ShoSli.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "ShoSli.f"
    7        DO row = 1, 3
(dbx)
```

Example: Print row 3:



```
(dbx) print a(3:3,1:4)
'ShoSli'MAIN'a(3:3, 1:4) =
        (3,1)   31
        (3,2)   32
        (3,3)   33
        (3,4)   34
(dbx)
```

Example: Print column 4:



```
(dbx) print a(1:3,4:4)
'ShoSli'MAIN'a(3:3, 1:4) =
        (1,4)   14
        (2,4)   24
        (3,4)   34
(dbx)
```

C.8 Intrinsic Functions

dbx recognizes FORTRAN intrinsic functions.

Example: Show an intrinsic function in dbx:

```
demo% cat ShowIntrinsic.f
    INTEGER i
    i = 2
    END
demo% f77 -g -silent ShowIntrinsic.f
demo% dbx a.out
(dbx) stop in MAIN
(dbx) run
Running: a.out
(process id 10903)
stopped in MAIN at line 2 in file "ShowIntrinsic.f"
    2        i = 2
(dbx) whatis abs
Generic intrinsic function: "abs"
(dbx) print abs(i)
abs(i) = 0
(dbx) quit
demo%
```

C.9 Complex Expressions

dbx also recognizes FORTRAN complex expressions.

Example: Show a complex expression in dbx:

```
demo% cat ShowComplex.f
    COMPLEX z
    z = ( 2.0, 3.0 )
    END
demo% f77 -g -silent ShowComplex.f
demo% dbx a.out
(dbx) stop in MAIN
(dbx) run
Running: a.out
(process id 10953)
stopped in MAIN at line 2 in file "ShowComplex.f"
    2       z = ( 2.0, 3.0 )
(dbx) whatis z
complex*8  z
(dbx) print z
z = (0.0,0.0)
(dbx) next
stopped in MAIN at line 3 in file "ShowComplex.f"
    3       END
(dbx) print z
z = (2.0,3.0)
(dbx) print z+(1.0,1.0)
z+(1,1) = (3.0,4.0)
(dbx) quit
demo%
```

C.10 Logical Operators

dbx can locate FORTRAN logical operators and print them.

Example: Show logical operators in dbx:

```
demo% cat ShowLogical.f
        LOGICAL a, b, y, z
        a = .true.
        b = .false.
        y = .true.
        z = .false.
        END
demo% f77 -g -silent ShowLogical.f
demo% dbx a.out
(dbx) list 1,9
    1           LOGICAL a, b, y, z
    2           a = .true.
    3           b = .false.
    4           y = .true.
    5           z = .false.
    6           END
(dbx) stop at 5
(2) stop at "ShowLogical.f":5
(dbx) run
Running: a.out
(process id 15394)
stopped in MAIN at line 5 in file "ShowLogical.f"
    5           z = .false.
(dbx) whatis y
logical*4  y
(dbx) print a .or. y
a.OR.y = true
(dbx) assign z = a .or. y
(dbx) print z
z = true
(dbx) quit
demo%
```

C.11 Miscellaneous Tips

The following tips and general concepts might help while debugging Fortran77 programs.

C.11.1 Current Procedure and File

During a debug session, `dbx` defines a procedure and a source file as current. Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, `stop at 5` sets one of three different breakpoints, depending on whether the current file is `a1.f`, `a2.f`, or `a3.f`.

C.11.2 Uppercase Letters

In general, if your program has uppercase letters in any identifiers, then the debugger recognizes them. You need not give it any specific case-sensitive or case-insensitive commands, as in some earlier versions.

`f77` and `dbx` must be in the same case-sensitive or case-insensitive mode:

- To compile and debug in case-insensitive mode, do so without the `-U` option. The debugger default then is: `dbxenv case insensitive`.

  If the source has a variable named `LAST`, then in dbx, both the `print LAST` or `print last` commands work. Both `f77` and `dbx` consider `LAST` and `last` to be the same, as requested.

- To compile and debug in case-sensitive mode, use `-U`. The debugger default is then `dbxenv case sensitive`.

  If the source has a variable named `LAST`, but one named `last`, then in dbx, `print LAST` works, but `print last` does *not* work. Both `f77` and `dbx` distinguish between `LAST` and `last`, as requested.

Note – File or directory names are always case-sensitive in both `debugger` and `dbx`. This rule is true even if you have set the `dbxenv case insensitive` environment attribute.

C.11.3 Optimized Programs

To debug optimized programs:

- Compile the main program with `-g` but with no `-O`*n*.
- Compile every other routine of the program with the appropriate `-O`*n*.
- Start the execution under `dbx`.
- Use `fix -g` *any*`.f` on the routine you want to debug, *but no* `-O`*n*.
- Use `continue` with that routine compiled.

C.12 Help

At the debugger prompt, to get:

- *All commands*—a list of commands, grouped by action, type `help`.
- *Details of one command*—a command explanation, type `help` *cmdname*.
- *Changes*—a list of the new and changed features, type `help changes`
- *FAQ*—answers to frequently asked questions, type `help FAQ`

Example: Command Summary (output varies with release).

```
(dbx) help
                                   Command Summary
Execution and Tracing
  cancel      catch       clear       cont        delete      fix
  fixed       handler     ignore      intercept   next        pop
  replay      rerun       restore     run         save        status
  step        stop        trace       unintercept when        whocatches
Displaying and Naming Data
  assign    call      demangle  dis       display   down      dump
  examine   exists    frame     hide      inspect   print     undisplay
  unhide    up        whatis    where     whereami  whereis   which
<many commands omitted>
```

Example: “help *cmdnam*”—details for the where command.

```
(dbx) help where
where                      # Print a procedure traceback
where <num>                # Print the <num> top frames in the traceback
where -f <num>             # Start traceback from frame <num>
where -h                   # Include hidden frames
where -q                   # Quick traceback (only function names)
where -v                  # Verbose traceback (include function args and line
info)

Any of the above forms may be followed by a thread or LWP ID to obtain the
traceback for the specified entity.
(dbx) ■
```

Quick Start: Debugging Fortran 90 D

This appendix is organized as follows:

This appendix *introduces* some `dbx` features likely to be used with Fortran 90. Use it as a *quick start* for debugging Fortran 90. Be sure to try the `help` feature.

Note – Before you use the debugger, you must install the appropriate Tools package—read *Installing SunPro Software on Solaris* for details.

With `dbx` you can display and modify variables, set breakpoints, trace calls, and invoke procedures in the program being debugged without having to recompile.

The `debugger` program lets you make more effective use of `dbx` by replacing the original, terminal-oriented interface with a window- and mouse-based interface.

D.1 Sample Program for Debugging

The following program, with bug, and consisting of files `a1.f90`, `a2.f90`, and `a3.f90`, is used in several examples of debugging.

Example: Main for debugging.

a1.f90

```
PROGRAM TryDbx
   INTEGER, PARAMETER :: n=2
   REAL, DIMENSION(n,n) :: twobytwo
   DATA (( twobytwo(k,j),k=1,n),j=1,n) / 4*-1 /
   CALL mkidentity( twobytwo, n )
   PRINT *, determinant( twobytwo )
END
```

Example: Function for debugging.

a3.f90

```
REAL FUNCTION determinant ( a )
   REAL a(2,2)
   determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
END
```

Example: Subroutine for debugging.

a2.f90

```
SUBROUTINE mkidentity ( array, m )
   REAL, DIMENSION(m,m) :: array
   DO i = 1, m
    DO j = 1, m
        IF ( i .eq. j ) THEN
        array(i,j) = 1.0
        ELSE
        array(i,j) = 0.0
        END IF
    END DO
   END DO
END
```

D.2 A Sample dbx Session

The following examples use the sample program above.

- *Compile*—To use dbx or debugger, compile and link with the -g flag. You can do this in one step or two, as shown in the examples below.

  Example Compile and link *in one step*, with -g.

```
demo$ f90 -o silly -g a1.f90 a2.f90 a3.f90
```

  • Example: Compile and link *in separate steps*.

```
demo$ f90 -c -g a1.f90 a2.f90 a3.f90
demo$ f90 -o silly a1.o a2.o a3.o
```

- *Start* dbx—To start dbx, type dbx and the name of your executable file.

  Example: Start dbx on the executable named silly.

```
demo$ dbx silly
Reading symbolic information…
(dbx) ■
```

- *Quit* dbx—To quit dbx, enter the quit command.

  Example: Quit dbx.

```
(dbx)quit  {Skip this for now so you can do the next steps.}
demo$ ■
```

- *Breakpoint*—To set a breakpoint, at the dbx prompt; type "stop in *subnam*", where *subnam* names a subroutine or function, and *subnam* can be upper case or lower case.

  Example: A way to stop at the first executable statement in a main program.

```
(dbx) stop in main
(2) stop in main
(dbx) ■
```

- *Run Program*—To run a program from within dbx, enter the run command.

  Example: Run a program from within dbx.

```
 (dbx) run
 Running: silly
(process id 8786)
 stopped in main at line 5 in file "a1.f90"
    5    CALL mkidentity( twobytwo, n )
 (dbx) ■
```

  When the breakpoint is reached, dbx displays a message showing where it stopped, in this case at line 3 of the a1.f90 file.

- *Print*—To print a value, enter the print command.

  Example: Print the variable n. Note that dbx does parameters.

```
(dbx) print n
n = 2
(dbx) ■
```

  Example: Print the matrix twobytwo (format may vary with release).

```
(dbx) print twobytwo
twobytwo =
   (1,1) -1.0
   (2,1) -1.0
   (1,2) -1.0
   (2,2) -1.0
(dbx) ■
```

Example: Print the matrix `array`.

```
(dbx) print array
dbx: "array" is not defined in the current scope
dbx: see 'help scope' for details
(dbx) ■
```

In the above example:

- The print fails because `array` is not defined here—only in `mkidentity`.
- The error message details may vary with the release, and translation.

- *Next Line*—To advance execution to the next line, enter the `next` command.

Example: Advance execution to the next line.

```
(dbx) next
stopped in main at line 6 in file "a1.f90"
    6    PRINT *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo =
    (1,1) 1.0
    (2,1) 0.0
    (1,2) 0.0
    (2,2) 1.0
(dbx) quit
demo$ ■
```

Note that "`print twobytwo`" now displays the unit matrix.

The `next` command executes the current source line, then stops at the next line. It counts subprogram calls as single statements.

Compare `next` with `step`. The `step` command executes the next source line, or the next step into a subprogram, and so forth. In general, if the next executable source statement is a subroutine or function call, then

- `step` sets a breakpoint at the first source statement of the subprogram.
- `next` sets the breakpoint at the first source statement after the call but still in the calling program.

D.3 Segmentation Fault—Finding the Line Number

If a program gets a *segmentation fault* (SIGSEGV), it referenced a memory address outside of the memory available to it.

D.3.1 Some Causes of SIGSEGV

The most frequent causes are the following:

- An array index being outside the declared range
- The name of an array index is misspelled
- The calling routine has a REAL argument; called routine has it as INTEGER
- An array index is miscalculated
- The calling routine calls with fewer arguments than required
- A pointer is used before it is defined

You can locate the offending source line using -Xlist or dbx.

- Recompile with the -Xlist option to get global program checking
- Use dbx to find the source code line where a segmentation fault occurred

Example: Program to generate a segmentation fault.

```
demo 4% cat WhereSEGV.f90
    INTEGER a(5)
    j = 2000000
    DO i = 1,5
        a(j) = (i * 10)
    END DO
    PRINT *, a
    END
demo 5% ■
```

Example: Use dbx to locate a segmentation fault.

```
demo 5% f90 -g WhereSEGV.f90
demo 6% a.out
Segmentation fault (core dumped)
demo 7% dbx a.out
Reading symbolic information for a.out
    ... other messages ...
(dbx)run
Running: a.out
(process id 8813)
signal SEGV (no mapping at the fault address) in main at line 4
        in file "WhereSEGV.f90"
    4                    a(j) = (i * 10)
(dbx) quit
demo 8% ■
```

D.4 Exception—Finding the Line Number

Example: Find where an exception occurred.

WhereExcept.f90

You can find the source code line number where a floating-point exception occurred by using the ieee_handler routine with either dbx or debugger.

Note the “catch FPE” dbx command

Æ

```
    EXTERNAL myhandler                             !  Main
    INTEGER ieeer, ieee_handler, myhandler
    REAL :: r=14.2, s=0.0
    ieeer = ieee_handler('set', 'all', myhandler)
    PRINT *, r/s
    END
    INTEGER FUNCTION myhandler(sig, code, context) ! Handler
!   {This handler is OK in SunOS 4.X/5.0 since it just aborts.}
    INTEGER sig, code, context(5)
    CALL abort()
    END
demo$ f90 -g WhereExcept.f90
demo$ dbx a.out
Reading symbolic information for a.out
 ...
(dbx) catch FPE
(dbx) run
Running: a.out
signal FPE (floating point divide by zero)
     in main at line 5 in file "WhereExcept.f"
    5      PRINT *, r/s
(dbx) ■
```

D.5 Trace of Calls

Sometimes a program stops with a core dump, and you need to know the sequence of calls that brought it there (a *stack trace*).

Example: Show the sequence of calls, starting at where the execution stopped.

ShowTrace.f90 is a program contrived to get a core dump a few levels deep in the call sequence—to show a stack trace.

Execution stopped, line 23 Æ

calcB called from calc, line 9 Æ

calc called from main, line 3 Æ

Note reverse order: main called calc, calc called calcb.

```
demo$ f90 -g ShowTrace.f90
demo$ a.out
Segmentation Fault (core dumped)
quil 174% dbx a.out
Reading symbolic information for a.out
...
(dbx) run
(process id 8939)
Running: a.out
(process id 1089)
signal SEGV (no mapping at the fault address) in calcb
           at line 23 in file "ShowTrace.f"
   23                  v(j) = (i * 10)
(dbx) where
=>[1] calcb(v =  ARRAY , m = 2), line 23 in "ShowTrace.f90"
  [2] calc(a =  ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f90"
  [3] main(), line 3 in "ShowTrace.f90"
(dbx) ■
```

The `where` command shows where in the program flow execution stopped (how execution reached this point), that is, a *stack trace* of the called routines. This can be helpful, since you no longer get an *automatic* traceback.

D.6 Pointer to a Scalar

Example: Pointer to a scalar, in dbx.

PtrScal.f90

```
demo% f90 -g PtrScal.f90
demo% dbx a.out
(dbx) list 1,99
    1   PROGRAM PtrScalar
    2     REAL, POINTER :: p
    3     REAL, TARGET :: r
    4     p => r
    5     r = 2.3
    6     PRINT *, p
    7     p = 3.2
    8     PRINT *, r
    9   END
(dbx) stop at 8
(2) stop at "PtrScal.f90":8
(dbx) run
Running: a.out
(process id 12367)
 2.29999995
stopped in main at line 8 in file "PtrScal.f90"
    8     PRINT *, r
(dbx) whatis p
real*4  p  ! f90 pointer
(dbx) whatis r
real*4  r
(dbx) print p
p = 3.2
(dbx) print r
r = 3.2
(dbx) quit
demo$
```

D.7 Pointer to an Array

Example: Pointer to an array, in dbx.

PtrArray.f90

```
demo% f90 -g PtrArray.f90
demo% dbx a.out
(dbx) list 1,99
    1   PROGRAM PtrArray
    2           INTEGER, TARGET :: a(5,5)
    3           INTEGER, POINTER :: corners(:,:)
    4           DO i = 1,5
    5              a(i,:) = i
    6           END DO
    7           corners => a(1:5:4, 1:5:4)
    8           PRINT *, corners
    9   END
(dbx) stop at 8
(2) stop at "PtrArray.f90":8
(dbx) run
Running: a.out
(process id 12397)
stopped in main at line 8 in file "PtrArray.f90"
    8           PRINT *, corners
(dbx) whatis a
integer*4  a(1:5,1:5)
(dbx) whatis corners
integer*4 , corners(1:2,1:2) ! f90 pointer
(dbx) print corners
corners =
    (1,1)       1
    (2,1)       5
    (1,2)       1
    (2,2)       5
(dbx) quit
demo$
```

D.8 User Defined Types

Example: Structures—user defined types, in dbx.

DebStruc.f90

```
demo% f90 -g DebStruc.f90
demo% dbx debstr
(dbx) list 1,99
    1   PROGRAM Struct  ! Debug a Structure
    2      TYPE product
    3         INTEGER        id
    4         CHARACTER*16   name
    5         CHARACTER*8    model
    6         REAL           cost
    7         REAL           price
    8      END TYPE product
    9
   10      TYPE(product) :: prod1
   11
   12      prod1%id = 82
   13      prod1%name = "Schlepper"
   14      prod1%model = "XL"
   15      prod1%cost = 24.0
   16      prod1%price = 104.0
   17      WRITE ( *, * ) prod1%name
   18   END
(dbx) stop at 17
(2) stop at "Struct.f90":17
(dbx) run
Running: a.out
(process id 12326)
stopped in main at line 17 in file "Struct.f90"
   17      WRITE ( *, * ) prod1%name
(dbx)  whatis prod1
product  prod1
(dbx) whatis -t product
type product
    integer*4 id
    character*16 name
    character*8 model
    real*4 cost
    real*4 price
end type product
(dbx)
```

Example: Structures—user defined types, in dbx.

```
(dbx) print prod1
prod1 = (
    id    = 82
    name  = 'Schlepper'
    model = 'XL'
    cost  = 24.0
    price = 104.0
)
(dbx) quit
(dbx) 
```

D.9 Pointer to User Defined Type

Example: Structures—user defined types, and pointers, in dbx.

`DebStruc.f90`

`Declare a user-defined type.`

`Declare variables` *prod1* `and` *prod2* `to be of that type and targets.`

`Declare variables` *curr* `and` *prior* `as pointers to the type.`

`Make` *curr* `point to` *prod1*.
`Make` *prior* `point to` *prod1*.

`Initialize` *prior*.

`Set` *curr* `to` *prior*.
`Print` *name* `from` *curr* `and` *prior*.

```
demo% f90 -o debstr -g DebStruc.f90
demo% dbx debstr
(dbx) stop in main
(2) stop in main
(dbx) list 1,99
     1   PROGRAM DebStruPtr ! Debug structures & pointers
     2      TYPE product
     3         INTEGER        id
     4         CHARACTER*16   name
     5         CHARACTER*8    model
     6         REAL           cost
     7         REAL           price
     8      END TYPE product
     9
    10      TYPE(product), TARGET :: prod1, prod2
    11      TYPE(product), POINTER :: curr, prior
    12
    13      curr => prod2
    14      prior => prod1
    15      prior%id = 82
    16      prior%name = "Schlepper"
    17      prior%model = "XL"
    18      prior%cost = 24.0
    19      prior%price = 104.0
    20      curr = prior
    21      WRITE ( *, * ) curr%name, " ", prior%name
    22   END PROGRAM DebStruPtr
(dbx) █
```

Example: Structures—set a breakpoint, and run under dbx.

The exact layout and messages may vary with each release.

```
(dbx) stop at 21
(1) stop at "DebStruc.f90":21
(dbx) run
Running: debstr
(process id 10972)
stopped in main at line 21 in file "DebStruc.f90"
   21      WRITE ( *, * ) curr%name, " ", prior%name
(dbx) ■
```

Example: Structures—print an item of user-defined type.

```
(dbx) print prod1
 prod1 = (
    id = 82
    name = "Schlepper        "
    model = "XL      "
    cost = 24.0
    price = 104.0
)
(dbx) ■
```

Above, dbx displays all fields of the user-defined type, including field names.

Example: Structures—inquire about an item of user-defined type.

Ask about the variable

Ask about the type (-t)

```
(dbx) whatis prod1
product prod1
(dbx) whatis -t product
type product
   integer*4 id
   character*16 name
   character*8 model
   real cost
   real price
end type product
(dbx) ■
```

Example: Structures—print a pointer, then quit `dbx`.

dbx displays the contents of a pointer, which is an address. This address can be different with every run.

```
(dbx) print prior
prior = (
    id    = 82
    name  = 'Schlepper'
    model = 'XL'
    cost  = 24.0
    price = 104.0
)
(dbx) quit
demo$ ■
```

D.10 Allocated Arrays

Example: Allocated arrays in `dbx`.

The exact layout and messages may vary with each release.

Alloc.f90

Unknown size is at line 6 Æ

Known size is at line 9 →

buffer(1000) holds1000 →

```
demo% f90 -g Alloc.f90
demo% dbx a.out
(dbx) list 1,99
    1   PROGRAM TestAllocate
    2   INTEGER n, status
    3   INTEGER, ALLOCATABLE :: buffer(:)
    4           PRINT *, 'Size?'
    5           READ *, n
    6           ALLOCATE( buffer(n), STAT=status )
    7           IF ( status /= 0 ) STOP 'cannot allocate buffer'
    8           buffer(n) = n
    9           PRINT *, buffer(n)
   10           DEALLOCATE( buffer, STAT=status)
   11   END
(dbx) stop at 6
(2) stop at "alloc.f90":6
(dbx) stop at 9
(3) stop at "alloc.f90":9
(dbx) run
Running: a.out
(process id 10749)
 Size?
1000
stopped in main at line 6 in file "alloc.f90"
    6           ALLOCATE( buffer(n), STAT=status )
(dbx) whatis buffer
integer*4 , allocatable::buffer(:)
(dbx) next
continuing
stopped in main at line 7 in file "alloc.f90"
    7           IF ( status /= 0 ) STOP 'cannot allocate buffer'
(dbx) whatis buffer
integer*4  buffer(1:1000)
(dbx) cont
stopped in main at line 9 in file "alloc.f90"
    9           PRINT *, buffer(n)
(dbx) print n
n = 1000
(dbx) print buffer(n)
buffer(n) = 1000
(dbx) ■
```

D.11 Print Arrays

Example: dbx recognizes arrays. It can print arrays.

Arraysdbx.f90

```
demo$ dbx a.out
(dbx) list 1,25
    1           DIMENSION iarr(4,4)
    2           DO i = 1,4
    3                   DO j = 1,4
    4                           iarr(i,j) = (i*10) + j
    5                   END DO
    6           END DO
    7           END
(dbx) stop at 7
(1) stop at "Arraysdbx.f90":7
(dbx) run
Running: a.out
stopped in main at line 7 in file "Arraysdbx.f90"
    7           END
(dbx) print IARR
iarr =
    (1,1) 11
    (2,1) 21
    (3,1) 31
    (4,1) 41
    (1,2) 12
    (2,2) 22
    (3,2) 32
    (4,2) 42
    (1,3) 13
    (2,3) 23
    (3,3) 33
    (4,3) 43
    (1,4) 14
    (2,4) 24
    (3,4) 34
    (4,4) 44
(dbx) print IARR(2,3)
    iarr(2, 3) = 23  ← order of user-specified subscripts ok
(dbx) quit
demo$ ■
```

D.12 Print Array Slices

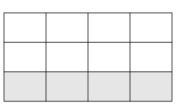
This section shows one way of printing portions of large arrays.

Example: dbx prints array *slices* if you specify which rows and columns.

ShoSli.f90

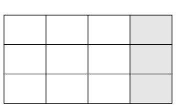
```
demo$ f90 -g ShoSli.f90
demo$ dbx a.out
(dbx) list 1,12
    1      INTEGER*4  a(3,4), col, row
    2      DO row = 1,3
    3          DO col = 1,4
    4            a(row,col) = (row*10) + col
    5          END DO
    6      END DO
    7      DO row = 1, 3
    8           write(*,'(4I3)') (A(row,col),col=1,4)
    9      END DO
    10     END
(dbx) stop at 7
(1) stop at "ShoSli.f90":7
(dbx) run
Running: a.out
stopped in main at line 7 in file "ShoSli.f90"
    7       DO row = 1, 3
(dbx)■
```

Example: Print row 3.



```
(dbx) print a(3:3,1:4)
a(3:3, 1:4) =
        (3,1)   31
        (3,2)   32
        (3,3)   33
        (3,4)   34
(dbx)■
```

Example: Print column 4.



```
(dbx) print a(1:3,4:4)
a(3:3, 1:4) =
        (1,4)   14
        (2,4)   24
        (3,4)   34
(dbx)■
```

D.13 Generic Functions

Example: Generic function, cube root.

Generic.f90

```
(dbx) list 1,99
MODULE cr
  INTERFACE cube_root
    FUNCTION s_cube_root(x)
      REAL :: s_cube_root
      REAL, INTENT(IN) :: x
    END FUNCTION s_cube_root
    FUNCTION d_cube_root(x)
      DOUBLE PRECISION :: d_cube_root
      DOUBLE PRECISION, INTENT(IN) :: x
    END FUNCTION d_cube_root
  END INTERFACE
END MODULE cr
FUNCTION s_cube_root(x)
    REAL :: s_cube_root
    REAL, INTENT(IN) :: x
    s_cube_root = x ** (1.0/3.0)
END FUNCTION s_cube_root
FUNCTION d_cube_root(x)
    DOUBLE PRECISION :: d_cube_root
    DOUBLE PRECISION, INTENT(IN) :: x
    d_cube_root = x ** (1.0d0/3.0d0)
END FUNCTION d_cube_root
USE cr
  REAL :: x, cx
  DOUBLE PRECISION :: y, cy
  WRITE(*,"('Enter a SP number: ')")
  READ (*,*) x
  cx = cube_root(x)
  y = x
  cy = cube_root(y)
  WRITE(*,'("Single: ",F10.4, ", ", F10.4)') x, cx
  WRITE(*,'("Double: ",F12.6, ", ", F12.6)') y, cy
  WRITE(*,"('Enter a DP number: ')")
  READ (*,*) y
  cy = cube_root(y)
  x = y
  cx = cube_root(x)
  WRITE(*,'("Single: ",F10.4, ", ", F10.4)') x, cx
  WRITE(*,'("Double: ",F12.6, ", ", F12.6)') y, cy
  END
```

Example: `dbx` with a generic function, cube root.

If asked “What is cube_root?”, dbx tells you there are two, and asks you to select one.

If asked for cube_root(8) dbx tells you there are two, and asks you to select one.

If told to stop in cube_root, dbx tells you there are two, and asks you to select one.

From inside s_cube_root, show current value of x.

```
(dbx) stop at 26
(2) stop at "Generic.f90":26
(dbx) run
Running: Generic
(process id 14633)
stopped in main at line 26 in file "Generic.f90"
   26     WRITE(*,"('Enter a SP number : ')")
(dbx) whatis cube_root
More than one identifier 'cube_root.'
Select one of the following names:
 1) `Generic.f90`cube_root s_cube_root ! real*4 s_cube_root
 2) `Generic.f90`cube_root s_cube_root ! real*8 d_cube_root
> 1
 real*4  function  cube_root (x)
 (dummy argument) real*4  x
(dbx) print cube_root(8.0)
More than one identifier 'cube_root.'
Select one of the following names:
 1) `Generic.f90`cube_root ! real*4 s_cube_root
 2) `Generic.f90`cube_root ! real*8 d_cube_root
> 1
cube_root(8) = 2.0
(dbx) stop in cube_root
More than one identifier 'cube_root.'
Select one of the following names:
 1) `Generic.f90`cube_root ! real*4 s_cube_root
 2) `Generic.f90`cube_root ! real*8 d_cube_root
> 1
(3) stop in cube_root
(dbx) cont
continuing
Enter a SP number:
8
stopped in cube_root at line 16 in file "Generic.f90"
   16       s_cube_root = x ** (1.0/3.0)
(dbx) print x
x = 8.0
(dbx)■
```

D.14 Miscellaneous Tips

The following tips and background concepts can help.

D.14.1 Current Procedure and File

During a debug session, the debugger defines a procedure and a source file as *current*. Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, "`stop at 5`" sets one of three different breakpoints, depending on whether the current file is `a1.f90`, `a2.f90`, or `a3.f90`.

D.14.2 Uppercase Letters

In general, if your program has uppercase letters in any identifiers, then the debugger recognizes them. You do not need to give it any specific *case sensitive/insensitive* commands, as in some earlier versions.

In fact, for `f90` 1.0, f90 and `dbx` must both be in the case insensitive mode; that is, do *not* set "`dbxenv case sensitive`".

Note – Names of *files* or *directories* are always case sensitive in both `debugger` and `dbx`. This is true even if you have set the "`dbxenv case insensitive`" environment attribute.

D.15 Main Features of the Debugger

Be sure to read *Debugging a Program* for the following:

- The full range of features in the debugger
- The window- and mouse-based interface

D.15.1 Overview of `dbx` Features Useful for FORTRAN

The debugger provides event management, process control, and data inspection. It allows you to watch what is happening during program execution.

With dbx, you can do such things as the following:

- *Set watchpoints* to stop or trace if a specified item changes
- Solaris 2.1 *Collect data* for the performance-tuning Analyzer
- *Graphically monitor* variables, structures, and arrays—Data Inspector
- *Set breakpoints* (set places to halt in the program) at lines or in functions
- *Show values*—once halted, show or modify variables, arrays, structures, …
- *Step* through program, one source line at a time (or one assembly line)
- *Trace* program flow (show sequence of calls taken)
- *Invoke procedures* in the program being debugged
- *Step* over or into function calls; step up and out of a function call
- *Run, stop, and continue* execution (at the next line or at some other line)
- *dbx-safe I/O* in the command window—Program Input/Output Window
- *Save and then replay* all or part of a debugging run
- *Stack*—Examine the call stack; move up and down the call stack
- *Program* scripts by embedded Korn shell
- Solaris 2.1 *Follow programs* as they fork(2) and exec(2)

D.16 Help

At the debugger prompt, to get:

- *All commands*—a list of commands, grouped by action, type help.
- *Details of one command*—a command explanation, type help *cmdname*.
- *Changes*—a list of the new and changed features, type help changes
- *FAQ*—answers to frequently asked questions, type help FAQ

Example: Command Summary (output varies with release).

```
(dbx) help
                                     Command Summary
Execution and Tracing
  cancel      catch       clear       cont        delete      fix
  fixed       handler     ignore      intercept   next        pop
  replay      rerun       restore     run         save        status
  step        stop        trace       unintercept when        whocatches
Displaying and Naming Data
  assign    call      demangle  dis       display   down      dump
  examine   exists    frame     hide      inspect   print     undisplay
  unhide    up        whatis    where     whereami  whereis   which
<many commands omitted>
```

Example: "`help` *cmdnam*"—details for the `where` command.

```
(dbx) help where
where                      # Print a procedure traceback
where <num>                # Print the <num> top frames in the traceback
where -f <num>             # Start traceback from frame <num>
where -h                   # Include hidden frames
where -q                   # Quick traceback (only function names)
where -v                  # Verbose traceback (include function args and line
info)

Any of the above forms may be followed by a thread or LWP ID to obtain the
traceback for the specified entity.
(dbx) ■
```

Index

E

F

G

H

I

K

L

Q

R

S

T

U

V

W