HP/DDE Debugger User's Guide

HP 9000 Series 700/800 Computers

HP Part No. B3476-90015
Printed in USA July 1996

First Edition
E0796

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Copyright © 1983-96 Hewlett-Packard Company

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. 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 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY
3000 Hanover Street
Palo Alto, California 94304 U.S.A.

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

©Copyright 1979, 1980, 1983, 1985-93 Regents of the University of California

This software is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California.

©Copyright 1980, 1984, 1986 Novell, Inc.
©Copyright 1986-1992 Sun Microsystems, Inc.
©Copyright 1985-86, 1988 Massachusetts Institute of Technology.
©Copyright 1989-93 The Open Software Foundation, Inc.
©Copyright 1986 Digital Equipment Corporation.
©Copyright 1990 Motorola, Inc.
©Copyright 1990, 1991, 1992 Cornell University
©Copyright 1989-1991 The University of Maryland
©Copyright 1988 Carnegie Mellon University

Trademarks. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

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

OSF/Motif is a trademark of the Open Software Foundation, Inc. in the U.S. and other countries.

"Sun" and the Sun logo are trademarks of Sun Microsystems, Inc. SunOS, Solaris, SPARC, and SPARCstation are trademarks of Sun Microsystems, Inc. NFS is a trademark of Sun Microsystems, Inc. Copyright © 1986, 1987, 1988 Sun Microsystems, Inc.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. We may issue a technical addendum or release notes as supplements.

The software version number printed alongside the date indicates the version of HP/DDE at the time we issued the manual.

B3476-90015	July 1996	Version 4.0
B3476-90011	June 1995	Version 3.2
B3476-90004	April 1994	Version 3.0

Preface

The *HP/DDE Debugger User's Guide* describes the HP Distributed Debugging Environment (HP/DDE), the high-level language debugger for the HP-UX operating system.

HP/DDE is also the default SoftBench Program Debugger which runs on both HP-UX and Solaris systems.

The debugger software consists of the main debugger and a set of managers. The main debugger provides such basic debugger functions as program control, process control, program and data monitoring, program information, and expression evaluation. The managers enable the debugger to handle different source languages, target machines, object file formats, and user interfaces.

This manual contains information on how to perform debugging tasks using HP/DDE's OSF/Motif user interface. It also contains information on the line-mode interface and the debugger managers in appendixes.

Note HP/DDE is also the default SoftBench Program Debugger. The SoftBench version of HP/DDE has a graphical user interface that differs somewhat from the interface presented in this manual. The debugging commands are the same.

The SoftBench Program Debugger runs on both HP-UX and Solaris systems.

Reference information on debugger commands and information on performing debugging tasks are available online through the HP Help System. This help is available from the Help Manager on the HP VUE front panel. It is also available from within the debugger if you use the `Help` menu or issue the `help` command.

Audience

This manual is written for programmers in C, C++, FORTRAN, or Pascal.

Summary of Technical Changes

This edition of the *HP/DDE Debugger User's Guide* documents the HP Distributed Debugging Environment (HP/DDE) Version 4.0 on HP-UX workstations.

HP/DDE Version 4.0 provides the following new features and enhancements:

- A new and easier to use graphical user interface.
- Support of PA-RISC 2.0 assembly language.
- Enhanced debugging of optimized code in C.
- Support for ANSI C++, including:
  - Allowing object specific breakpoints.
  - Automatic detection of most overloaded operators.
  - Allowing watchpoints on reference-type variables.
  - Support for `long long` types.
  - Support for `dynamic_cast<`*type*`>(`*expr*`)` operator.
- Support for FORTRAN 90, including:
  - Printing FORTRAN 90 values.
  - Kind Suffixes.
  - Specifying ranges in arrays.
- Improvements to the `call` command, including:
  - Support of string literal and union arguments, which allows calls to `printf`.
  - Support of calls to functions in images outside of the current location.
  - Support of calls to shared library functions with return values.
- Performance improvement for very large applications.

Related Documentation

For more information on HP-UX programming, refer to the following documents:

- *Programming on HP-UX* (B2355-90653) provides an overview of programming on HP-UX. It includes information about linking programs, creating and managing user libraries, optimizing programs, and porting programs.
- *HP/PAK Performance Analysis Tools User's Guide* (B3476-90016) describes the performance tools provided by the HP Program Analysis Kit (HP/PAK).
- The *HP FORTRAN/9000 Programmer's Reference* (B3906-90002) and *HP FORTRAN/9000 Programmer's Guide* (B3906-90001) describe the FORTRAN programming language on HP-UX systems.
- The *HP C/HP-UX Reference Manual* (92453-90024) and *HP C Programmer's Guide* (92434-90002) describe the C programming language on HP-UX systems.
- The *HP C++ Programmer's Guide* (92501-90026) and the *HP C++ Quick Reference Card* (B1637-90001) describe the C++ programming language on HP-UX systems.
- The *HP Pascal/HP-UX Reference Manual* (92431-90005) and *HP Pascal/HP-UX Programmer's Guide* (92431-90006) describe the HP Pascal programming language on HP-UX systems.
- The *Assembly Language Reference Manual* (92432-90001) describes assembly language programming on HP-UX Series 700/800 systems. The *ADB Tutorial* (92432-90005) introduces the assembly language debugger.
- The *Precision Architecture and Instruction Set Reference Manual* (09740-90014) and *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* (09740-90039) describe, respectively, the PA-RISC 1.0 and PA-RISC 1.1 architectures. *PA-RISC 2.0 Architecture*, by Gerry Kane (Prentice-Hall, ISBN 0-13-182734-0), describes the PA-RISC 2.0 architecture.

- The *Procedure Calling Conventions Reference Manual* (09740-90015) describes procedure calling conventions on PA-RISC systems.
- The *HP-UX Symbolic Debugger User's Guide* (B2355-90044) describes the `xdb` debugger.
- *The X Window System User's Guide*, O'Reilly & Associates, Inc., provides information about the X Window System.

To order manuals, call HP DIRECT at 1-800-637-7740. Outside the USA, please contact your local sales office.

Typographical Conventions

`computer font`	Computer font indicates commands, keywords, options, literals, source code, system output, and path names. In syntax formats, computer font indicates commands, keywords, and punctuation that you must enter exactly as shown.
underlined text	In interactive examples, underlined text represents user input.
italic type	In syntax formats, words or characters in italics represent values that you must supply. Italics are also used for book titles and for emphasis.
boldface type	Boldface words in glossary definitions indicate terms that are also defined in the glossary.
[ ]	In syntax formats, square brackets enclose optional items.
{ }	In syntax formats, braces enclose a list from which you must choose an item.
|	In syntax formats, a vertical bar separates items in a list of choices.
...	In syntax formats, a horizontal ellipsis indicates that you can repeat the preceding item one or more times.
⋮	A vertical ellipsis means that irrelevant parts of a figure or example have been omitted.
(key)	Type the corresponding key on the keyboard.
name(N)	An italicized word followed by a number in parentheses indicates a page and section number in the *HP-UX Reference*. For example, *cc*(1) refers to the *cc* page in Section 1 of the *HP-UX Reference*.

`Menu:Item`	This notation indicates a choice from the menu bar. For example, since `Quit` is on the `File` menu, the menu bar selection is written as `File:Quit`.

In This Book

The following is a brief description of the contents of this manual:

Chapter 1 Presents an overview of the debugger's graphical user interface and online help system.

Chapter 2 Describes how to compile, load, and execute a target program.

Chapter 3 Describes how to use breakpoints, watchpoints, traces, and intercepts.

Chapter 4 Describes how to view and manipulate target program data.

Chapter 5 Describes how to use the debugger's command line.

Chapter 6 Describes how to customize the debugger.

Chapter 7 Describes the concepts of blocks and environments, scope and visibility rules, and the use of qualified names.

Chapter 8 Describes how to use the debugger to handle special application requirements.

Appendix A Describes the debugger's line-mode interface.

Appendix B Describes the debugger's language managers.

Appendix C Describes the debugger's target managers.

Appendix D Describes the debugger's object managers.

Appendix E Describes the debugger's user interface managers.

This manual also contains a Glossary and Index.

Contents

1. Overview

HP/DDE at a Glance 1-2
HP/DDE Online Help 1-6
Using HP/DDE Online Help 1-7

2. Compiling, Loading, and Executing the Target Program

Preparing the Target Program 2-2
Invoking the Debugger 2-3
Setting PATH and MANPATH Variables 2-3
Stopping the Debugger 2-3
Invoking and Loading a Target Program During Debugger Startup 2-4
Invoking and Loading a Target Program From the Debugger . . 2-5
Using the File Menu 2-5
Using the debug Command 2-6
Attaching the Debugger to a Running Process 2-7
Using the File Menu 2-7
Using the debug Command 2-7
Stopping the Target Program 2-9
Restarting the Target Program 2-9
Using the File Menu 2-9
Using the restart Command 2-9
Interrupting a Running Program 2-10
Interrupting in System or Nondebuggable Routines 2-10
Examining Source Files 2-12
Executing the Target Program 2-14
Using Command Buttons 2-14
Using the go Command 2-15
Using the step Command 2-16
Using the Mouse 2-17

Looking at the Call/Return Stack 2-18
Using the tb Command 2-19
Using the environment Command 2-19

3. Using Monitors (Breakpoints, Watchpoints, Traces, and Intercepts)
Using Monitors 3-2
Setting Breakpoints 3-3
Using the Mouse 3-3
Using the breakpoint Command 3-3
Specifying Locations 3-4
Specifying Actions 3-4
Breaking at Blocks or Routines 3-4
Setting Breakpoints in Alternate Source Files 3-4
Using Breakpoints When Debugging Loops 3-5
Using the Break Menu 3-5
Using the Breakpoint Set/Change Dialog Box 3-6
Setting Watchpoints 3-8
Viewing and Modifying Watchpoints 3-9
Using Command Buttons 3-10
Using the Watch Menu 3-11
Using the Data Watchpoint Set/Change Dialog Box 3-12
Using the watchpoint Command 3-13
Setting Traces 3-14
Using the Trace Menu 3-15
Using the Trace Set/Change Dialog Box 3-16
Using the trace Command 3-17
Setting Intercepts 3-18
Using the Intercepts Dialog Box 3-18
Using the intercept Command 3-20

4. Viewing and Manipulating Target Program Data
Examining Variables and Expressions 4-2
Using Command Buttons 4-3
Using the Mouse 4-3
Using the Data Value Menu 4-4
Using Debugger Commands 4-4
Examining Arrays 4-5
Examining Objects Referenced by Pointers 4-6

Examining Linked Lists 4-7
Examining Buffers 4-8
Examining Registers 4-10
Using Register Commands 4-12

5. **Using Debugger Commands**
Abbreviating Debugger Commands 5-2
Entering Multiple Debugger Commands on One Line 5-2
Using Command Lists 5-2
Continuing Commands on the Next Line 5-3
Resolving Syntax Conflicts 5-3
Resolving Case Sensitivity 5-4
Editing the Command Line 5-5
Using the Command History Facility 5-5
Recording Command Sequences for Later Playback 5-6
Invoking Shell Commands from the Debugger 5-7
Redirecting Input and Output 5-7
Creating Alias and Define Macros 5-9
Using Reserved Identifiers and Special Macros 5-11
Combining Debugger Commands Using Action Lists 5-15
Creating Action Lists 5-15
Creating Conditional Action Lists 5-16
Understanding Action List Execution in Special Circumstances 5-16
Errors in Action Lists 5-16
Execution of Multiple Action Lists 5-16
Action List Execution Following an Interactive step Command 5-17
Placing step and go Commands in Action Lists 5-17

6. **Customizing the Debugger**
Using the Options Menu 6-2
Using Startup Command Files 6-3
Using a Personal Startup File to Customize the Debugger . . 6-4
A Sample Personal Startup File 6-5
Emulating Other Debuggers 6-8
Compatibility with xdb 6-8
Compatibility with dbx 6-9

7. **Identifying Program Objects**
Understanding Blocks and Environments 7-2
Changing the Environment 7-3
Overriding the Current Language 7-4
Applying Scope and Visibility Rules 7-4
Using Qualified Names 7-6
Block Qualified Names 7-6
Fully Qualified Names 7-7
Image Qualified Names 7-9
Special Block Qualified Forms 7-9
Qualified Names for Predefined, User-Declared, and External Symbols 7-9
Frame Block Qualified Names 7-10

8. **Debugging in Special Situations**
Examining Core Files 8-2
Attaching to a Core File 8-2
Core File Debugging 8-3
Debugging Shared Libraries 8-4
Debugging Multi-Threaded Applications 8-6
Making libdce.sl Writable 8-7
Stripped and Unstripped Versions of libdce.sl 8-7
Viewing and Manipulating Threads 8-8
Setting Breakpoints on Threads 8-9
Environment in Multi-Threaded Applications 8-10
Thread-Specific Debugger Commands 8-11
Assembly Level Debugging 8-13
Using the Assembly Instructions Dialog Box 8-13
Using Debugger Commands 8-14
Saving Assembly Code in a File 8-16
Debugging Optimized Code 8-17
Optimized Code and Unoptimized Code 8-18
What Optimization Does to Program Logic 8-18
What Optimization Does to Data 8-21
Debugging Parent and Child Processes 8-22
Debugging Applications That Use ioctl or curses 8-23
Running the Debugger Remotely 8-24

A. **Line-Mode User Interface**
Invoking the Line-Mode User Interface A-2
The User Interface Startup File A-3
Screen Display Conventions A-4
Examples A-5

B. **Language Managers**
C Language Manager B-3
C++ Language Manager B-7
FORTRAN Language Manager B-13
HP Pascal Language Manager B-18
HP-UX PA-RISC Assembly Language Manager B-22
Solaris SPARC Assembly Language Manager B-25

C. **Target Managers**
HP-UX PA-RISC Target Manager C-3
Solaris SPARC Target Manager C-7

D. **Object Managers**
HP SOM Object Manager D-3
Solaris SPARC Object Manager D-4

E. **User Interface Managers**
Graphical User Interface Manager E-3
Line-Mode User Interface Manager E-4
SoftBench User Interface Manager E-5

Glossary

Index

Figures

1-1. Debugger Main Window 1-3
1-2. The Introductory HP/DDE Help Screen 1-7
2-1. Load/Rerun Dialog Box 2-6
2-2. Source File Display 2-12
2-3. Stack View Dialog Box 2-18
3-1. The Break Menu 3-6
3-2. The Breakpoint Set/Change Dialog Box 3-7
3-3. The Data Watchpoints Dialog Box 3-9
3-4. The Watch Menu 3-11
3-5. The Data Watchpoint Set/Change Dialog Box 3-12
3-6. The Trace Menu 3-15
3-7. The Trace Set/Change Dialog Box 3-16
3-8. The Intercepts Dialog Box 3-19
4-1. The Data Value Tear-Off Menu 4-4
4-2. Show Registers Dialog Box (General) 4-11
6-1. The Options Menu 6-2
6-2. A Sample Personal Startup File 6-6
7-1. Sample Module Illustrating Scope and Visibility 7-5
7-2. Sample Modules Illustrating Fully Qualified Names 7-8
7-3. Sample Call/Return Stack and Program 7-11
8-1. The Threads Dialog Box 8-8
8-2. Stack View Dialog Box Showing Current Thread 8-10
8-3. Assembly Instructions Dialog Box 8-14
8-4. Unoptimized Code: Statement-to-Instruction Mapping 8-19
8-5. Optimized Code: Statement-to-Instruction Mapping 8-20

Tables

5-1. Reserved Identifiers and Special Macros 5-12

1

Overview

The HP Distributed Debugging Environment (also referred to as "the debugger" or "HP/DDE") is a high-level language debugger for the HP-UX operating system. The debugger operates on object files generated by HP compilers and the HP assembler.

The debugger provides a powerful graphical user interface based on OSF/Motif. For users without access to OSF/Motif, the debugger also provides a line-mode user interface.

The debugger supports both expert and novice users with an easy-to-use debugging environment that you can customize to fit your application.

This chapter gives a brief introduction to the debugger's graphical user interface and to the debugger's extensive online help.

Note HP/DDE is also the default SoftBench Program Debugger. The SoftBench version of HP/DDE has a graphical user interface that differs somewhat from the interface presented in this manual. However, the debugger commands are the same.

The SoftBench Program Debugger runs on both HP-UX and Solaris systems.

HP/DDE at a Glance

The debugger provides a powerful graphical user interface based on OSF/Motif. You control the debugger by executing commands from menus, from the command entry line, or from customizable command buttons. (The debugger also has a line-mode user interface which is described in Appendix A.)

The main window of the debugger displays source code, debugger output, and program I/O. Some of the features of the debugger main window, shown in Figure 1-1, are:

1 Menu Bar

Invoke commands or display dialog boxes.

Some dialog boxes allow you to specify options to debugger commands. Other dialog boxes include dynamic displays that allow you to view and manipulate assembly code, registers, variables, threads, call/return stacks, and more.

2 Input Box

Enter parameters for command buttons and pull-down menus.

You can enter information either by typing or by selecting text (from the source code, debugger output, or program I/O areas) for use with these commands. Text can be selected either by dragging the mouse, or by double-clicking.

3 Interrupt Button

Interrupt the debugger or the target program.

This button changes from `Interrupt Debugger` to `Interrupt Program` when the target program is executing.

4 Location Buttons

Change current location up and down the call/return stack.

Indicators in this area show when the current location differs from the program counter (`PC`) location. Locations are specified as function names.

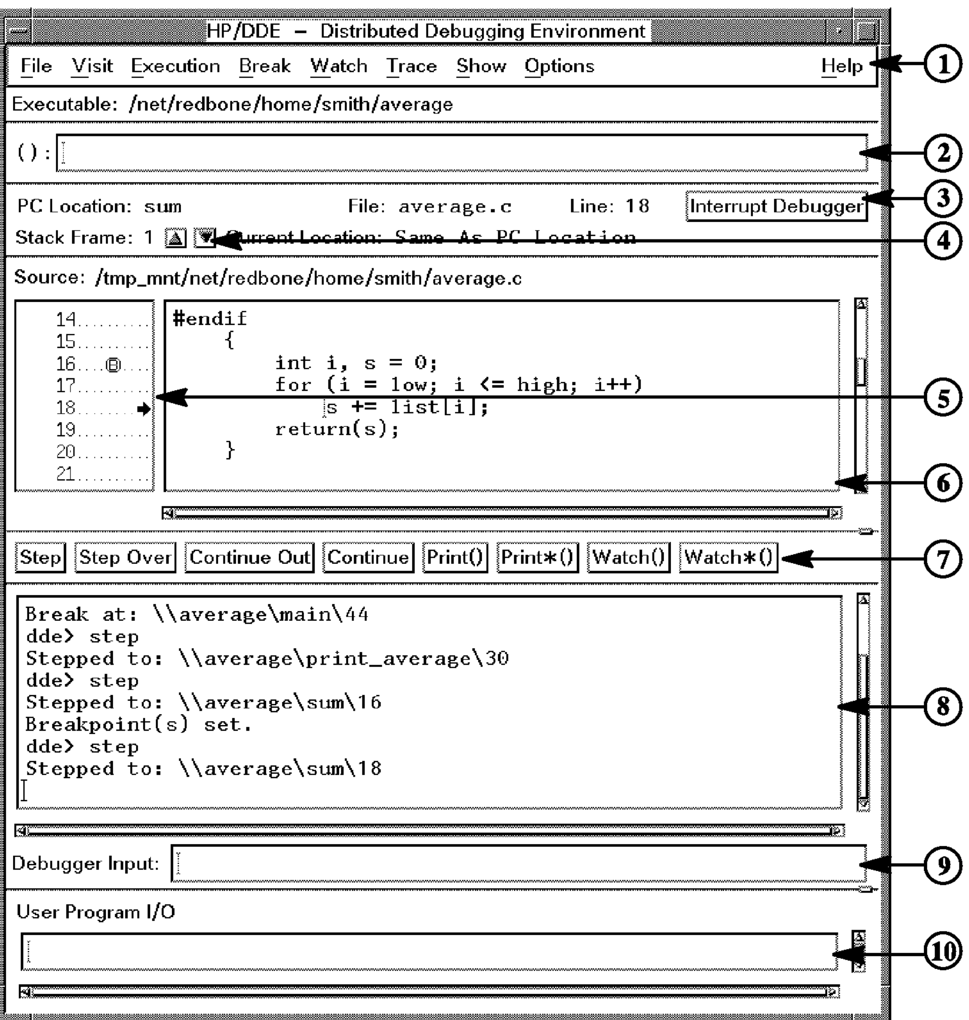


Figure 1-1. Debugger Main Window

5 Annotation Margin

Shows source file line numbers.

Click with the left mouse button on a line number to set a breakpoint. Click on the circled `B` symbol to delete the breakpoint.

An arrow indicates the current program counter location.

6 Source File Display Area

Clicking the right mouse button in this area invokes the Source Actions pop-up menu, which contains a number of program execution and monitoring commands.

To search for text strings in the source file, select `Visit:Search`.

7 Customizable Command Buttons

Invoke common debugger commands.

You can change these buttons or create additional buttons by selecting `Options:User Configurable Buttons`.

8 Debugger Output Area

This transcript pad echoes debugger commands, displays messages and warnings, and displays the output from debugger commands.

9 Debugger Command Input Box

Enter debugger commands.

For a complete list of debugger commands, their options, and usage examples see the online command reference.

10 User Program I/O Area

Target program output is displayed here. Input to target programs is entered here.

The debugger's GUI often provides several methods of performing a particular task. For example, you can:

- Enter debugger commands in the command input box.
- Invoke actions from menus, dialog boxes, or command buttons.
- Invoke actions by pointing and clicking with the mouse.

This manual attempts to describe the most common alternatives for performing each task.

In general, entering debugger commands in the command entry box provides the widest range of options and allows the greatest flexibility and control. This manual shows only the more commonly used options to debugger commands. For complete information on a particular command, see `Command Reference` in the debugger's online help. You can also enter `help` *command_name* on the debugger's command line.

Note If, when you start HP/DDE 4.0, the main debugger window does not look the way it does in Figure 1-1 (for example, if it is very small), you probably have defined the X resource `DDE.geometry` for an earlier release of HP/DDE. Comment out or remove this definition from your X resources.

HP/DDE Online Help

The debugger has an extensive hypertext-based online help facility. You can use keyword searches or click on hyperlinks to navigate through the help system.

Figure 1-2 shows the introductory help screen that you can invoke from the menu bar of the debugger's main window. Click on `Help` and then select `Overview`.

You can also invoke the debugger's online help facility from the `Top Level` selection in the Help Manager, which is located on the Front Panel.

The following is a brief description of the major topical divisions of the online help:

- `Getting Started` contains a quick start guide that shows the basic functions of the debugger's graphical user interface, and a tutorial that walks you through basic debugging procedures.
- `Common Debugging Tasks: Graphical User Interface` describes how to invoke general debugger functions with an emphasis on using the graphical user interface.
- `Common Debugging Tasks: Command Line` describes how to invoke general debugger functions with an emphasis on using debugger commands.
- `Graphical User Interface` describes the various debugger display windows and shows how to customize the graphical user interface.
- `Command Reference` describes debugger commands and options and contains useful examples. (You can also invoke help on a particular command by invoking `help` *command_name* in the debugger command input box.)
- `Commenting on the HP/DDE Online Help System` describes how you can send your comments about the online help to Hewlett-Packard.

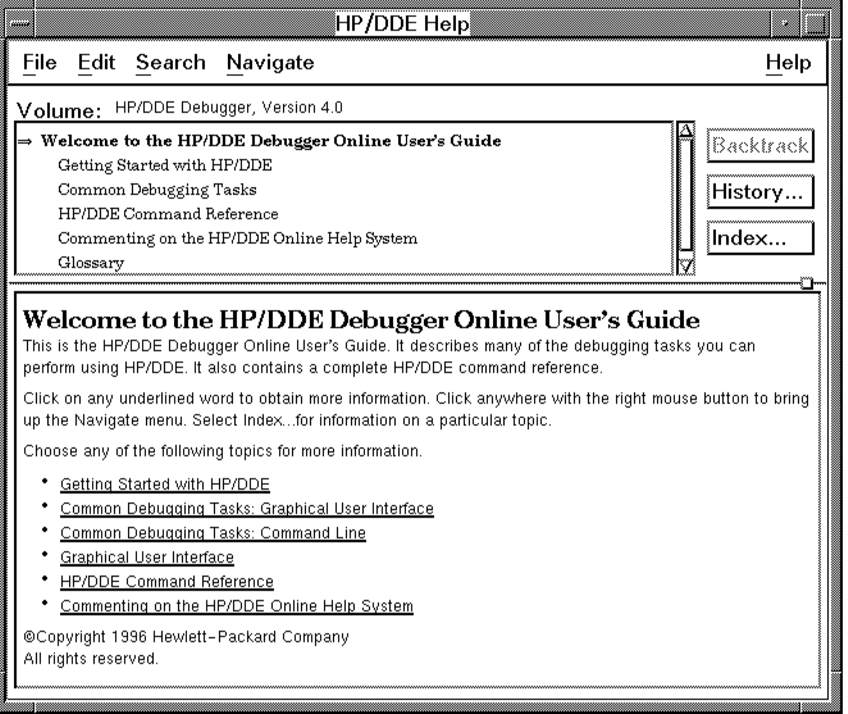


Figure 1-2. The Introductory HP/DDE Help Screen

Using HP/DDE Online Help

HP/DDE online help provides several ways to find the topic you are looking for. Here are some of the methods you can use:

- Clicking on a hyperlink. Hyperlinks connect topics within and among the major divisions of the online help. Hyperlinks are displayed as underlined text.
- Searching for a keyword. Select the `Index` button to invoke a dialog box that allows you to search through the keyword index. Click on an index item to display the help on that item.
- Using the `help` command. The command `help` *command_name* invokes the reference page for the specified command.

- Pressing F1 to obtain context-sensitive help in dialog boxes and display windows.

You can navigate through the HP/DDE online help using several methods, including:

- Clicking on an item from the topic hierarchy in the Help window to go to another topic.
- Clicking on an item in the history list to return to a previous topic. Select the `History` button from to invoke the a dialog box that shows a list of topics that you have visited.
- Selecting the `Backtrack` button to return to the previous topic.

More information on using the HP Help System is available from the debugger's menu bar by clicking on `Help` and then selecting `Using Help`.

2 Compiling, Loading, and Executing the Target Program

This chapter describes:

- Preparing a program for debugging.
- Starting and stopping the debugger.
- Load and unloading a target program.
- Examining source files.
- Controlling the execution of the target program.
- Viewing the current execution call stack.

For more information on the debugger commands mentioned in this chapter see the online command reference. You can also invoke help on a particular command by entering `help` *command_name* in the debugger command input box.

The online help also contains more information on using the debugger's graphical user interface.

Preparing the Target Program

To prepare a target program for debugging, compile the program using the compiler's option for debugging, usually `-g`. A compiler's debugging option causes it to add information needed by the debugger to the program's object file. For example:

```
$ cc -g average.c
```

You can find `average.c`, a sample program that you can practice on, in the directory `/opt/langtools/dde/examples`. In that directory, you can also find sources for `average` in FORTRAN, C++, and Pascal.

If you attempt to debug a program that was compiled without the debugging option, the debugger displays a message similar to the following:

```
(Warning) Object file has no debug information.
Limited debugging will be available.
```

Some of the limitations on debugging code compiled without the `-g` option are:

- You can access the values of global variables but not their descriptions.
- Line numbers are not available in the Assembly Instructions dialog box.
- You can set breakpoints on entry points using procedure names, entry point names, and virtual addresses. However, if some modules were compiled with `-g` and some were compiled without `-g`, you must use the `initialize -altdbinfo` command to set breakpoints on those modules compiled without `-g`. (Use of the `initialize -altdbinfo` command is not necessary on Solaris systems.)
- The debugger will not automatically display source code. You can use the `use source` command to display source code. However, you cannot follow program execution or invoke debugger commands in the source file display area.

Shared libraries, which are often compiled without `-g`, are a special case. See "Debugging Shared Libraries" in Chapter 8 for more information.

Invoking the Debugger

To invoke the debugger, enter the `dde` command in a shell process window. For detailed information on the `dde` command and its options, refer to the *dde*(1) man page.

When you invoke the debugger, it first executes a user interface startup file. Then, it executes any commands specified with the `-do` option. For example, the following command line specifies where the debugger should search for source files:

```
$ dde -do "property sdir ~/src"
```

Next, the debugger executes startup command files, including your personal startup file if you have created one. See "Using Startup Command Files" in Chapter 6 for more information on startup files.

When you invoke the debugger, the debugger's main window appears (see Figure 1-1).

Note Starting the debugger from the command line is not supported on Solaris systems. See the *softdebug*(1) man page for information on starting the debugger.

Setting PATH and MANPATH Variables

You must have the `/opt/langtools/bin` directory in your `PATH` in order to use the `dde` command. If you have `/usr/softbench/bin` in your `PATH`, `/opt/langtools/bin` must precede it.

You must have the `/opt/langtools/share/man` directory in your `MANPATH` to access the *dde*(1) man page.

Stopping the Debugger

To stop and exit the debugger, choose `File:Quit` from the menu bar or enter the `quit` command.

If a target program is running, the debugger kills it. However, if the target program is an attached process (see "Attaching the Debugger to a Running Process"), the debugger frees it.

Invoking and Loading a Target Program During Debugger Startup

If you want the debugger to load the target program at startup, then enter the target program invocation as the last argument to the `dde` command. For example, if the target program name is `test`, enter

```
$ dde test
```

If the target program requires arguments, precede the target program and its arguments with a double hyphen (`--`). Using the double hyphen prevents options (particularly X Window System options) from being interpreted as options to the `dde` command.

For example, if the target program is called `test`, and `test` takes the X option `-synchronous`, enter

```
$ dde -- test -synchronous
```

Any options to the `dde` command are entered before the double hyphen.The following example shows the use of the X options `-bg` and `-fg` to set the colors for the debugger's windows:

```
$ dde -bg black -fg white  -- test -synchronous
```

Invoking and Loading a Target Program From the Debugger

You can invoke and load a target program from the File menu or by using the `debug` command.

Using the File Menu

Select `File:Load Executable` to invoke a target program. The Load/Rerun dialog box, shown in Figure 2-1, appears.

The Load/Rerun dialog box allows:

1. Specifying the path name of the target program. (The [...] buttons invoke a dialog box containing a list of directories and file names under the current working directory.)
2. Specifying program arguments.
3. Changing the current working directory.
4. Redirecting `stdin`, `stdout`, and `stderr`. `Append` and `Replace` buttons allow `stdout` and `stderr` to be added to or to overwrite existing files.
5. Specifying the environment variables to be passed to the target programs. A window above the input boxes lists the current settings.

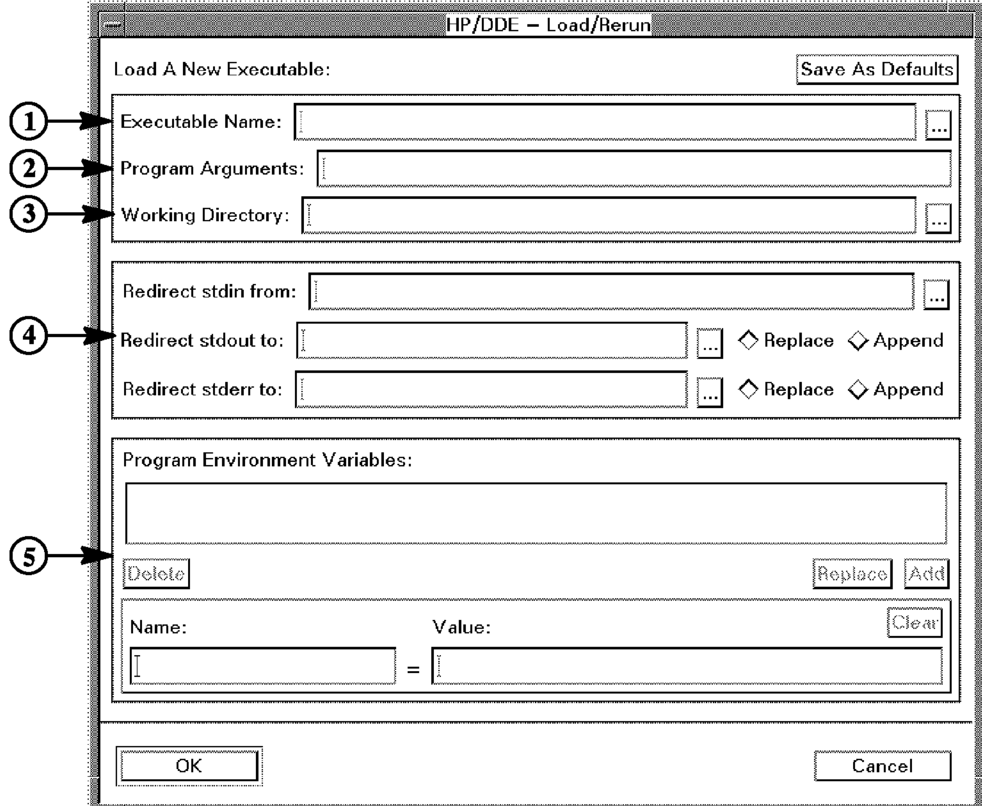


Figure 2-1. Load/Rerun Dialog Box

Using the debug Command

To invoke a target program from the debugger command input box, enter the `debug` command followed by the target program invocation.

For example, if the target program `test` takes the argument `-F testfile`, enter the following command:

```
debug test -F testfile
```

To redirect the target program's standard input and output, use the `debug` command's `-input` and `-output` options.

Attaching the Debugger to a Running Process

To debug a running process, you must either have the same user ID as the target program or you must be logged in as `root`.

Using the File Menu

To attach to a running process from the debugger menu bar:

1. If the debugger's current directory is not the same as the directory that contains the target process, change it using `File:Change Working Directory`.
2. Enter the name of the process in the `():` parameter input box.
3. Select `File:Debug Running Process()`.

Using the debug Command

From the command input box, enter the `debug` command with the `-attach` option, specifying a process ID. The *ps*(1) command displays process IDs.

If the debugger's current directory is not the same as the directory that contains the target process, use either the *object_program_pathname* argument or the `-wd` option to the `debug` command.

You can set breakpoints in the main program of an attached process. On HP-UX systems, you cannot set breakpoints in shared libraries used by the program unless you run

`/usr/lib/pxdb -s on` *executable_file*

on the program before executing it. See "Debugging Shared Libraries" in Chapter 8 for more information on debugging shared libraries.

When you attach to a running process, the process may be executing in kernel code (for example, if it is waiting for input). In this case, no source file appears in the source file display area, and the debugger issues a warning message.

You may find it helpful to use the `tb` and `environment` commands to place you in the source code. The `tb` command displays a traceback of the call/return stack. From the traceback you can find a frame associated with a procedure in

the source code. Use the frame block qualifier, `‘main(n)`, from the traceback as an argument to the `environment` command. For example:

```
debug -attach 1189
Attached to process 1189; initializing "/home/smith/progs/and".
Stopped at: ‘va(800b8658) (800B8658)
step
(Warning) Unable to determine statement boundaries;
          stepping 1 instruction.
Stepped to: read+0010 (800B8658)
tb
‘main(7):   Stopped at: read+0010 (800B8658)
‘main(6):   Called from: _filbuf+0100 (800A5CC0)
‘main(5):   Called from: _doscan+0a78 (80096D2C)
‘main(4):   Called from: _scanf+0024 (8007E8F4)
‘main(3):   Called from: \\and\main\12 (00001DCC)
‘main(2):   Called from: _start+0068 (80041D5C)
‘main:      Called from: $START$+0094 (0000177C)
env ‘main(3)
Environment: \\and\main\12 (00001DCC) (frame ‘main(3))
Stopped at: read+0010 (800B8658)
breakpoint 13
go
```

If you get a `Permission denied` error message when you attach to a running process, it is likely that you are running either the debugger or the target process over an NFS link. The relevant file system may be mounted with the default `intr` option. You must mount the file system with the `nointr` option to resolve this problem. Use a command like the following to mount the file system containing the debugger:

```
$ mount -o nointr[,other_options] system:/opt/langtools /tools
```

Use a command like the following to mount the file system containing the target process:

```
$ mount -o nointr[,other_options] system:/test_area /test
```

It is probably easier to create an auxiliary mount for the file system than to unmount and remount it.

Stopping the Target Program

To terminate the target program, use the `kill` command or select `File:Unload Executable`. When the target program terminates, the debugger continues to run. Use `quit` or select `File:Quit` to terminate the debugger.

Use the `free` command to detach the debugger from the target program while allowing the target program to continue executing.

Restarting the Target Program

Using the File Menu

To restart the target program from the menu bar, select:

- `File:Rerun` to restart the target without changing the runtime environment.
- `File:Rerun ...` to restart the target with changes to the runtime environment.

Using the restart Command

Use the `restart` command to restart the target program while preserving breakpoints and watchpoints.

The `restart` command restarts the currently loaded target program. If no target program is loaded, it restarts the last target program that ran in the current debugging session.

A program may not restart correctly if its object file was modified after the original `debug` command. Breakpoints, for example, may be reset at inappropriate locations after restarting.

If you do not want to preserve breakpoints, watchpoints, and symbol information, kill the target program (or let it finish executing) and re-enter the `debug` command.

Without options, the `restart` command restarts the target program using the same arguments specified in the previous invocation. Use the `-args` option if you want to specify different arguments in the target program invocation.

For example, the following command restarts the currently running target program, replacing previously specified arguments with the argument `-s`:

```
restart -args -s
```

Interrupting a Running Program

When the PC Location indication is `Running` ... , your program has control, and you cannot interact with the debugger. (A small clock animation is also displayed.) Any commands to the debugger (except commands that restart, kill, or unload your program) will be queued until your program returns control to the debugger.

Selecting [Interrupt Program] will flush any queued commands and return control to the debugger.

If you interrupt your program while it is executing code that was compiled with the debug options on, you can continue working just as if you had encountered a breakpoint at that location. A PC arrow appears in the annotation margin and the source for the code is displayed. At this point the PC Location shows a valid location and you can enter debugger commands.

Interrupting in System or Nondebuggable Routines

If you interrupt the program while it is executing some system-supplied routine, or while it is executing a routine that was compiled without debugging information, the PC Location may consist of a virtual address. The source file display area will be cleared to indicate that no source is available.

You cannot examine local variables or step through statements. You can only step by assembly instructions, and examine other procedures on the call stack.

You can run the nondebuggable routine until it reaches the point where it returns to its calling procedure by selecting [Continue Out]. You can continue

doing this until your program returns to debuggable code. You could also set a breakpoint at some later point in debuggable code.

If the nondebuggable code is in an infinite loop, or will not return for some other reason, you must kill or rerun the program.

Examining Source Files

When you load a target program, the source file is displayed in the source file display area. See Figure 2-2.

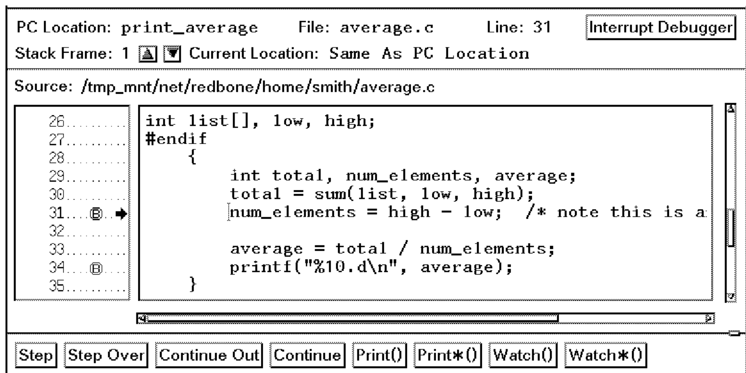


Figure 2-2. Source File Display

An arrow symbol in the annotation margin indicates the current point of execution. The circled `B` symbol designates a breakpoint.

Ordinarily, the debugger sets breakpoints at the program's entry and exit statements and executes the program up to its entry statement. You can create and delete breakpoints by clicking with the left mouse button on a line number in the annotation margin.

You can search for text strings in the source by selecting `Visit:Search` or CTRL-S.

If the target program was compiled from a number of source files, you can display other files by selecting `Visit:File()` after entering the source file name in the `():` input box.

In addition, you can use the `environment` command. You must enclose the source file name in double quotes. For example:

```
env "source_filename"
```

If you move a source file after compiling the program, the debugger may be unable to find the source file. It will issue the warning: `Unable to access`

`object`. Select `File:Add Source Directories` or use the `property sdir` command to put the source file in the debugger's search path.

Also, the source will not appear if the main program is not debuggable (even if there are debuggable modules within the target program). A message like the following may appear during startup:

```
(Warning) end.o has not been linked in.
(Warning) Notification of dynamic loader events
          will be unavailable.
Executing image in process 8981: "/home/smith/a.out".
Stopped at: $START$ (00001894)
```

However, you can view debuggable modules by selecting `Visit:Environment()`, `Visit:File()` or using the `environment` command. Specify "*source_filename*", which is the leaf name (enclosed with double quotes) of the source file of a debuggable module.

Executing the Target Program

The debugger can execute the target program by stepping through one or more statements at a time.

The easiest method for stepping through a target program is to use one of the step or continue command buttons. They are located below the source file display area.

The debugger commands `go`, `step`, and `goto` are useful when you want to implement more complicated stepping procedures. The less commonly used `call` command is particularly useful for executing user-created debugging routines or for testing new routines in your program.

In addition, you can invoke execution commands from the `Execute` menu or from a popup menu in the source file display area. The popup menu appears when you click with the right mouse button.

The following sections give a brief description of the command buttons and the debugger commands.

Using Command Buttons

The following buttons are located below the source file display area:

Step — Execute one statement, then stop. This is called single step execution.

Step Over — Execute a statement, treating any procedure call as a single statement. The procedure is called, but control does not return to the debugger until the procedure returns. When the PC is just before a procedure call, this has the effect of "stepping over" the call.

Continue Out — Execute until the current procedure completes and returns to its caller, or until a breakpoint (or another event that halts execution) is encountered. This is very useful when you accidentally step into a procedure that you do not want to step through, or when you interrupt your program in the middle of nondebuggable code. Each **Continue Out** will cause your program to "pop out" one procedure level.

(Continue) Execute until a breakpoint (or another event that halts execution) is encountered.

When you select one of these buttons, the PC arrow moves to the next statement to be executed.

Routines that are not debuggable, such as system library routines and routines that were not compiled with the `-g` option, will be stepped over even when using (Step).

To pause at a specific point in your program, see "Setting Breakpoints" in Chapter 3.

You can modify the default behavior of these buttons by selecting `Options:User Configurable Buttons`. The `User Configurable Buttons` dialog box is displayed. See the online help for this dialog box for more information.

Using the go Command

The `go` command begins or resumes target program execution.

Execution begins at the current point of execution. If you have just entered the `debug` command, the current point of execution is the first executable statement in the source code.

If you enter `go` without options, execution continues until a program event occurs, such as a breakpoint, program signal, or program exit.

If you want program execution to occur up to a certain program location, use the `-until` option. For example,

```
go -until freestack
```

executes the target program until the entry statement for the routine `freestack` is reached and

```
go -until 110
```

executes the target program until line 110 of the source code is reached.

Using the step Command

The `step` command advances target program execution one source code statement at a time. If you supply a numeric argument to the `step` command (for example, `step 5`), it will advance execution by the specified number of source code statements.

If you specify `step -instruction`, execution advances one assembly instruction. The PC location arrow changes to a broken variant to indicate when the current point of execution is either at another source statement on the same line or at an instruction beyond the statement's first instruction.

However, you must invoke the Assembly Instructions dialog box to see stepping through assembly instructions. Invoke the Assembly Instructions dialog box from `Show:Assembly Instructions`.

If the current point of execution is a routine call, `step` advances execution to the first executable statement in the called routine. To advance execution without stepping into a routine, use `step -over`.

If you accidentally step into a routine, use the `go -return` command to advance execution to the statement following the routine call.

The `step` command also has a `-return` option, but it takes much longer to complete.

By default, the `step` command does not step into system or library calls. If you want to step into system or library calls, you can enable the loading of symbol information from the Dynamic Images dialog box. Select `Execution:Enable Images/Libraries` to invoke it.

Alternatively, use the `property libraries` command with the appropriate library as an argument if you want to step into system or library calls.

See "Debugging Shared Libraries" in Chapter 8 for information about debugging shared libraries.

Using the Mouse

A number of execution commands are available by holding down the right mouse button in the source file display area. The `Source Actions` popup menu appears.

For example, you can:

1. Scroll to a particular line number in the source file.
2. Position the cursor on the line.
3. Invoke the `Continue Until` command from the `Source Actions` menu and it will execute the target up to that line in the code.

Looking at the Call/Return Stack

Select `Show:Stack` to display the Stack View dialog box.

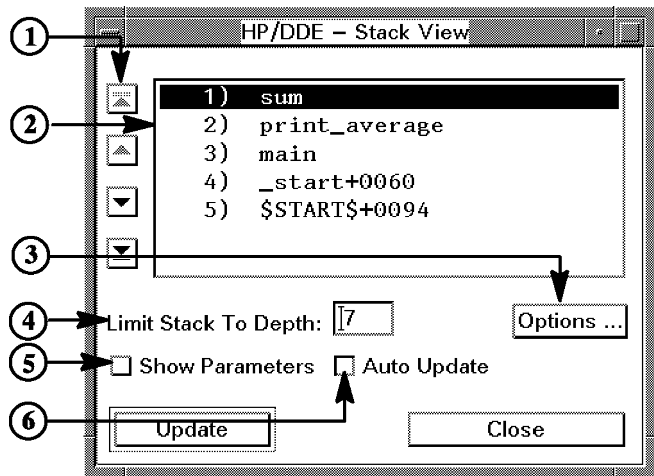


Figure 2-3. Stack View Dialog Box

The Stack View dialog box, shown in Figure 2-3, contains:

1. Arrow buttons to move to the top of the stack, up one level, down one level, or to the bottom of the stack. The associated source code is displayed in the source file display area. (You can also use the arrow buttons above the source file display area.)
2. A scrollable listing of the call/return stack. When you click on an item in the list, the associated source code is displayed in the source file display area.
3. A button that invokes the Stack Options dialog box, which allows you to limit stack depth globally and to modify stack numbering behavior.
4. An input box that allows you the limit the stack depth displayed.
5. A toggle button to display or hide parameters for each entry in the call/return stack.
6. A toggle button to control when the display is updated.

Using the tb Command

Use the `tb` command to display a traceback of the call/return stack in the debugger output area.

The `tb` command numbers each frame in the stack using the notation `` `main(``*n*`)`, where *n* increases in the direction of the most recent frame. The following example illustrates this notation which was displayed after stepping to line 33 of the sample program `average`:

```
tb
`main(4):   Stopped at: \\average\print_average\33
`main(3):   Called from: \\average\main\44 (0000203C)
`main(2):   Called from: _start+0068 (80041D9C)
`main:      Called from: $START$+0094 (0000192C)
```

Using the environment Command

To move up and down the call/return stack to view the associated source code, enter the `environment` command. Use the notation `` `env(-1) `` to represent the caller of the current routine, as shown in the following example:

```
environment `env(-1)
```

After you enter the preceding command, you can easily examine the program data local to the routine shown in the source file display area.

To move down the stack, enter the following command:

```
environment `env(+1)
```

To return to the current point of execution, regardless of the source currently displayed, enter the following command:

```
environment `run
```

For details on the `` `env `` and `` `run `` notation, see "Frame Block Qualified Names" in Chapter 7. For a description of the HP/DDE concept of environment, see "Understanding Blocks and Environments" in Chapter 7.

3 Using Monitors (Breakpoints, Watchpoints, Traces, and Intercepts)

This chapter describes the use of breakpoints, watchpoints, traces, and intercepts to monitor and control target program execution. Topics include:

- Using Monitors
- Setting Breakpoints
- Setting Watchpoints
- Setting Traces
- Setting Intercepts

For more information on the debugger commands mentioned in this chapter see the online command reference. You can also invoke help on a particular command by entering `help` *command_name* in the debugger command input box.

The online help also contains more information on using the debugger's graphical user interface.

Using Monitors

Monitors are useful for stopping program execution at specific statements, for tracing program execution, for watching variables for a change in value, and for intercepting program events. The debugger provides these types of monitors:

- A breakpoint stops execution at a specified source code statement, then reports the current target program location. Execution is resumed only when you enter another program execution command, such as `go` or `step`.
- A watchpoint monitors a selected variable or address range and reports the value of the variable or address range only when that value changes. You can specify whether a watchpoint is in effect at every source statement, at every instruction, or only at routine entry or exit points.
- A trace suspends execution, reports the current program location, then continues executing the target program. You can specify whether a trace is in effect at every source statement, at every instruction, or only at routine entry or exit points.
- An intercept monitors selected programming events such as the reception of signals from the operating system, the loading or removal of an image from a program's address space, and the termination of the program. The Intercepts dialog box (invoked from `Execution:Signals/Intercepts`) contains a list of the available intercepts and their current status.

The debugger provides several general commands that allow you to view and modify monitors as a group. These include:

- `list monitors`
- `delete monitors`
- `suspend monitors`
- `activate monitors`

For more information on these commands and their options, see the debugger's online command reference.

In addition, the debugger provides commands, menus, and dialog boxes for controlling each type of monitor. The following sections describe how to set, view, and modify breakpoints, traces, watchpoints, and intercepts.

Setting Breakpoints

Set breakpoints on lines in your code where you want execution to stop.

When a breakpoint is set, a circled `B` symbol symbol appears next to the line number in the annotation margin.

The target program stops just prior to the execution of the statement on which you have set a breakpoint. Each time execution stops at a breakpoint, the debugger output area displays the line number of the next statement that will execute.

Using the Mouse

Perhaps the easiest method for manipulating breakpoints is to use the mouse as follows:

- Click the left mouse button on a line number to set a new breakpoint. The circled `B` breakpoint symbol appears.
- Click the left mouse button on a breakpoint symbol to delete an existing breakpoint. The breakpoint symbol disappears.
- Click the middle mouse button on a breakpoint symbol to suspend it. The breakpoint symbol is redisplayed with a slash through it.
- Click the middle mouse button on a suspended breakpoint to activate it. The slash in the breakpoint symbol disappears.
- Press (Shift) and click the left mouse button on a line number to invoke the Breakpoint Set/Change dialog box. The dialog box allows you to set various attributes for breakpoints.

Using the breakpoint Command

The `breakpoint` command allows greater flexibility than the mouse in both setting and in customizing breakpoints. The following sections indicate some of that flexibility. See the online command reference for more information.

Specifying Locations

To set a breakpoint at a specific program location, enter the `breakpoint` command. For example,

`breakpoint sum` Sets a breakpoint at the first executable statement in the procedure `sum`.

`breakpoint 16` Sets a breakpoint at the statement on line 16 of the source file.

Specifying Actions

If you want a specific action to occur automatically when the target program encounters a breakpoint, use the `-do` option. For example,

```
breakpoint 43 -do [print tmp; go] -silent
```

In this example, the debugger prints the variable `tmp` in the debugger output area, then issues the `go` command whenever the target program encounters the breakpoint at line 43. The `-silent` option prevents the breakpoint from being reported in the debugger output area; only the value of the variable `tmp` is printed.

Breaking at Blocks or Routines

If you want to set breakpoints at every routine in a block or file, you can do so with a single `breakpoint` command. Use the `-in` option to specify the block name or file name. File names must be enclosed in double quotation marks, as shown in the following example:

```
breakpoint -in "iface.c" -do [args; go]
```

For detailed information on how to specify blocks and files, see the online help regarding specifying locations to the debugger.

Setting Breakpoints in Alternate Source Files

When a target program is compiled from multiple source files, the Source Display window displays the file that contains the current point of execution.

You can use the environment command with a "*filename*" argument to display another source file in the Source File Display area. Then you can set breakpoints using one of the methods described in this section.

Using Breakpoints When Debugging Loops

The `breakpoint` command option `-after` *count* is useful for debugging large loops in code. The `-after` *count* option specifies that the statement associated with the breakpoint must execute *count* times before execution is interrupted. This lets you control how many iterations of the loop occur before the debugger interrupts target program execution.

Also note that if you try to set a breakpoint at the beginning or end of a loop, the following warning may be displayed:

```
(Warning) Due to optimization, address resolved
to more than one location
```

Although the code may not be optimized, lines at the beginning and end of loops contain multiple fragments. You will need to follow the instructions on setting breakpoints in optimized code described in the online help.

Using the Break Menu

A variety of breakpoint commands are available from the `Break` menu, including support for setting breakpoints in C++ programs. Many entries on the `Break` menu allow you to set a breakpoint associated with an expression entered in the `( ):` input box.

An easy way to enter tokens in the `( ):` input box is to use the mouse to highlight expressions in the source display. Highlight by dragging with the left mouse button depressed. When you release the left mouse button, the expression appears in the `( ):` input box.

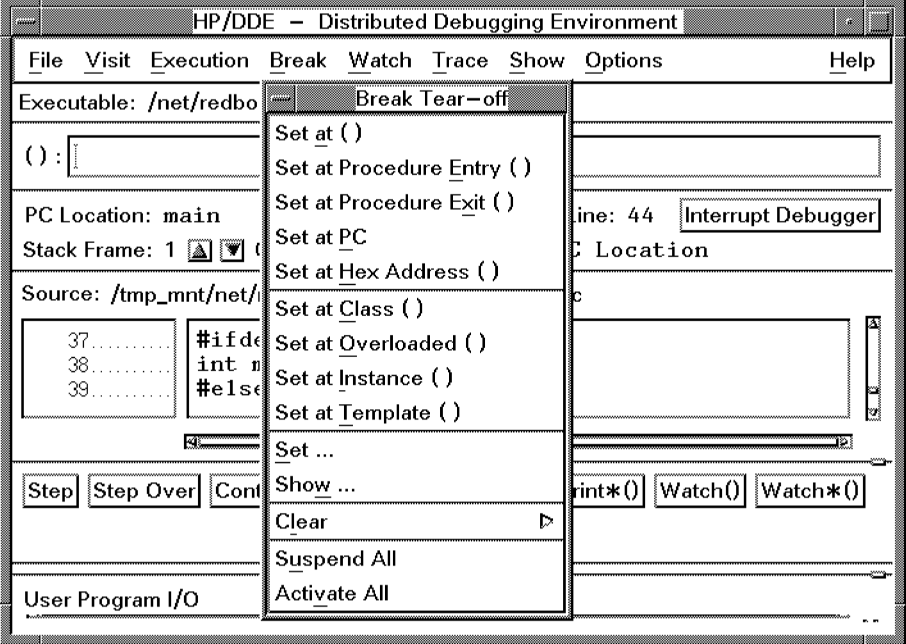


Figure 3-1. The Break Menu

As Figure 3-1 shows, the `Break` menu is a Tear-off Menu. When you click on the dashed line at the top of the `Break` menu, the menu is displayed in its own window. The menu persists so that you can invoke commands from it without having to redisplay it. You can also move the menu to a convenient place in your workspace.

Using the Breakpoint Set/Change Dialog Box

The Breakpoint Set/Change dialog box allows you to specify all aspects of a breakpoint. Invoke it by selecting `Breakpoint:Set`. The dialog box shown in Figure 3-2 appears.

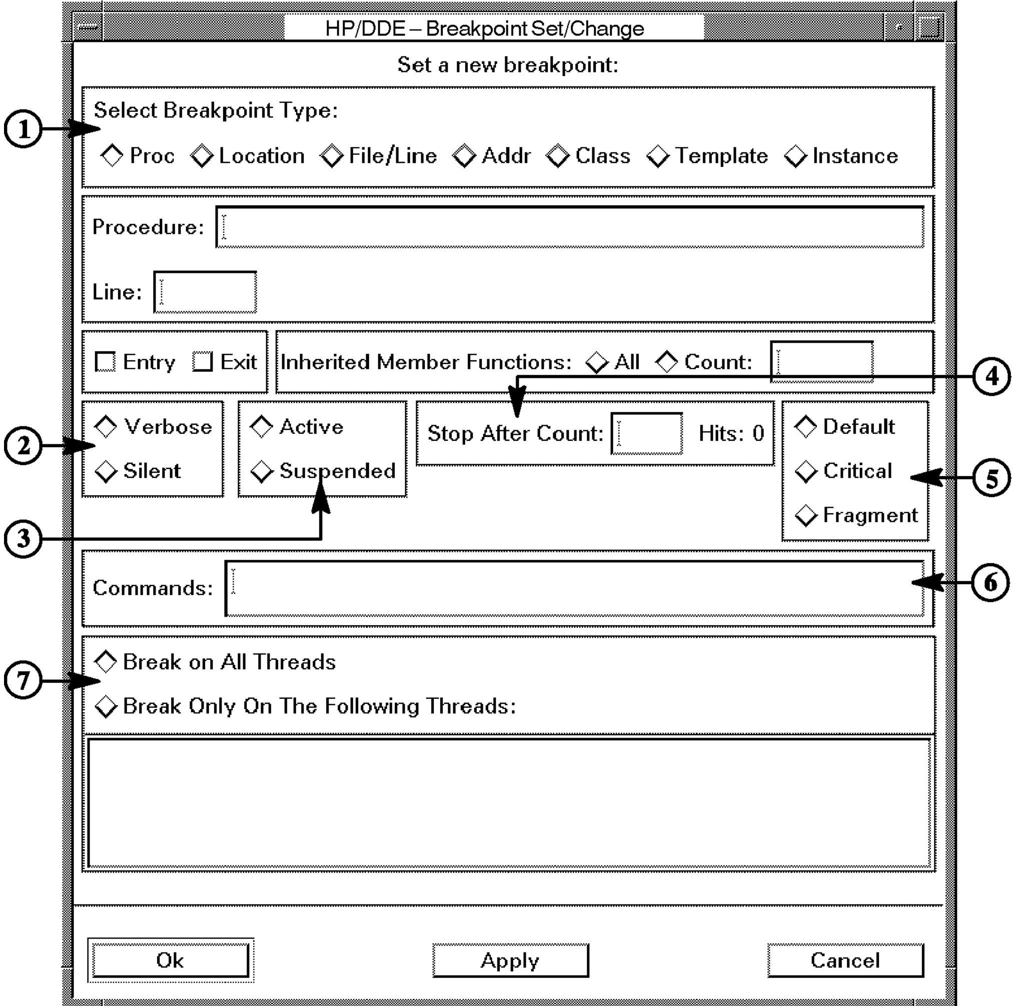


Figure 3-2. The Breakpoint Set/Change Dialog Box

From the Breakpoint Set/Change dialog box, you can:

1. Select the type of breakpoint.

   The area directly under the `Select Breakpoint Type` radio buttons changes to reflect the information needed by the breakpoint type. Enter the appropriate information for the selected breakpoint type.
2. Control whether or not messages appear when a breakpoint is hit.
3. Temporarily disable or re-activate breakpoints.
4. Stop execution only after the breakpoint has been reached a specified number of times.
5. Set location mapping for the breakpoint when debugging optimized code.
6. Specify debugger commands to execute when a breakpoint is hit.
7. Set breakpoints on specified threads when debugging a multi-threaded application.

To see a listing of breakpoints, select `Break:Show`.

Setting Watchpoints

Use watchpoints to monitor the value of a variable or memory range. (See "Examining Registers" in Chapter 4 for information on monitoring registers.) Values are displayed in the Data Watchpoint dialog box as well as in the debugger output area.

When you create a watchpoint, you specify the expression or address range to monitor, and a granularity. The granularity specifies how often the value should be reported: on procedure entry, on procedure exit, on procedure entry *and* exit, at every statement, at every machine instruction, or whenever the program stops and returns control to the debugger.

The default granularity is to stop and report value changes only when the target program itself stops and returns control to the debugger. If you needed to monitor a variable more closely, you could specify a instruction- or statement-level granularity. At instruction-level granularity, for example,

the value of a variable is checked after every assembly instruction executes. Execution of the target program stops when the value changes.

For performance purposes, you should set the granularity as coarsely as possible. For example, if you only need to know the value of the monitored expression each time you enter a procedure, there is no sense in monitoring it after every assembly instruction. Typically you would locate a problem by using a granularity of procedure entry/exit to narrow the source of the problem down to one procedure. Once the problem is localized, use a finer granularity (such as instruction or statement) but limit it to a particular procedure by entering the procedure name in the `When In` input box in the Data Watchpoint Set/Change dialog box.

Viewing and Modifying Watchpoints

The following sections describe some of the methods for setting watchpoints. When you set a watchpoint by any method, the Data Watchpoints dialog box is displayed (you can also invoke it by selecting `Watch:Values Display`). See Figure 3-3.

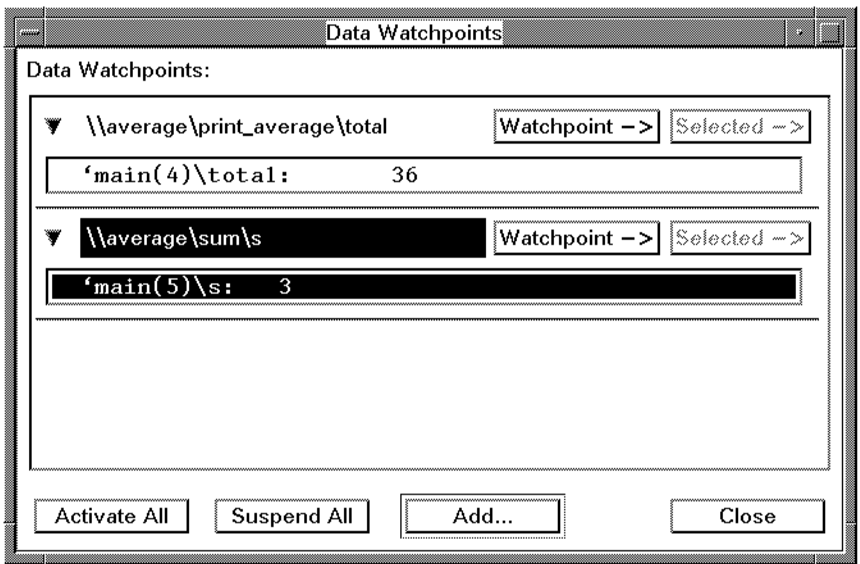


Figure 3-3. The Data Watchpoints Dialog Box

The Data Watchpoints dialog box shows detailed information on each watchpoint and it allows you to manipulate both the watchpoints and the display itself.

Each watchpoint displays the variables or memory area it is attached to. The data display highlights when the data being watched is modified. Complete watchpoints can be hidden by selecting the ▼ button next to the watchpoint. Select the button again (which is now ▶) to redisplay the watchpoint.

Individual elements in a watchpoint (such as some elements in an array) can be hidden by selecting them and selecting Selected -> `Hide`. Select Watchpoint -> `Show All Hidden Values` to redisplay them.

Compound objects (such as arrays or structures) can be collapsed to simplify the display, either by double-clicking the beginning of the compound object, or by selecting it and choosing Selected -> `Collapse`. Double-click the collapsed object, or choose Selected -> `Expand`, to display the entire object.

To modify the watchpoint, choose Watchpoint -> `Change`. The `Data Watchpoints` dialog box is displayed, allowing you to modify all editable attributes of the watchpoint.

To change the `Active/Suspend` status of a watchpoint, use the `Data Watchpoints` dialog box. You can also activate or suspend all watchpoints by selecting the Activate All or Suspend All buttons in the `Data Watchpoints` window, or by choosing the `Watch: Suspend ALL` or `Watch: Activate ALL` menu selections.

Use Watchpoint -> `Delete` to eliminate a watchpoint.

Using Command Buttons

The following buttons are located below the source file display area:

Watch() — Set a watchpoint on the contents of the `( ):` input box. Check the watchpoint whenever the debugger stops.

Watch*() — Set a watchpoint on the location pointed to by the pointer in the `( ):` input box. Check the watchpoint whenever the debugger stops.

An easy way to enter tokens in the `( ):` input box is to use the mouse to highlight expressions in the source display. Highlight by dragging with the

left mouse button depressed. When you release the left mouse button, the expression appears in the ( ): input box.

Use Options:User Configurable Buttons to modify the default behavior of the command buttons.

Using the Watch Menu

A variety of watchpoint commands are available from the Watch menu. Many entries on the Watch menu allow you to set a watchpoint associated with an expression entered in the ( ): input box.

As Figure 3-4 shows, the Watch menu is a Tear-off Menu. When you click on the dashed line at the top of the Watch menu, the menu is displayed in its own window. The menu persists so that you can invoke commands from it without having to redisplay it. You can also move the menu to a convenient place in your workspace.

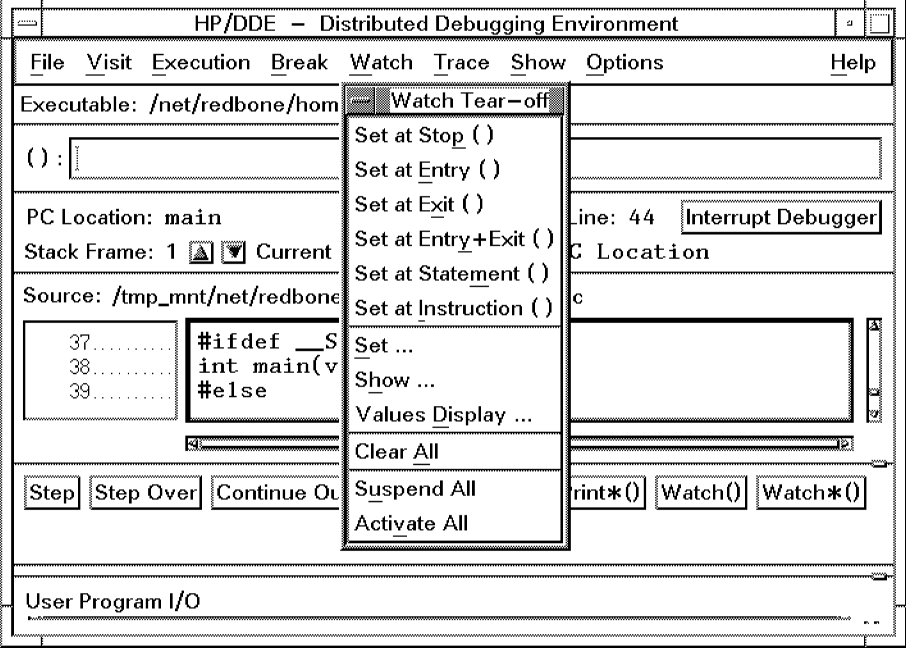


Figure 3-4. The Watch Menu

Using the Data Watchpoint Set/Change Dialog Box

The Data Watchpoint Set/Change dialog box allows you to specify all aspects of a watchpoint. Invoke it by selecting `Watch:Set`. The dialog box in Figure 3-5 appears.

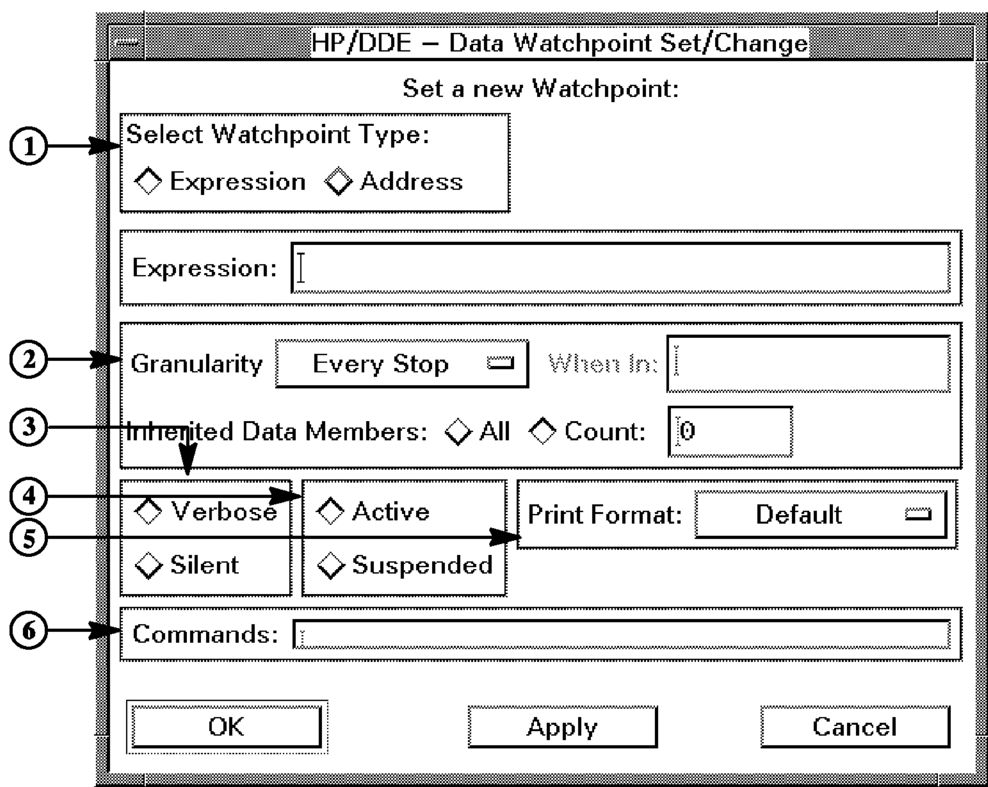


Figure 3-5. The Data Watchpoint Set/Change Dialog Box

From the Data Watchpoint Set/Change dialog box, you can:

1. Select the type of watchpoint.

   The area under the `Select Watchpoint Type` radio buttons changes to reflect the information needed by the watchpoint type. Enter the appropriate information for the selected watchpoint type.

Note that you use the (Print ( )) button on the expression &*var* to find the address of the variable *var*.

2. Select the granularity (how often the value is checked).

   The default, `Every Stop`, causes the debugger to check values only when target program execution stops for some event like a breakpoint.

   If you want target program execution to stop whenever the value changes, choose `Statement` granularity. Be aware, however, that `Statement` granularity can slow down performance of the target program.

   You can also restrict a watchpoint to be active only in a specified block.

   In C++ programs, you can specify how many levels of inherited data members should be included.

3. Control whether or not messages appear when a watchpoint is hit.
4. Temporarily disable or re-activate watchpoints.
5. Specify the format in which the value is displayed. The default is to display the value as it is declared.
6. Specify debugger commands to execute when a watchpoint is hit.

To see a listing of watchpoints, select `Watch:Show`.

Using the watchpoint Command

The following command sets a watchpoint on the variable `idx`. By default, the watchpoint is in effect at every statement in the program; that is, the debugger checks the value of `idx` after each program statement executes, and it stops program execution if the value of `idx` changes.

```
watchpoint idx
The initial value of \\test\print_msg\idx is 0
go
The value of \\test\print_msg\idx has changed from 0 to 9.
Stopped at: \\test\print_msg\28
```

See `Monitoring Memory Ranges` in the online help for information on monitoring addresses with the `watchpoint` command.

Setting Traces

Traces are useful for monitoring the flow of a program. After setting a trace, for example, you can see when a particular function is called or when certain statements are executed.

By default, setting a trace will display every statement, as it is executed, in the debugger output area. However, you can also set traces that stop execution or that execute debugger commands.

Several levels of trace granularity are available:

- At every procedure entry
- At every procedure exit
- At the entry *and* exit of every procedure
- Every statement
- Every assembly instruction

Traces can also be restricted to certain blocks. The debugger allows you to narrow traces to a particular file, C function or C++ object. See Chapter 7 for an explanation of blocks.

Traces on procedure entry or exit can take a significant amount of time for the debugger to execute. Eliminating entry/exit granularity causes the debugger to run faster, but target program execution may be much slower.

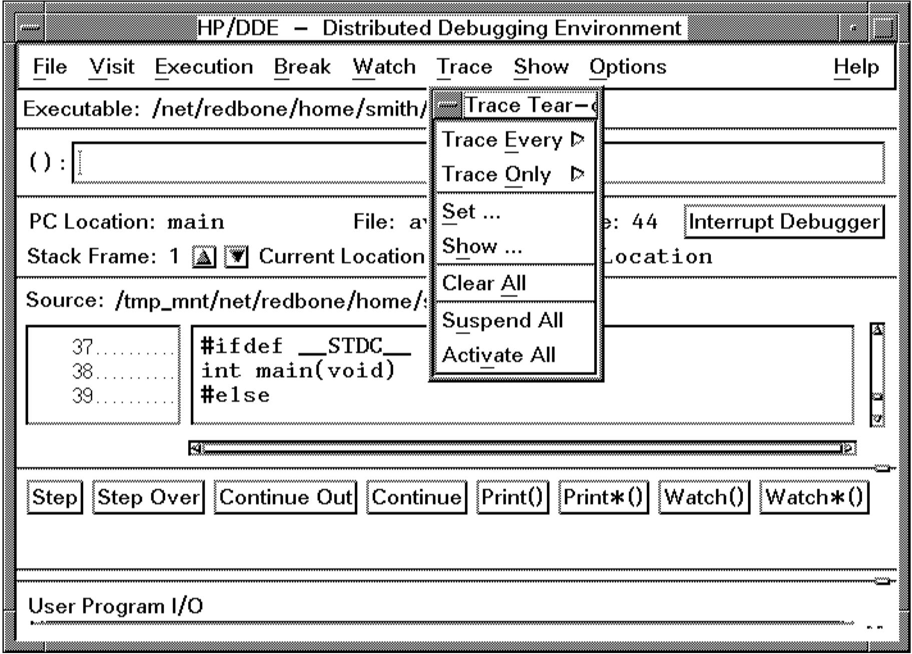


Figure 3-6. The Trace Menu

Using the Trace Menu

A variety of trace commands are available from the `Trace` menu. Choose the appropriate option under `Trace:Trace Every` to enable tracing.

As Figure 3-6 shows, the `Trace` menu is a Tear-off Menu. When you click on the dashed line at the top of the `Trace` menu, the menu is displayed in its own window. The menu persists so that you can invoke commands from it without having to redisplay it. You can also move the menu to a convenient place in your workspace.

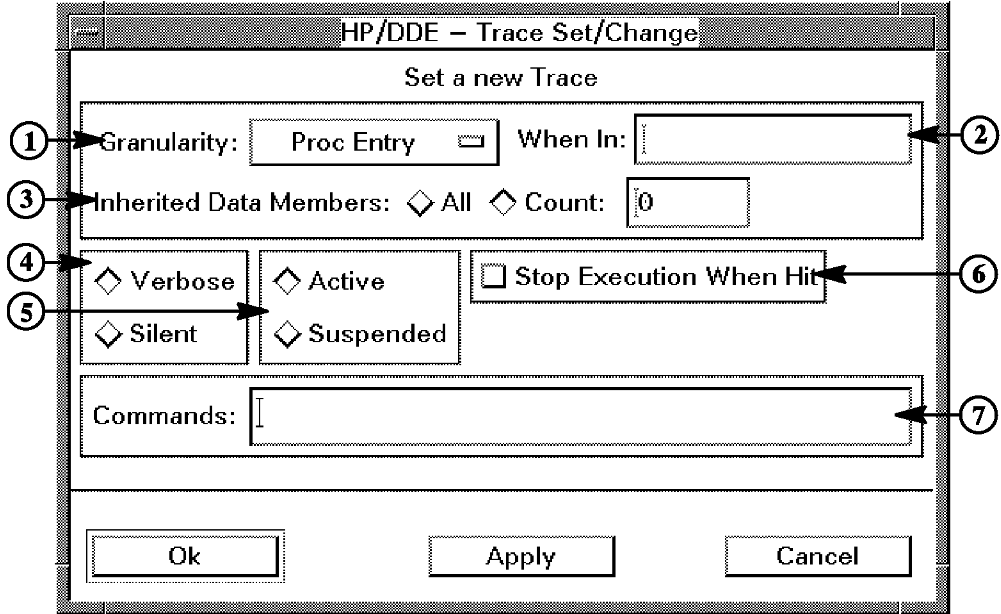


Figure 3-7. The Trace Set/Change Dialog Box

Using the Trace Set/Change Dialog Box

The Trace Set/Change dialog box allows you to specify all aspects of a trace. Invoke it by selecting `Trace:Set`. The dialog box in Figure 3-7 appears.

From the Trace Set/Change dialog box, you can:

1. Select the granularity (how often the trace is triggered).
2. Restrict a trace to be active only in a specified block.
3. Specify how many levels of inherited data members should be included (for C++ programs).
4. Control whether or not messages appear when a trace occurs.
5. Temporarily disable or re-activate traces.
6. Stop target program execution when a trace occurs.
7. Specify debugger commands to execute when a trace occurs.

To see a listing of traces, select `Trace:Show`.

Using the trace Command

The following example shows the result of invoking default tracing on the sample program `average`:

```
trace
go
Trace at: \\average\print_average\25
Trace at: \\average\print_average\30
Trace at: \\average\sum\12
.
.
.
Trace at: \\average\print_average\35
Break at: \\average\main\46
Trace at: \\average\main\46
```

In addition, you can use the `trace` command to:

- Display only selected statements, instructions, routine entry points, and routine exit points as they execute
- Stop execution following each trace event
- Suppress the display of trace locations
- Execute a command list after each trace event

See the online command reference for more information on the `trace` command and its options.

Setting Intercepts

Intercepts are like breakpoints that are set on signals and other events. For example, when a signal event occurs, target program execution stops, and a message is displayed indicating which signal was generated. Execution stops before the signal is delivered to the target program.

By default, the debugger sets intercepts all HP-UX signals. It also sets, then suspends, intercepts on image loading and unloading, certain thread events, and exit events. In C++ programs, you can also set intercepts on `catch` and `throw` events.

Using the Intercepts Dialog Box

The Intercepts dialog box allows you to view and modify intercepts. Invoke it by selecting `Execution:Signals/Intercepts`. The dialog box in Figure 3-8 appears.

From the Intercepts dialog box, you can:

1. View the current status of all available intercepts.

   This list shows all the attributes of each intercept, and also shows the monitor number and the number of times the intercept has been received.

   The monitor number does *not* correspond to the HP-UX signal number. See Appendix C for more information on signal numbers.

2. Temporarily disable or re-activate selected intercepts.
3. Control whether or not messages appear when an intercept occurs.
4. Temporarily disable or re-activate all intercepts.
5. Specify how many times an intercept should be ignored before the debugger handles it.
6. Specify debugger commands to execute when an intercept occurs.

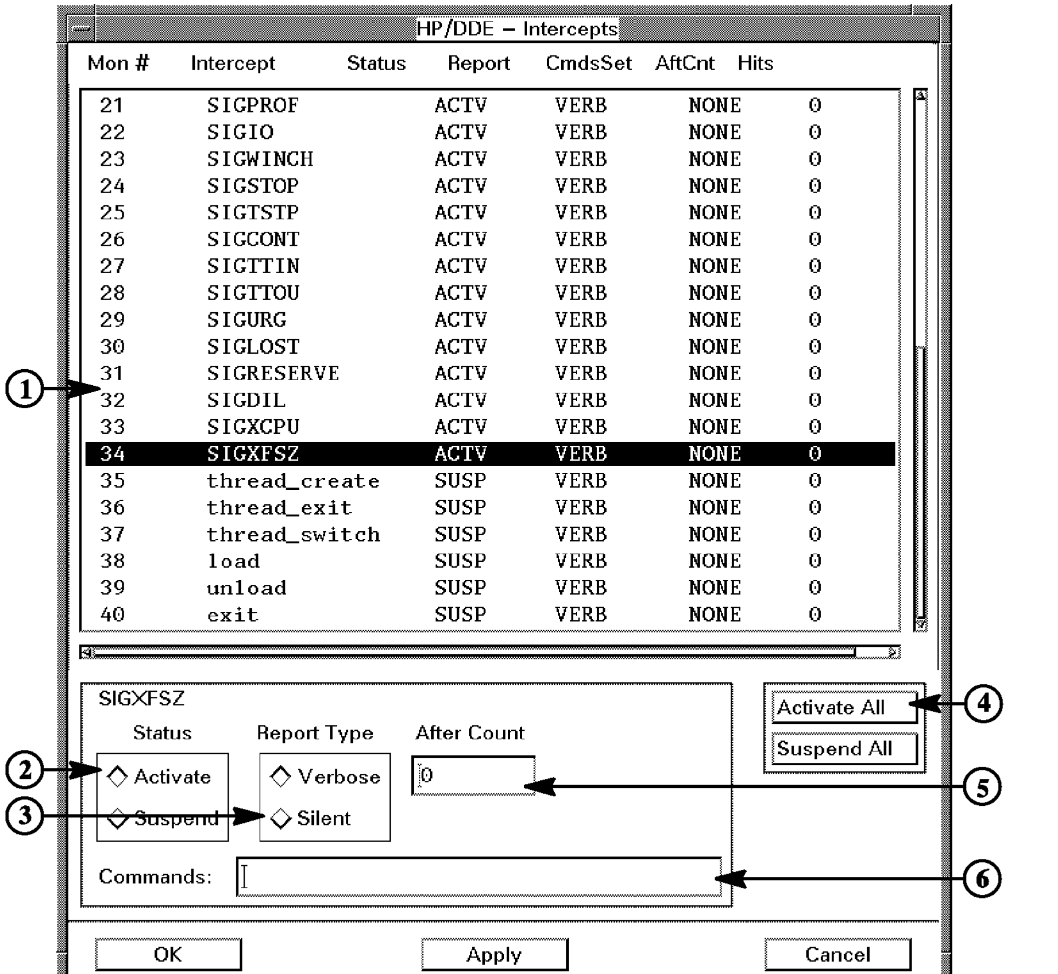


Figure 3-8. The Intercepts Dialog Box

Using the intercept Command

You can use the `intercept` command to create and control intercepts.

In addition, you can use:

- `list intercepts` to show all intercept requests and the monitor number for each intercept.
- `activate intercepts` to restore suspended intercepts.
- `suspend intercepts` to suspend active intercepts.
- `delete intercepts` to remove intercepts.

See the online command reference for more information on these commands.

4

Viewing and Manipulating Target Program Data

This chapter describes how to examine and change various types of target program data. Topics include:

- Examining variables and expressions
- Examinging registers

For more information on the debugger commands mentioned in this chapter see the online command reference. You can also invoke help on a particular command by entering `help` *command_name* in the debugger command input box.

The online help also contains more information on using the debugger's graphical user interface.

Examining Variables and Expressions

Variables are evaluated in the scope of the current location, as indicated in the `Current Location:` line above the source file display. Usually the current location follows the PC (program counter) location, so variables are evaluated in the environment where your program is executing. For example, when you are single-stepping through your source, the current location is in the procedure you are stepping through.

If you want to evaluate a variable in the scope of another function on the current call stack, use the (▲) and (▼) buttons next to the `Stack Frame:` label in the current location line. (Or choose `Show:Stack` and choose a stack frame in the Stack View dialog box.) This sets the current location to the specified function in the call stack.

To evaluate in the scope of a function that is not in the current call stack, enter the function name in the `( ):` input box and choose `Visit:Procedure()` to set the current location to that function.

Finally, you can always specify the variable fully with the appropriate DDE syntax (see Chapter 7). This syntax overrides the current location.

The PC arrow points to the line that will be executed next. (The PC location arrow changes to a broken variant to indicate when the current point of execution is either at another source statement on the same line or at an instruction beyond the statement's first instruction.) When the arrow points to an assignment statement, the assignment has not yet been executed. To see the result of an assignment statement, step past it (or step over it if it calls a function).

When reporting a value, the debugger uses qualifiers in the form:

`\\`*module_name*`[\`*routine_name*`[\`*subroutine_name*`]...]\`*object_name*

For example, the following `print` command output indicates that `x` is local to the routine `sum` of the module `test`:

```
print x
\\test\sum\x:3
```

You can eliminate qualifiers by entering the command `property qual_max 0`.

For more information on qualifiers, see "Using Qualified Names" in Chapter 7.

Using Command Buttons

There are two command buttons that allow you to display values of variables. The values are printed in the debugger output area. The following buttons are located below the source file display area:

Print() — Print the value of the contents of the `( ):` input box.

You can also evaluate expressions and assign values to expressions. For example, in C syntax, if `n/2` is in the input box, the result of `n` divided by `2` is printed. If `n = 4` is in the input box, the value `4` is assigned to the variable `n`.

If the expression, `ptr`, is a pointer, printing `ptr` displays the address of the variable pointed to. Printing `*ptr` displays the value pointed to.

Print*() — Print the value pointed to by the pointer in the `( ):` input box.

For example, if `ptr` (declared as `int *ptr;`) is in the input box, the integer pointed to by `ptr` is displayed.

You can double-click or highlight the variable name or expression in the program source to copy it to the `( ):` input box.

If the current location points to the procedure containing the variable, you can use the name of the variable without any qualifiers. If not, specify it according to the rules defined in “Using Qualified Names” in Chapter 7.

Using the Mouse

Another method for examining variables and expressions is to use the mouse as follows:

1. Position the cursor over an expression in the source file display area.
2. Click the right mouse button. The `Source Actions` popup menu appears.
3. Select either `Print` or `Print*`. Note that the expression under the cursor appears as an argument to these commands.

Using the Data Value Menu

As shown in Figure 4-1, a number of `print` commands are available from the `Data Value` menu. Invoke it from `Show:Data Value`.

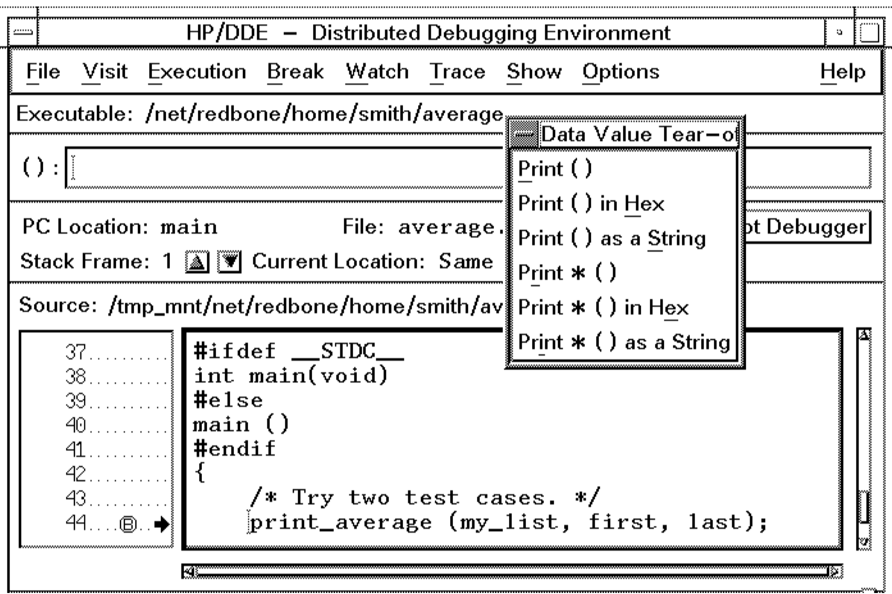


Figure 4-1. The Data Value Tear-Off Menu

The `Data Value` menu is a Tear-off Menu. When you click on the dashed line at the top, the menu is displayed in its own window The menu persists so that you can invoke commands from it without having to redisplay it. You can also move the menu to a convenient place in your workspace.

Using Debugger Commands

If you prefer to use debugger commands, the following list shows some common usages:

`print x`	Displays the value of `x`.
`print y + z`	Displays the sum of `y` plus `z`.

`print x = y + z`	Assigns the value of `y + z` to `x` (using C syntax). Displays the result in the debugger output area.
`set x = y + z`	Assigns the value of `y + z` to `x` (using C syntax). Does not display the result in the debugger output area.
`declare int idx`	Creates the temporary variable `idx` of type `int` (using C syntax).
`list declares`	Show all user-defined (temporary) variables.
`args`	Show the values of arguments of the current routine

The next sections give you some idea of the the versatility and flexibility of these commands. They describe how to use debugger commands to examine arrays, pointers, linked lists, and buffers.

The online command reference gives more examples and describes all the options for each command.

Examining Arrays

To print an entire array, enter the `print` command and specify the array name without a subscript. For example, suppose the array `list` is declared as follows:

```
static int list[5] = {3,4,2,0,5};
```

The following command prints the array `list`:

```
print list
\\parray\list: (array)
\\parray\list[0]: 3
\\parray\list[1]: 4
\\parray\list[2]: 2
\\parray\list[3]: 0
\\parray\list[4]: 5
```

To print a portion of an array, enter the range as the subscript. Specify a range in the form *element*..*element* for C, C++, and Pascal, and as *element*:*element* for FORTRAN. For example, the following command prints `list[1]` through `list[3]` in the C language array `list`:

```
print list[1..3]
\\parray\list[1]: 4
\\parray\list[2]: 2
\\parray\list[3]: 0
```

To specify a limit on the number of elements the `print` command displays, use the `property array_dim_max` command. For example, `property array_dim_max 10` sets the limit at 10.

Examining Objects Referenced by Pointers

To examine an object referenced by a pointer, enter the `print` command and either dereference the pointer using language-specific syntax or use the `print` command's `-indirect` option.

For example, assume that C is the current language and that `int_ptr` contains the address of the variable `num`, whose value is 7. You could print the value pointed to by `int_ptr` using C language syntax for dereferencing pointers, as the following example illustrates:

```
print int_ptr
\\test_program\main\int_ptr: 7B03A558
print *int_ptr
*\\test_program\main\int_ptr: 7
```

In languages that support special interpretation of pointers to characters:

- Printing the value of a pointer will also print the string that is pointed to.
- Printing the value of a dereferenced pointer will print a single character.

For example, assume that C is the current language and that `char_ptr` points to the first element in the string `success`:

```
print char_ptr
\\test_program\main\char_ptr: 7B03A541
*\\test_program\main\char_ptr: "success"
print *char_ptr
*\\test_program\main\char_ptr: 's'
```

Instead of using language-specific syntax, you can use the `-indirect` option to the `print` command as follows:

`-indirect all`	Follow all pointers encountered and show the value of the object pointed to.
`-indirect` *count*	Follow pointers no further than *count* levels.
`-indirect`	Follow pointers one level.

Examining Linked Lists

Use the `print` command's `-indirect` option to examine linked lists.

Enter `-indirect` *count* to print a specific number of records in a linked list. For example, the following command prints the record pointed to by `first_item`, as well as the next record in the linked list:

```
print first_item -indirect 2
\\main\first_item: 00088000
*\\main\first_item: (record)
\\main\first_item->item_text: "Class"
\\main\first_item->font: "<F21>"
\\main\first_item->next: 00088C00
*\\main\first_item->next: (record)
\\main\first_item->next->item_text: "Type"
\\main\first_item->next->font: "<F21>"
\\main\first_item->next->next: 00089800
```

To print an entire linked list you could specify `-indirect all`. But if the linked list is large, the output may become unwieldy, since every link to each record is shown.

To print all the records in a large linked list, you can make the debugger walk through the list and print each record separately. Use the `while` command, as shown in the following example:

```
set item = first_item; \
while item != 0 -loop [print item -indirect 1; \
set item = item->next] >tmp.out
```

The `set` command assigns the value of `first_item` to the pointer `item`. The `while` command then executes the commands between the brackets as long as `item` points to an address. The commands between the brackets print the current record and increment `item`. All output is redirected to the file `tmp.out`.

For more information on grouping commands to perform useful tasks, see "Combining Debugger Commands Using Action Lists" in Chapter 5.

Examining Buffers

Use the `describe`, `print`, and `dump` commands to examine buffers.

The `describe` command with the `-va` option is useful for finding the virtual address of an element in a buffer. In the following example, the `describe` command is used to display the virtual address of an element in a character buffer (called `buff`) from a C language program:

```
describe buff[60] -va
4000110C
```

The `print` command can also display the address of an element. However, it also displays the contents of the buffer starting from the specified element. Notice the use of the `&` operator in the following example:

```
print &buff[60]
4000110C
*: "a rather serious crime\nTo marry two wives at a time."
```

Use `print` with the following syntax to display the contents of a single element in the buffer:

```
print buff[60]
*: 'a'
```

For large buffers, the `dump` command is useful because you can use the `-from` and `-to` options to specify address ranges. Also, `dump` takes a number of options that allow you to format the output.

The following example shows a sequence of commands that determine an address range in a buffer. Then, the `dump` command prints the data in the address range:

```
describe buff[60] -va
4000110C
describe buff[92] -va
4000112A
dump -from 4000110C -to 4000112A -char -bits 256
"a rather serious crime\nTo marry "
```

Notice that the `-char` option causes the output to be displayed as characters. The `-bits 256` option specifies that the output should be formatted in units of 256 bits or 32 characters.

See the online command reference for more information on the `describe`, `print`, and `dump` commands.

Examining Registers

Select one of the `Show:Registers` menu items to display a dialog box that shows the contents of a group of hardware registers. Figure 4-2 shows a dialog box displaying general registers. You can also display floating point and special registers.

From the Show Registers dialog box, you can:

1. View register contents.

   Register values that change during execution are highlighted.

2. Specify how the the register values are updated (similar to setting a watchpoint's granularity).

   You can also temporarily suspend updating, which allows the target program to run faster. When register tracing is activated and set to anything other than `Every Stop`, the debugger creates an implicit watchpoint to check the register values at the appropriate granularity.

3. Change from the default hexadecimal display format.

   The floating point register dialog box allows you to specify whether the registers are displayed as two 4-byte values (single precision) or one 8-byte value (double precision).

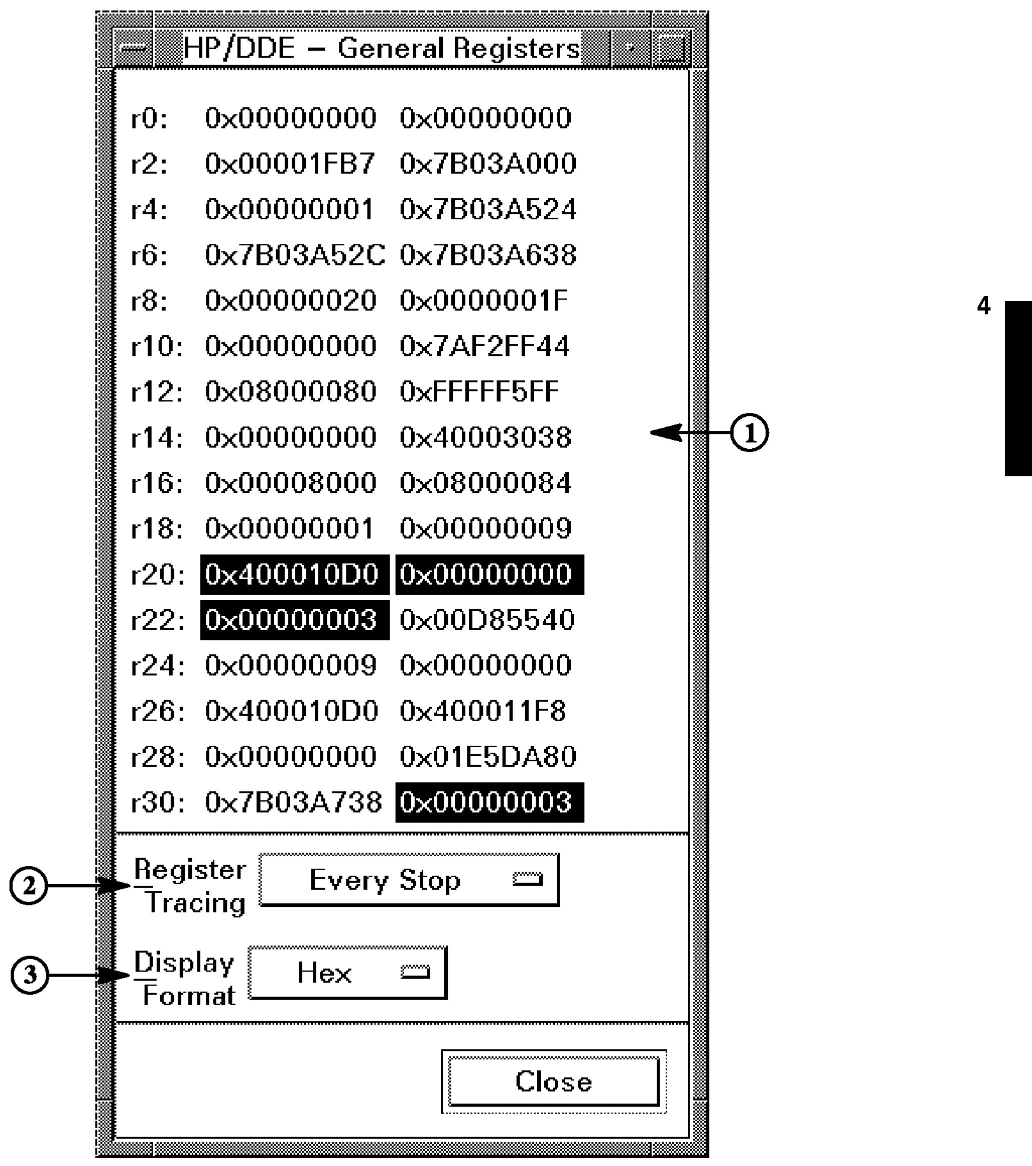


Figure 4-2. Show Registers Dialog Box (General)

Using Register Commands

Use the following commands to display the contents of groups of registers in the debugger output area:

`regs`	General registers
`cregs`	Control registers
`fregs`	Double-precision floating-point registers
`fsregs`	Single-precision floating-point registers
`fdregs`	Double-precision floating-point registers
`sregs`	Space registers

The commands listed above are aliases for the `dump` command. They are defined in the target manager startup file (described in "Using Startup Command Files" in Chapter 6), which is in:

```
/opt/langtools/dde/tgt/tgt_hpux_pa.startup
```

(See "Solaris SPARC Target Manager" in Appendix C for information on the register dumping macros on Solaris systems.)

To display a single register, use `dump -from` *address*. Specify a register name for *address*. For example:

```
dump -from r8
r8: 40006110
```

See Appendix C for more information on register names.

5

Using Debugger Commands

This chapter describes the conventions you need to follow when you issue commands from the debugger command input box, in startup files, or as arguments to the `dde` command `-do` option. Topics include:

- Abbreviating debugger commands
- Entering multiple debugger commands on one line
- Using command lists
- Continuing commands on the next line
- Resolving syntax conflicts
- Editing the command line
- Using the command history facility
- Recording command sequences for later playback
- Invoking shell commands from the debugger
- Redirecting input and output
- Using reserved identifiers and special macros
- Creating alias and define macros
- Combining debugger commands using action lists

For more information on the debugger commands mentioned in this chapter see the online command reference. You can also invoke help on a particular command by entering `help` *command_name* in the debugger command input box.

See "Using Startup Command Files" in Chapter 6 for more information on startup files.

Abbreviating Debugger Commands

When you enter debugger commands, you can abbreviate any debugger command or command argument to its first three characters.

For example, `des total` is equivalent to `describe total` and `bre 17` is equivalent to `breakpoint 17`.

Entering Multiple Debugger Commands on One Line

When you enter a single debugger command, terminate it with an end-of-line (EOL), which is usually done by pressing Return. When you enter multiple commands on one line, separate each command with a semicolon (;).

The `alias`, `debug`, `define`, `target command`, `property flags`, and `shell` commands require the rest of the line as an argument. They only recognize an EOL, not a semicolon, as a terminator.

However, you can combine these commands with others on the command line if you enclose them in square brackets `[]`, braces `{}`, or parentheses `()`. For example, you could create an alias and then invoke it on the same command line:

```
[alias ph print -hex]; ph foo
```

Using Command Lists

A command list is one or more debugger commands with arguments, separated by semicolons and enclosed in square brackets `[]`, braces `{}`, or parentheses `()`.

For example, you can direct the output of several commands into a file:

```
[list breakpoints; list defines; list aliases] >save
```

Notice that the output redirection operator must have a space before it but no space after it.

Continuing Commands on the Next Line

The continuation character, a backslash (\), lets you continue a command past the end of the line. The backslash must follow a space and must be the last character on the line. When you press Return, the debugger echoes the backslash after the `Debugger Input:` prompt.

Some debugger commands take a list of items separated by commas as an argument. If you enter one of these commands and press Return when a comma is the last non-blank character on the line, the debugger automatically continues the line.

The debugger also continues a line that contains unbalanced brackets, braces, or parentheses. Note, however, that the debugger balances only delimiters of the same type as the outermost delimiter.

When unbalanced brackets, braces, or parentheses are entered in menus or in dialog boxes, the debugger prompts for completion on the command line.

Resolving Syntax Conflicts

Some debugger commands, such as the `print` command, take programming language expressions as arguments. This may cause a problem if the programming language's syntax conflicts with the debugger's command language syntax.

For example, a conflict occurs if you use a valid option to a debugger command, such as `-hex`, within a program expression.

You can usually resolve syntax conflicts by enclosing expressions that are used as arguments in parentheses. Consider the following `print` commands:

`print (abc -hex)`	Prints the value of `abc` minus the value of `hex`.
`print abc -hex`	Prints the value of `abc` in hexadecimal.

Resolving Case Sensitivity

Debugger commands and options are not case sensitive. Arguments to commands are case sensitive. Therefore, the debugger considers the command

```
debug -output temp my_prog
```

to be the same as

```
DEBUG -OUTPUT temp my_prog
```

but different from

```
debug -output temp MY_PROG
```

Programming language expressions are case sensitive if the current language is case sensitive. If your current language is not case sensitive and you want to refer to another part of the program that is case sensitive, you must first reset your language using the `property language` command. See the online command reference for more information on the `property language` command.

Commands that have a `-language` option eliminate the need to reset your language with the `property language` command. The `-language` option allows the command to use a language other than the current one to parse or evaluate an expression. The `print` and `set` commands, for example, can be invoked with the `-language` option.

Editing the Command Line

Choose `Options:Command Input Mode` to select an edit mode for the debugger command input box. The two choices are:

- `Use Ksh Mode`. This invokes a KornShell-like edit mode, which emulates either `vi` or `emacs` depending on the value of the `EDITOR` environment variable.
- `Use Motif Mode`. This invokes a simplified Motif edit mode, in which basic editing keys (Backspace, arrows) are active.

Using the Command History Facility

When the debugger is in Motif edit mode (see "Editing the Command Line") a history buffer appears above the `Debugger Input` box. Click on an item in the history buffer to copy it into the input box, or double-click on it to re-execute the command.

When the debugger is in Ksh edit mode, you can use key commands to scroll through the command history. For example:

- If `EDITOR` is set to `emacs`, CTRL-P accesses the previous command and CTRL-N accesses the next command.
- If `EDITOR` is set to `vi` and you are in `vi` command mode, k accesses the previous command and j accesses the next command.

Command history is stored in a file called `.ddehist` in the directory where the debugger is started, and is preserved between debugging sessions.

Recording Command Sequences for Later Playback

The `property record` command is useful if you want to save debugger command sequences for reuse at another time. It records all commands whether they are invoked from menus, command buttons, or from the debugger command input box. The syntax is:

```
property record {pathname [-append | -replace]
                | -on
                | -off
                }
```

The *pathname* argument specifies the command file where the command sequences will be stored. You can add or delete entries in the command file with a text editor.

The `-on` and `-off` arguments start and stop recording. After specifying *pathname* when you first start recording, you need not specify it every time you start and stop. Only enter *pathname* again when you want to change to a different command file.

The `-append` and `-replace` arguments specify whether to add to, or to overwrite an existing command file.

You can invoke the commands in a command file by using the input operator (`<`*pathname*) or the `input` command (`input -from` *pathname*). See "Redirecting Input and Output" for more information.

For more information about the `property record` command, see the online command reference.

Invoking Shell Commands from the Debugger

To invoke shell commands from the debugger command line, use the `shell` command. For example, the following command compiles the test program, `average.c`, for debugging:

```
shell cc -g average.c
```

The value of the `SHELL` environment variable determines which shell is invoked.

Note Do not enter the `shell` command without an argument or invoke a command that expects standard input. The command will fail when it attempts to read from standard input

For more information on the `shell` command, see the online command reference.

Redirecting Input and Output

You can direct the debugger to read its input from a file with an I/O operator or with the `input` command. File input is aborted if the debugger encounters an error while reading from an input file. Use the `property abort` command to have the debugger continue reading from the file when it encounters an error.

You can use the `property record` command to create an input file of debugger commands. See "Recording Command Sequences for Later Playback" for more information about `property record`.

From the debugger command input box, you can direct the debugger to read its input from a file using an I/O operator as follows:

`<`*pathname* Read and execute debugger commands from the file *pathname*.

You can also use I/O operators to read command input from a file and redirect output to a file as follows:

```
<pathname.in >pathname.out
print abc >pathname.out
```

You cannot use parameters with input files.

You can redirect debugger command output by issuing a command followed by I/O operators (similar to those found in most shells) from the command entry line. Output remains redirected until the command has been completed.

cmd >*pathname*	Redirect standard output to *pathname*.
cmd >>*pathname*	Redirect standard output and append it to *pathname*.
cmd 2>*pathname*	Redirect standard error output to *pathname* (Korn and Bourne shell notation).
cmd 2>>*pathname*	Redirect standard error output and append it to *pathname*.
cmd >>?*pathname*	Redirect standard error output and append it to *pathname*.

Note A space must separate the command from the I/O operator. Spaces are not allowed between I/O operators or between an I/O operator and *pathname*.

You can also group the output of several commands by using brackets. For example, the command

[*cmd1*; *cmd2*; *cmd3*] >*pathname*

redirects the output from all three commands to *pathname*.

Creating Alias and Define Macros

You can use the `alias` and `define` commands to create two types of **macros**:

- An alias macro serves as a synonym for one or more debugger commands. The debugger expands alias macros only when the macro is at the beginning of a command line, or when the macro follows a semicolon in a line of commands.
- A define macro serves as a synonym for any string of text, including debugger commands. The debugger always expands a define macro no matter what position the macro occupies in a command line.

For example, consider the following series of commands:

```
alias show print
alias ali_X total
define def_X total
show ali_X
"ali_X" not found in current environment.
show def_X
total: 20
```

The `alias` commands create `show` as a synonym for the `print` command, and `ali_X` as a synonym for the variable `total`. The `define` command also creates a synonym for `total`, but gives it the name `def_X`.

The debugger does not expand `ali_X` in the command `show ali_X` since `ali_X` does not appear at the beginning of the line and it does not follow a semicolon. However, the debugger does expand `def_X` since define macros can appear anywhere on the command line. Also, since the alias `show` appears first in both of the last two command lines, the debugger expands `show` in both cases.

The debugger expands macros from left to right in the command line. Upon finding a macro with arguments, the debugger inserts the arguments into the macro definition. Then the debugger rescans the command line, starting from where it found the macro.

You can define macros in terms of other macros and nest them. Note that macros cannot be recursive.

Macro names may begin with a letter, a grave (‘) accent, a hyphen (`-`) or an underscore (`_`). Macros accept parameters and are defined like C language macros. See the descriptions of the `alias` and `define` commands in the online command reference for syntax details.

You must delimit a macro name in a command string with characters not allowed in a macro name, such as the space character. For example:

`define X len`	Defines macro `X` as `len`.
`print X`	Expands to `print len`.
`print myX`	Does not expand because the debugger does not treat the letter `y` as a delimiter.

Start a macro name with a grave accent (‘) to form a macro that can concatenate its text with other strings. For example:

`define ‘X len`	Defines macro `‘X` as `len`.
`print ‘X`	Expands to `print len`.
`print my‘X`	Expands to `print mylen`.

Like arguments to debugger commands, macros are case-sensitive. You can list alias macros using the `list aliases` command; use `list defines` to list define macros.

You can specify any string as the value for an actual parameter. The debugger recognizes any string within matching brackets `[]`, braces ‘{}’’, or parentheses `()` as a single argument. However, bracket pairs within the string are treated in a special manner, and quotation marks are treated as part of the string. The outermost brackets (within the parentheses that delimit the argument) must balance; brackets, characters, and punctuation marks within the outermost brackets are considered part of the argument.

A period (`.`) preceding a command string prevents the debugger from trying to expand the string in an alias or define macro. For example, using the period can prevent infinite looping in the following alias:

```
alias s[tep] .step -ignore
```

Without the period, when the debugger encounters the `step` command string, it would continue to apply the alias definition indefinitely.

Using Reserved Identifiers and Special Macros

The debugger reserves certain identifier and macro names for special purposes. Table 5-1 contains a complete list of reserved identifiers and special macros. You can use these reserved identifiers and special macros for several operations, including the following:

- Specifying commands that the debugger is to execute in specific situations, such as after the invocation of the target program or after a fault occurs.

  For example, after loading a target program, the debugger checks the `'after_debug` macro and executes any commands assigned to it.

  By default, `'after_debug` is undefined. Use the `alias` command to define it. For example, the following sets the `'after_debug` macro to delete the intercept for the `SIGVTALRM` signal after a target program is loaded:

  ```
  alias 'after_debug delete intercept signal SIGVTALRM
  ```

- Identifying an environment other than the current environment (the target program procedure which the debugger refers to when evaluating expressions).

  For example, `'env(+`*n* `|` `-`*n*`)` is useful for changing locations on the call/return stack. The following command changes the environment up one frame on the stack:

  ```
  environment 'env(-1)
  ```

- Referring explicitly to a module name in a particular image, a language-defined symbol name, or a variable that you create with the `declare` command.

  For example, if you use the `declare` command to create temporary variables, you can use `'declared` to refer to them. The following command would list all the temporary variables that you defined:

  ```
  list blocks 'declared -full
  ```

Table 5-1. Reserved Identifiers and Special Macros

Reserved Identifiers and Special Macros	Function
`'after_debug`	[1]Specifies debugger commands that the debugger executes after invoking the target program. See the **debug** command entry in the online command reference for details.
`'after_fault`	[1]Specifies debugger commands that the debugger executes if the target program faults.
`'amb`	Identifies an overloaded C++ function. See online help on C++ debugging for details.
`'asm`	[1]Identifies the appropriate assembly language manager for the target machine. Defined in the target manager startup file.
`'command`	[1]Contains the most recently entered command string. A command string can be a single debugger command or several debugger commands separated by semicolons. All characters up to the carriage return will be repeated.
`'cr`	[1]Specifies debugger commands that the debugger executes when you enter Return on an empty command line. By default, `'cr` is set to `'command` which causes the most recently entered command to be invoked.
`'declared`	Specifies the outer block where the debugger stores variables that you create using the `declare` command. You can use `'declared` to refer explicitly to variables created with `declare`.
[1]Indicates that the entry is a macro.	

Table 5-1. Reserved Identifiers and Special Macros (continued)

Reserved Identifiers and Special Macros	Function
`‘env(+`*n* `|` `-`*n*`)`	If you do not specify *n*, `‘env` refers to the current environment (that is, the target program procedure which the debugger refers to when evaluating expressions). If you do specify *n*, `‘env` refers to an environment relative to the current environment. See Chapter 7 for more information.
`‘image(`*name*`)`	Associates a module name with a particular image. This is useful when your program consists of more than one image (that is, when your program uses dynamically loaded libraries) and multiple images use modules with the same name. See “Image Qualified Names” in Chapter 7 for details.
`‘label(`*statement_label*`)`	Uses a label in the source code to identify a line. Any alphanumeric character string that is a valid label in the source language is a valid argument. Often used to specify the location of breakpoints.
`‘long(`*expression*`)`	Converts the value of *expression* to a 32-bit integer type.
`‘main(`*n*`)`	Identifies the frame that is *n* frames away from the oldest frame on the call/return stack. If you don’t specify *n*, `‘main` specifies the oldest frame on the call/return stack. See “Frame Block Qualified Names” in Chapter 7 for details.

5

Table 5-1. Reserved Identifiers and Special Macros (continued)

Reserved Identifiers and Special Macros	Function
`` `predefined( ``*lang_type, obj_type*`)`	Specifies special blocks where the debugger stores names defined by programming languages (*lang_type*) and object type (*obj_type*). You can use `` `predefined `` to refer explicitly to a language-defined symbol name. See Appendix C and Appendix D for information on how to specify *lang_type* and *obj_type*.
`` `run( ``*n*`)`	Identifies the frame that is *n* frames away from the most recent frame on the call/return stack. If you do not specify *n*, `` `run `` specifies the location at which the program stopped. See "Frame Block Qualified Names" in Chapter 7 for details.
`` `short( ``*expression*`)`	Transfers the value of *expression* to a 16-bit integer type.
`` `thread( ``*n*`)`	Specifies a thread according to the target manager-defined identifier, *n*. Use the command `list threads` to display threads and their identifiers.
`` `va( ``*address*`)`	Specifies that *address* is a machine location. `va is useful for printing or changing the contents of a machine location. See "Assembly Level Debugging" in Chapter 8 for details.

Combining Debugger Commands Using Action Lists

An action list is a series of debugger commands associated with a specific breakpoint, watchpoint, intercept, or tracing request. The debugger executes an action list after executing to the breakpoint location, detecting a change in the watched value, receiving an intercept, or encountering a tracing event. (See the `breakpoint`, `watchpoint`, `intercept`, or `trace` command description in the online command reference for more information.)

The following sections describe how to create action lists and how the debugger handles action lists in special situations.

Creating Action Lists

Use the `-do` option to specify an action list for a `breakpoint`, `watchpoint`, `intercept` or `trace` command. For example, consider the following `breakpoint` command:

```
breakpoint 30 -do [args; go]
```

Because of the `-do` option in the preceding example, when execution reaches line 30, the debugger first prints the value of the arguments to the current routine (that is, the debugger executes the `args` command) and then resumes execution (executes `go`). Now consider the following `watchpoint` and `trace` commands:

```
watchpoint p -do [print p/r]
trace -statement -in sum -do [print s]
```

In the preceding example, the `-do` option to the `watchpoint` command causes the debugger to print the value of `p` divided by `r` whenever the value of `p` changes during program execution. The `-do` option to the `trace` command, on the other hand, causes the debugger to print `s` at each statement (as specified by `-statement`) in the routine `sum` (as specified by `-in sum`).

Creating Conditional Action Lists

Use the `if` command to create conditional action lists. For example, the command

```
breakpoint 18 -do [if i > 5 -then [print s] -else [go]]
```

causes the debugger to invoke `print s` whenever `i` is greater than 5 at line 18. If `i` is 5 or less, the debugger invokes `go`.

You can also add a `return` command to an action list if you want the debugger to exit from the action list under certain conditions. For example, the following `return` command

```
breakpoint 18 -do [if i>10 -then [return]; \
    -else [set list[i] += 1; go]]
```

causes the debugger to exit the action list if `i` is greater than 10 at line 18. If `i` is less than 10, the debugger increments array element `list[i]` and resumes execution.

Understanding Action List Execution in Special Circumstances

The following sections describe how the debugger handles action lists in special situations, such as when an error occurs in an action list or when more than one action list is eligible for execution at the same time.

Errors in Action Lists

By default, if an error occurs in an action list, the debugger aborts the action list. Use the `property abort` command to change the default and direct the debugger to continue executing the action list even if it encounters an error. Refer to the description of the `property abort` command in the online command reference.

Execution of Multiple Action Lists

Because you can set action lists on more than one kind of program monitoring request, you can have multiple action lists eligible for execution at the same time. For example, the debugger may detect a change in a watched value at the same time as it encounters a breakpoint.

The debugger executes in an arbitrary order the action lists that are eligible for execution at the same time. If two or more such action lists have an unrestricted `go` command (a `go` command with no arguments), the debugger executes all action lists up to the first unrestricted `go` command it encounters, executes that `go` command, and then issues a warning that it discarded other `go` commands. This behavior prevents you from inadvertently losing control of the program when two action lists execute concurrently.

Action List Execution Following an Interactive step Command

If program execution stops following an interactive `step` command and the debugger executes one or more action lists, the debugger ignores all requests to execute the program from those action lists and issues a warning. This behavior prevents the program from continuing to execute while you are trying to step through it.

Placing step and go Commands in Action Lists

The debugger executes all `step` and `go` commands in the order they appear in the action list, except unrestricted `go` commands as described in "Execution of Multiple Action Lists". You can place `step` or `go` commands anywhere in an action list. However, it is usually best to them at the end, as in the following example:

`breakpoint 45 -do [`*some_command*`;` *another_command*`; go]`

Otherwise, you may find that execution of the action list generates unanticipated results. The commands following a `go` or a `step` command might execute at program locations you do not anticipate.

In addition, commands may execute in an order that you do not expect. This situation can occur when control returns to the debugger from the target program. The debugger executes any new action lists applying to the new location, and then executes any commands remaining from earlier action lists.

Generally, you can avoid unexpected results by placing a `step` or `go` command only at the end of an action list.

6

Customizing the Debugger

This chapter describes methods for customizing the debugger to suit your own preferences and needs. Topics include:

- Customizations available from the debugger's `Options` menu
- Using startup command files
- Emulating the `xdb` or `dbx` debuggers

The online help contains information about customizing the debugger's graphical user interface. Look for `Customizing HP/DDE` in the `User Interface` section.

6

Using the Options Menu

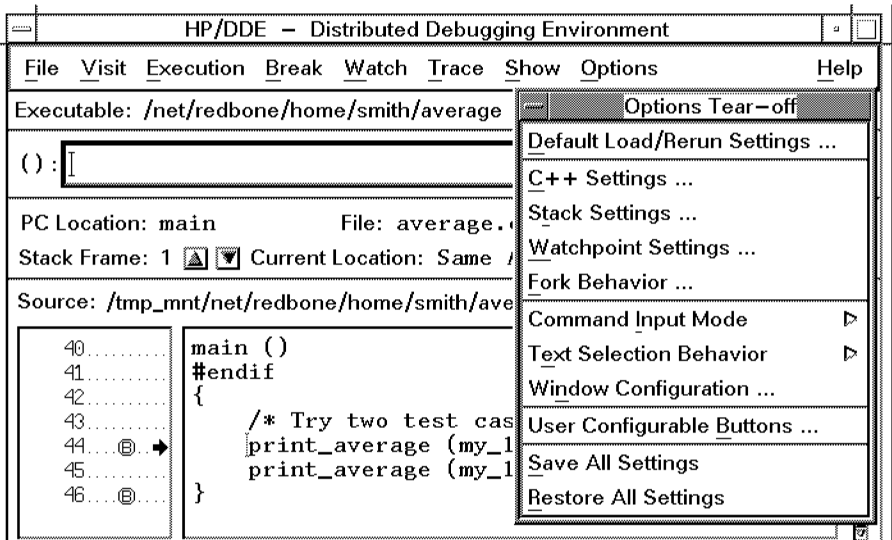


Figure 6-1. The Options Menu

A variety of customizations are available from the debugger's `Options` menu, shown in Figure 6-1. They include:

- Specifying the default program environment.
- Modifying breakpoint, watchpoint, and trace behavior for C++ programs.
- Changing the format and behavior of the stack display.
- Changing the format and behavior of the watchpoint display.
- Specifying the debugger's behavior when `fork()` calls are encountered.
- Selecting the edit mode used in the `Debugger Input` command box.
- Selecting the language-sensitive text selection behavior.
- Displaying sections of the debugger's main window as separate windows.
- Changing the command buttons on the front panel and the pop-up menus in the source and assembly displays.
- Saving and restoring all settings.

To get more help on these customizations, hold down the left mouse button on an item in the `Options` menu, then press F1.

Using Startup Command Files

When you invoke the debugger, it executes three startup command files. Startup command files contain commands that define command aliases and commands that customize the debugging environment.

The startup files and their order of execution are:

1. **User interface manager startup file.** Contains default definitions for macros and for debugging environment settings. You can override these defaults definitions using your personal startup file (see "Using a Personal Startup File to Customize the Debugger") . To find out the name and location of the user interface startup file for each user interface manager, see Appendix E.

Note If you specify `-do` on the `dde` command line, the debugger executes the `-do` option's command list after executing the user interface manager startup file.

2. **Personal startup file.** Contains commands that tailor the debugging environment to your personal tastes and that define personal command aliases. "Using a Personal Startup File to Customize the Debugger" describes personal startup files and how to create them.
3. **Target manager startup file.** Contains macro definitions specific to the target machine. For example, it contains a definition of the macro `regs`, which displays the machine registers.

   The debugger executes the target manager startup file after invoking or attaching to a target program, but before activating the target program. The debugger re-executes the target manager startup file whenever you issue the `debug` command. See Appendix C for more information on the target managers.

Using a Personal Startup File to Customize the Debugger

A personal startup file contains debugger commands that you want executed each time you invoke the debugger. To create a personal startup file, create a file called `.dderc` in either your current working directory or in your `$HOME` directory. In `.dderc`, place any commands that you want to execute at debugger startup.

When you invoke the debugger, it searches for `.dderc` in the current working directory. If the debugger does not find `.dderc` in your current working directory, it then searches your `$HOME` directory.

A personal startup file typically contains commands that:

- Set debugger properties (such as the maximum number of array elements that the debugger will print) to values that you use frequently (`property array_dim_max`)
- Define command aliases, which let you more easily enter commands or a sequence of commands that you use frequently (`alias`)

In addition, you can define a special alias, called `‘after_debug`. This alias can contain commands that you want the debugger to execute after it invokes the target program; for example, `‘after_debug` might set a breakpoint or invoke a command file containing commands specific to the target program. To use `‘after_debug`, define it with an `alias` command in your personal startup file. For example, the `alias` command

```
alias ‘after_debug breakpoint sum; go
```

causes the debugger to set a breakpoint at the first executable statement in the procedure `sum` and then to execute the `go` command.

A Sample Personal Startup File

The directory `/opt/langtools/dde/contrib` contains sample startup files that you can use as models for your own `.dderc` file. (On Solaris systems, the directory is `/opt/softbench/dde/dde/contrib`.) The file names and contents are:

`dderc_abbrev`	Shortcuts for HP/DDE commands
`dderc_hints`	Examples of specialized macros
`dderc_xdb`	Aliases that map HP/DDE commands to `xdb` equivalents
`dbx_macros`	Macros that simulate `dbx` syntax
`dderc_threads`	Modifications that are useful when debugging multi-threaded applications

Figure 6-2 shows the types of commands that are typically found in an HP/DDE personal startup file.

```
property array_dim_max 10      # Print a maximum of 10
                               # elements of an array.

property echo -graphic         # In the debugger output area,
                               # echo commands entered
                               # using dialog boxes, pop-up
                               # menus, or specially
                               # defined keys.

alias so step -over            # Use "so" on an empty
                               # line to step execution,
                               # stepping over called
                               # routines.

alias s step                   # Use "s" to step
                               # execution, stepping into
                               # called routines.

alias gr go -return            # Use "gr" to return to
                               # the calling procedure.

alias ‘after_debug delete intercept signal SIGVTALRM
                               # When starting up the
                               # target program, delete
                               # the intercept for the
                               # SIGVTALRM signal.

alias ls sh ls                 # Use "ls" to list directory.

                               # (continued)
```

Figure 6-2. A Sample Personal Startup File

```
# The following dump command aliases vary the format
# (hexadecimal, decimal, or character) used to display
# memory. "addr" is a memory address or mnemonic  (such
# as PC) that you must supply as an argument when you
# invoke one of the aliases.

alias Dh(addr) dump -from addr -bits 256 -hex
alias Dd(addr) dump -from addr -bits 256 -decimal
alias Da(addr) dump -from addr -bits 256 -character

# The following dump command aliases take two arguments:
# addr and n. Use n to specify the number of bits you
# want grouped as a unit in the resulting display of
# memory.

alias Dan(it,n) dump -from it -bits n -character
alias Dn(it,n) dump -from it -bits n -hex


      # The following alias searches for procedure names
      # containing the character string str, and prints the
      # fully qualified name for each procedure name displayed.

alias find(str) list blocks >tmp; sh grep str tmp
```

A Sample Personal Startup File (continued)

Emulating Other Debuggers

The HP/DDE install area contains sets of macros that you can use if you prefer to use `dbx` or `xdb` commands in place of HP/DDE commands. Additionally, you can enter any `dbx` command without using macros if you prefix the command with `dbx_`. The following sections detail how to use `dbx` or `xdb` commands in place of HP/DDE commands.

Note On Solaris systems, the command files mentioned in the following sections are found in:

```
/opt/softbench/dde/dde/contrib
```

Compatibility with xdb

The HP/DDE online help contains command maps that show HP/DDE equivalents for many common `xdb` commands. It also contains other maps that show HP/DDE equivalents to many `xdb` features. To access these maps, select `xdb Commands and HP/DDE Equivalents` from the `Common Debugging Tasks: Command Line` menu.

The command file `/opt/langtools/dde/contrib/dderc_xdb` defines some aliases for `xdb` commands. You can read the aliases directly into your debugging session with the following command:

```
</opt/langtools/dde/contrib/dderc_xdb
```

Alternatively, you can copy the command into a `.dderc` file in your home directory. The debugger automatically reads that file at startup.

Note The `dderc_xdb` file is not a complete list of the `xdb` equivalents of HP/DDE commands. Only commands that are easy to map are aliased in this file. See the online help for a complete `xdb`-to-HP/DDE command mapping.

See "Using a Personal Startup File to Customize the Debugger" for details on creating a `.dderc` file.

Be aware that the macros affect the *input* syntax of commands but have no effect on the way the debugger formats its output. Debugger command output does not resemble `xdb` output.

Compatibility with dbx

The debugger provides a set of commands that make `dbx` and debugger commands compatible. Compatibility commands for `dbx` consist of the prefix `dbx_` and standard `dbx` syntax. For example, the debugger command `break 27` and the compatibility commands `dbx_stop at 27` both set a breakpoint at line 27.

The command file `/opt/langtools/dde/contrib/dbx_macros` defines macros that let you enter `dbx` commands without prefixes. For example, you can specify `stop at 27` rather than `dbx_stop at 27`. You can read the macros directly into your debugging session with the following command:

```
</opt/langtools/dde/contrib/dbx_macros
```

Alternatively, you can copy the command into a `.dderc` file in your home directory. The debugger automatically reads that file at startup.

See "Using a Personal Startup File to Customize the Debugger" for details on creating a `.dderc` file.

Using the compatibility macros in the `dbx_macros` command file masks out some debugger commands. That is, the debugger executes the macro instead of the debugger command. Debugger commands that are masked out by macros remain available through the menus. Prefixing the command with a period (`.`) also inhibits macro expansion. For example, the `dbx_macros` command file defines a `step` macro. You can type `.step` to invoke the debugger's `step` command rather than the `step` macro.

Be aware that the macros affect the *input* syntax of commands but have no effect on the way the debugger formats its output. Debugger command output does not resemble `dbx` output.

7

Identifying Program Objects

This chapter describes the debugger's concept of environment, scope and visibility rules, and how you can refer to objects outside the scope of the current environment. Topics include:

- Understanding blocks and environments
- Changing the environment
- Overriding the current language
- Applying scope and visibility rules
- Using qualified names

The debugger's online help has more information on how to specify locations, blocks, and environments to the debugger. See the following sections, which are located under `Common Debugging Tasks`:

- `Specifying Locations to the Debugger`
- `Specifying Blocks to the Debugger`
- `Specifying Environments to the Debugger`

Understanding Blocks and Environments

A block is a program unit, such as a module, a main program, a subroutine, or a function. What constitutes a block depends on the language in which the program is written. Use the `list blocks` command to display the blocks in the program you are debugging.

A block defines and encloses a scope, the region of source code over which a name's declaration is active. The debugger also defines blocks called `'declared` (for user-declared symbols), `'predefined` (for data types for supported languages) and `'image` (for the program images).

The debugger defines two environments:

- The run environment is the block containing the current point of execution.
- The current environment is the block to which the debugger refers when evaluating expressions.

The current environment is identical to the run environment, except when you explicitly change it to another block (by using the `environment` command, for example). The source file display shows an arrow at the current point of execution. When the current environment is set to some other block, a horizontal bar appears within the block where the current environment is set.

In addition, the block containing the current point of execution is displayed after the `PC Location` label above the source display. The current environment block is displayed after the `Current Location` label.

Each time execution of the target program stops, the debugger sets the current environment to the run environment.

The debugger starts its search for objects within the block that corresponds to the current environment. If the debugger cannot find the object in the current environment, it extends its search into outer encompassing blocks. The debugger bases its search of outer blocks on the scope and visibility rules described in "Applying Scope and Visibility Rules".

For multi-threaded applications (implemented using HP DCE threads), the debugger also includes a thread component in its concept of environment. See "Debugging Multi-Threaded Applications" in Chapter 8 for more information on debugging multi-threaded applications.

Changing the Environment

Commonly, you may need to change environment because you cannot access a variable or expression from the current point of execution. For example, setting a watchpoint on a variable fails and a `not found in current environment` message is displayed.

To change the environment, use one of the following methods:

- Invoke one of the selections under the `Visit` menu.

  You can specify a procedure name, a line number, or a file name. Or you can specify the environment by using the debugger location syntax described in the online help.

  Use `Visit:PC` to return to the current point of execution.

- Use one of the up/down arrow buttons located next to the `Stack Frame:` label above the source display.

  These buttons allow you to change environment relative to frames on the call/return stack.

  Similar up/down arrow buttons are available from the Stack View dialog box that you can invoke from `Show:Stack`.

- Use the `environment` command and specify a location or a stack frame.

  See the online help for the syntax to use when specifying a location.

  The reserved identifier `'env(+`*n* `|` `-`*n*`)` is useful for changing environments relative to frames on the call/return stack. See "Frame Block Qualified Names" for more information.

  For example:

`env 17`	Changes environment to line 17 in the source code.
`env print_average`	Changes environment to the procedure, `print_average`.
`env 'env(-1)`	Changes environment up one frame (toward `'main`) on the call/return stack.

7

Overriding the Current Language

The debugger uses the source language of the current environment when evaluating expressions. You can use the `property language` command to change the language of evaluation, or you can use the `print -language` command to override the default language used to evaluate a single expression.

For example, when the current environment is using the C language manager, you can use FORTRAN as the language of evaluation for a command:

```
property language fortran
```

Now, although C is the language that corresponds to the current environment, you can evaluate a FORTRAN expression:

```
break 32 -do[if a.eq.b -then print a]
```

These commands are fully described in the online command reference.

Applying Scope and Visibility Rules

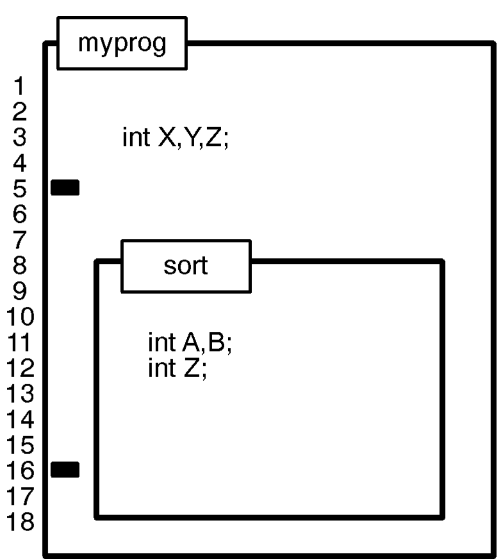
The debugger applies certain scope and visibility rules to find symbols.

The debugger first searches within the block that contains the current location. If the symbol is found, the search is completed. If the symbol is not found, the debugger searches outer, encompassing blocks. A symbol in the outer, encompassing block is visible to the inner block as long as no symbol of the same name exists in the inner block. See Figure 7-1 for an example.

If the search within the scope of the current environment fails, the debugger applies additional rules to locate the name. The following is a summary of the debugger's search order:

1. The block containing the current location.
2. Outer encompassing blocks (lexical parents).
3. The predefined language (`‘predefined`) block.
4. The user-declared block (`‘declared`).
5. Global symbols and top-level procedures (external).

Note Special language-specific search rules can also be used. For example, when debugging C++ programs, you can find class members.



In the environment `myprog\5`, the name `Z` resolves to the variable `Z` declared on line 3. (The name `A` is unknown in this environment.)

In the environment `sort\16`, the name `Z` resolves to the variable `Z` declared on line 12. (The name `A` resolves to the variable `A` declared on line 11.)

Figure 7-1. Sample Module Illustrating Scope and Visibility

In addition, you can explicitly reference symbols outside the scope of the current environment by using qualified names. The following section describes the syntax for using qualified names.

Using Qualified Names

You can use qualified names to have the debugger search in a specific block, a top-level procedure or module, an executable image (a loaded object file or shared library), or a frame on the call/return stack.

Block Qualified Names

The debugger uses **block qualified names** to refer to variables that are not visible from the current environment. A block qualified name explicitly identifies the block enclosing the object and the object's name; the format is *block**object_name*. Specifying a block in addition to a name changes the scope by specifying the starting block of the search.

For example, if the current environment is line 16 in Figure 7-1, you can use the block qualified name

```
myprog\Z
```

to refer to variable `Z` on line 3. Use the block qualified name

```
sort\Z
```

to refer to variable `Z` on line 12 in Figure 7-1.

With block qualified names, you can also combine variables from different blocks in a single expression. For example, use

```
myprog\Z + sort\Z
```

to refer to the sum of `Z` in `myprog` and `Z` in `sort`.

You can use block qualified names to explicitly identify variables that are not visible, including variables in currently inactive modules (files) and routines. Note, however, that if the value identified is a stack allocated variable, the variable is only visible when it is on the call stack.

Fully Qualified Names

A fully qualified name identifies all blocks that enclose the object.

To identify objects, the debugger uses fully qualified names in the following format:

module\\[*block*[*block*. . .]]*object_name*

where *module* is a module or a top-level name.

In Pascal, *module* is specified by the `module` keyword. In C, *module* is the source file name with the `.c` extension removed. In FORTRAN, modules do not exist, so a top-level name is the routine name—`MAIN`, a program name, or a subroutine.

Specifying a fully qualified name is useful for avoiding ambiguity, when you want to reference an object that is not visible from the current environment.

Use the `list blocks` command to display the fully qualified names for all the blocks in the target program.

For example, Figure 7-2 shows a target program in C compiled from two modules, `m1.c` and `m2.c`.

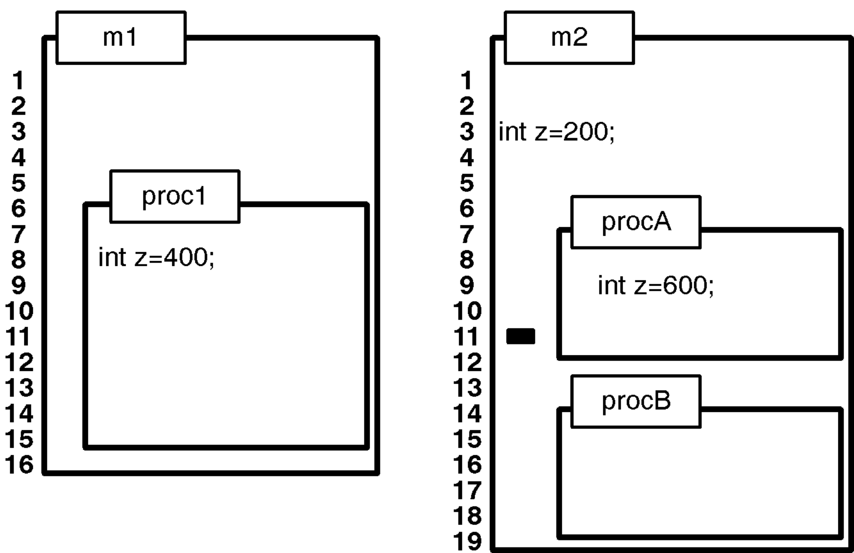


Figure 7-2. Sample Modules Illustrating Fully Qualified Names

The following list shows how to reference each instance of the variable `z` by using fully qualified names:

`\m2\z` References the variable `z` defined on line 3 of `m2`.

`\m2\procA\z` References the variable `z` defined on line 9 of `m2`.

`\m1\proc1\z` References the variable `z` defined on line 8 of `m1`.

Note that fully qualified names can be preceded by either a single backslash (`\`) or a double backslash (`\\`). The debugger, however, prints fully qualified names preceded by a double backslash. For example:

```
print \m2\procA\z
\\m2\procA\z: 600
```

By default, the debugger prints fully qualified names. You can change the default behavior with the `property qual_max` command. For example, if you specify `property qual_max 0` and issue a `print` command, the debugger prints only the object name and its value; other blocks enclosing the object are not identified.

Image Qualified Names

An image qualified name may be necessary to eliminate ambiguity among module names when an application consists of more than one executable image (loaded object file or shared library). This occurs whenever an application contains a dynamically loaded library. More than one image may contain modules with the same name.

An image qualified name starts with the notation `` `image( ``*image_name*`)`, where *image_name* is the shortest unique path name of the object file. Use the `list images` command to list the names of the known images.

For example, suppose that your application consists of the executable image `average` that contains the procedures `main`, `sum`, and `print_average`, and the dynamically loaded libraries `highlib` and `lowlib`. Since one or more of the libraries may contain a procedure named `print_average` or `sum`, you can use the image qualified name

```
`image(average)\average\sum
```

where `` `image(average) `` contains `average`, to name the procedure `sum`.

Special Block Qualified Forms

The debugger provides additional block qualified forms that you can use to refer to symbol names that are defined by the programming language or the user, and to refer to symbols that are externally declared.

Qualified Names for Predefined, User-Declared, and External Symbols

Symbol names that are defined by the programming language (usually type names such as `integer` or `int`) begin with `` `predefined `` and reside in outer debugger blocks. Use the `list blocks` command to view these blocks and names. You can explicitly refer to a language-defined symbol name by using a block qualified name of the form `` `predefined( ``*lang_type*`,`*obj_type*`)\`*name*. For example:

```
`predefined(lang_c,obj_som)\int
```

See Appendix B and Appendix D for information on the valid values for *lang_type* and *obj_type*.

In This Book

You may explicitly refer to user-declared symbol names, created with the `declare` command, by using a block qualified name of the form `‘declared\`*name*. User-declared symbol names are defined in an outer block named `‘declared`. Since program symbol names visible from the current environment are searched before user-declared symbol names, user-declared symbol names may be hidden by program symbol names.

When the debugger cannot find a symbol name by following the normal scope rules or by searching the outer language or user-declared blocks, it looks at variables declared to be external (that is, variables declared to be visible everywhere). Similarly, when the debugger cannot find a block name, it searches for a procedure not enclosed within another procedure.

Frame Block Qualified Names

You can identify dynamically activated symbols within recursive procedures using frame block qualifiers. **Frame block qualifiers** use both a block and a frame to specify the current environment.

Recursive procedures are called multiple times and produce multiple frames (invocations) on the call/return stack. The debugger distinguishes one invocation from another by identifying the current environment as both a block and a frame. The block serves as a naming context, as described in “Block Qualified Names”, and helps to determine which variable names are visible and which are not. The frame indicates which invocation of that block the debugger should use to locate local data.

When a recursive procedure produces more than one instance of a block on the call/return stack, local symbols in that block have more than one instance. The current environment consists of both a frame and a block. The debugger uses the block to determine which variable is intended, and it uses the frame to determine which instance of that variable to use.

If a block has not been invoked, then the block is said to be inactive, and you may only examine its static data objects. If a block has been invoked once, then that instance is used. In the example shown in Figure 7-3, use `print_answer\COUNTER` to refer to the `COUNTER` variable in frame `‘main(5)`. If a block has been invoked more than once, the most recent instance is chosen by default; for example, `binary_search\HI` refers to the `HI` variable in frame `‘main(4)` by default.

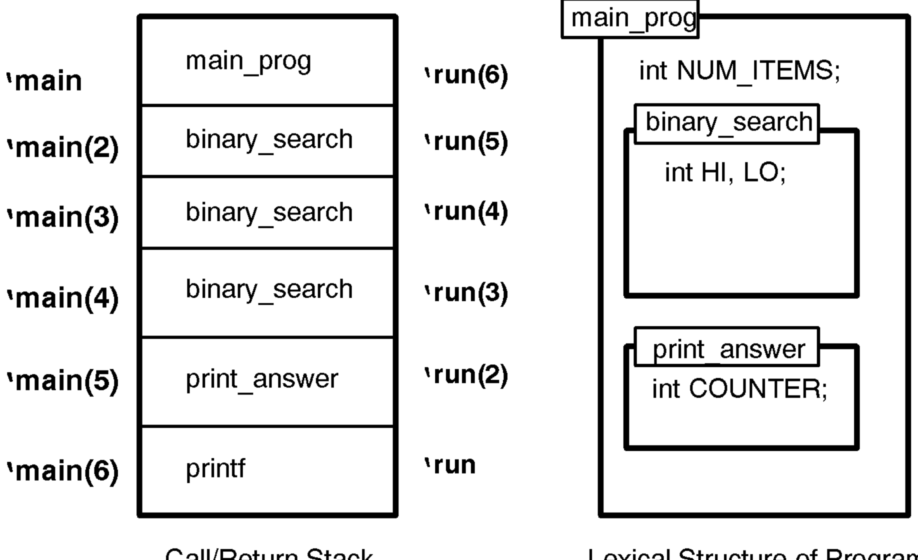


Figure 7-3. Sample Call/Return Stack and Program

There are three ways to refer explicitly to symbols using a frame block qualifier:

main relative

The notation is `'main(`*n*`)` where *n* is the number of the frame counting from the least recent frame of the call/return stack to the desired frame. In the example in Figure 7-3, the frame with `print_answer` is `'main(5)`. `'main` by itself refers to the oldest frame in the stack. This frame may not refer to your `main` program because in some cases a procedure in a run-time library may be at the base of your stack.

run relative The notation is `‘run(`*n*`)` where *n* is the number of the frame counting from the most recent frame in the program (that is, the location where the program is stopped). In Figure 7-3. the frame with `print_answer` is `‘run(2)`. `‘run` by itself refers to the location where the program is stopped.

environment relative The notation is `‘env(-`*n*`)` where *n* counts up (toward `‘main`) from the current environment. The notation is `‘env(+`*n*`)` where *n* counts down (toward `‘run`) from the current environment. `‘env` by itself refers to the current environment.

In Figure 7-3, you can refer to the instances of variable `HI` as `‘run(3)\HI`, `‘run(4)\HI`, and `‘run(5)\HI`. You could also refer to the variables with `‘main` or `‘env` notation. A frame qualified name may not contain block qualifiers (that is, `‘main\blk` is not valid).

Also, you can use frame names with the `environment` command to change your environment to any procedure on the call/return stack. For example, you can set the environment to the current point of execution using:

```
environment ‘run
```

8

Debugging in Special Situations

The HP/DDE debugger provides support for debugging in many different situations. This chapter describes the debugger's ability to:

- Examine core files
- Debug shared libraries
- Debug multi-threaded applications
- Debug assembly language code
- Debug optimized code
- Debug parent and child processes
- Debug applications that use `ioctl` or `curses`
- Run remotely

For more information on the debugger commands mentioned in this chapter see the online command reference. You can also invoke help on a particular command by entering `help` *command_name* in the debugger command input box.

The online help also contains more information on using the debugger's graphical user interface.

Examining Core Files

You can use the debugger to diagnose some run-time errors after a program has aborted and produced a core file.

The operating system generates a core file when a signal is not caught by the program. The core file records the state of the program at the time that the fatal error occurred.

You can use the debugger to:

- Determine which signal or signals caused the failure.
- Trace back the call/return stack.
- Examine the state of global static variables, local static variables, variables stored on the stack, and registers.

You cannot modify or execute the program.

Attaching to a Core File

When attaching the debugger to a core file on HP-UX systems, you must supply both the name of the core file (usually `core`) and the name of the program object module that was being executed. (On Solaris systems, you can specify just the name of the core file.)

Choose `File:Load Corefile` or use the `debug` command with the following syntax:

`debug` *core_file object_program_pathname*

If the core file was produced by a stripped version of the executable, you can debug it using the unstripped version of the executable. For example, suppose you have two executables, `prog.stripped` and `prog.unstripped`. If `prog.stripped` dumps core, issue the command

```
debug core prog.unstripped
```

The stripped and unstripped executables must be identical except for the stripping.

Core File Debugging

When your program is first loaded, the PC is set to the line that caused the core dump. The stack and all variables are as they were when the program was about to execute that line. Here are some useful actions that can help to locate problems:

- Examine the line in the Source File Edit Area that was responsible for the core dump to see if there is an obvious problem.
- Enter a local or global variable in the `( ):` Input Box, then select [Print ( )] to examine its value. Check for a value out of range.
- Choose `Show:Stack` to look at the procedure call stack. Check values of passed parameters to see if the problem originated earlier in your program.
- Choose `Show:Assembly Instructions`, and examine CPU registers or assembly instructions to get a low-level view of your program (see "Assembly Level Debugging").

Debugging Shared Libraries

You can debug **shared libraries** that are either implicitly or explicitly loaded by your program. Implicitly loaded shared libraries are libraries that your program is linked against. Explicitly loaded shared libraries are loaded by calls within your program to *shl_load* (3X). (On Solaris systems, shared libraries are loaded by calls to *dlopen*(3X).)

To debug shared libraries, you must compile your main program debuggable (with the `-g` option). A main program compiled without `-g` does not have enough information about shared libraries to allow you to debug them, even if the libraries were compiled with `-g`.

If the shared library you want to debug was compiled with `-g`, you can use full source-level debugging. If it was not compiled with `-g`, you must debug at the assembly level; but you can step into library routines, set breakpoints in the routines, and refer to symbols in the library.

The basic steps in debugging a shared library, and the debugger commands that accomplish these, are as follows:

1. Make the library writable using the `property flags tgt_shlib_debug` command. This command is issued by default when you invoke the debugger, but you can turn it off if you wish.
2. Intercept the loading of a library using the `intercept load` command. This command is required only if you want the debugger to stop at the loading of an explicitly loaded shared library. You can use `Execution: Signals/Intercepts` ... to perform this action.
3. Load the library. You do not need to issue a command to do this. An implicitly loaded shared library is loaded at program startup. An explicitly loaded shared library is loaded by a *shl_load*(3X) call.
4. Load debugging information for the library using the `property libraries` command. You can issue this command before or after the library itself is loaded.

   The Dynamic Images dialog box also allows you to perform this action. Invoke it by selecting `Execution:Enable Images/Libraries`.

5. Load alternate debugging information for a program or library that has been compiled partly with and partly without debugging information, so that you can debug the routines compiled without debugging information (`initialize -altdbinfo`). In this unusual situation, you must issue the `initialize` command after the library is loaded.

You may also find the following command useful:

`list images` This command gives you a list of the shared libraries your program uses. Use `list images -full` to find out whether these libraries are mapped writable. The `Execution:Enable Images/Libraries` ... dialog box also allows you to perform this action.

See the online help for more information and examples on debugging shared libraries.

Debugging Multi-Threaded Applications

HP/DDE supports the debugging of multi-threaded applications that are implemented with HP DCE threads. (The Softbench version of the debugger for Solaris systems supports Solaris Threads.)

HP/DDE thread support includes:

- A Threads dialog box for monitoring, examining, and manipulating threads
- Support for setting breakpoints on specific threads in the Breakpoint Set/Change dialog box
- A *thread_id* identifier that allows specification of individual threads
- A thread component added to the debugger's concept of environment
- Thread-specific commands and options

You can prepare and invoke a multi-threaded application for debugging like any other target program. See "Preparing the Target Program" in Chapter 2 for more information on preparing and invoking target programs for debugging.

Multi-threaded applications may execute quite differently when they are invoked by the debugger because the debugger takes control of the thread scheduler. Some debugger commands, like `step` and `thread -select`, override the thread scheduler.

See *Programming With Threads on HP-UX* for more information about HP DCE threads.

Making libdce.sl Writable

If your application uses `/usr/lib/libdce.sl` (the shared library version of HP DCE threads), you must make the shared library writable before you debug the application.

The following command maps all shared libraries as writable:

```
property flags tgt_shlib_debug
```

You can place the command in your `.dderc` file. You can also invoke it from the debugger command line before you issue the `debug` command.

When `libdce.sl` is read-only, you cannot step into routines or set breakpoints in them. Also, some thread-related debugger commands will not work. The affected commands are:

```
intercept thread_create
intercept thread_exit
intercept thread_switch
thread
```

If `libdce.sl` is read-only and you attempt to use one of the commands listed above, the debugger will display a `not mapped writable` message.

Stripped and Unstripped Versions of libdce.sl

If your application links in an unstripped version of `/usr/lib/libdce.sl`, all threads-related commands and options will work properly.

If, however, your program links in a stripped version of `/usr/lib/libdce.sl`, the threads-related commands and options will behave as if your program had only one thread.

Use the `nm` command to determine if your system has a stripped version of `libdce.sl`. The output of `nm` contains a `no symbols` message when `libdce.sl` is stripped. For example:

```
$ nm /usr/lib/libdce.sl
  nm:  /usr/lib/libdce.sl:  no symbols
```

If `libdce.sl` is stripped, link your program with the archive library `/usr/lib/libdce.a` in order to debug it.

Viewing and Manipulating Threads

Choose `Execution:Threads` to display the Threads dialog box. The Threads dialog box, shown in Figure 8-1, displays a list of threads in your program. The list is automatically updated as threads are created or change status.

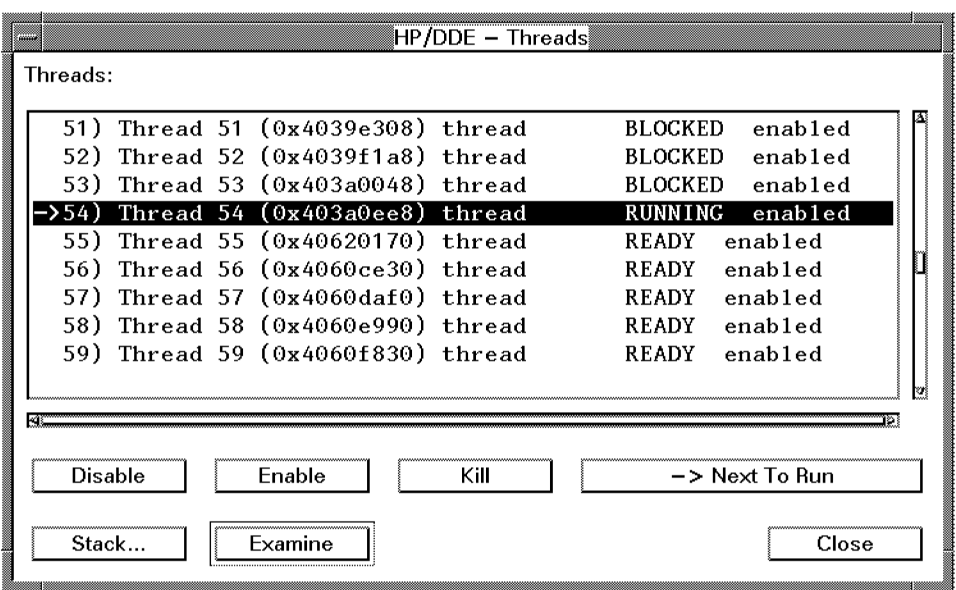


Figure 8-1. The Threads Dialog Box

Notice the following in the thread list:

- A numeric thread identifier is displayed for each thread. The thread number is assigned by the debugger.

  Thread 1 is the initial thread (also known as the distinguished thread) and Thread 2 is the system thread. The debugger always assigns the first two identifiers to these threads.

  You can use these numeric identifiers for commands that require a *thread_id* (for example, `breakpoint -thread` *thread_id*).

- Threads are also labeled by the initial procedure name. In this case, the initial procedure name was `thread`.

- Thread status is indicated with the labels `RUNNING`, `READY`, `BLOCKED`, and `TERMINATED`. Note that execution of `TERMINATED` threads has ended but the thread itself has not yet been deleted.
- The point where execution has stopped is indicated with the `->` marker.

You can also use the `list threads` command to display a list similar to the one in the Threads dialog box.

When you highlight one or more threads in the Threads dialog box, you can select a button to request the following actions:

Disable — Remove the selected thread(s) from the list of threads that can run when the program is resumed.

Enable — Add the selected thread(s) to the list of threads that can run when the program is resumed.

Kill — Kill the selected thread(s).

Stack Trace... — Display the selected thread's call/return stack in a new window.

To monitor the call/return stack of each thread as it executes, invoke the Stack View dialog box by selecting `Show:Stack` from the menu bar.

-> Next to Run — Make the selected thread the current *execution* thread. This thread will be the first to run when the program is resumed.

Examine — Change environment to the selected thread. This allows you to examine the status of a thread without changing the current Execution thread.

Setting Breakpoints on Threads

You can activate breakpoints on all threads, or on any selected set of threads.

The `Breakpoints Set/Change` dialog box contains an area at the bottom which allows you select which threads the breakpoint will be set on. Choose `Break:Set` to invoke the Breakpoints Set/Change dialog box.

Also, the `breakpoint` command has a `-thread` option. See the online command reference for more information and examples.

Environment in Multi-Threaded Applications

When the target program is multi-threaded, the debugger includes a thread component in its concept of environment. (See Chapter 7 for more information on the debugger's general concept of environment.)

For example, the following shows the output from a `list environment` command during a debugging session on a multi-threaded target program:

```
list environment
Stopped at: \\file_copy\worker\222 (`thread(3))
```

Notice the output indicates that the current environment is in Thread 3.

It is useful to invoke the Stack View dialog box when debugging multi-threaded programs. You can invoke it by selecting `Show:Stack` from the menu bar.

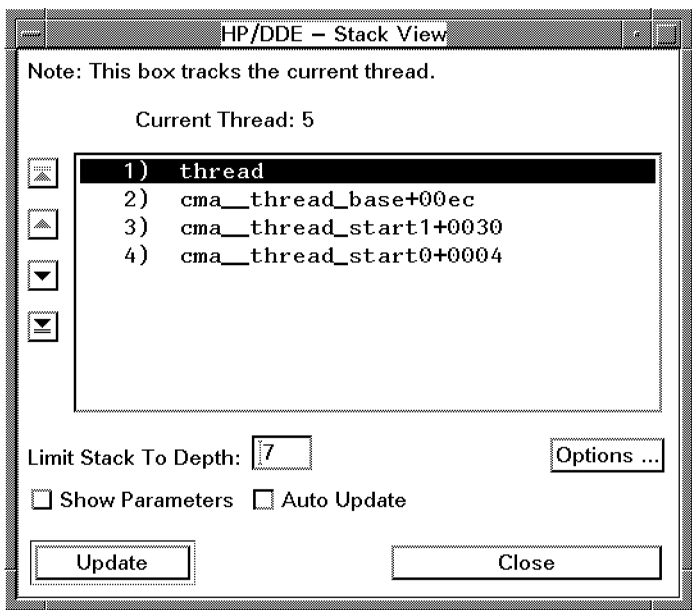


Figure 8-2. Stack View Dialog Box Showing Current Thread

As shown in Figure 8-2, the Stack View dialog box shows the call/return stack for the current thread. The current thread is the thread that the debugger

refers to when evaluating expressions. It differs from the running thread when you change environment from one thread to another. For example, you can use the thread specifier, `thread(*n*) to change the current thread, in the following context:

```
environment `thread(3)
```

Notice that the call/return stack in multi-threaded applications includes a number of system calls. In Figure 8-2, the system call names begin with the cma__ prefix.

Arrow buttons in the Stack View dialog box allow you to move up and down the call/return stack within the current thread. You can also use the frame specifiers described in Chapter 7 (`env, `run, and `main). The command environment `env(-1), for example, moves to the caller of the current frame without changing the current thread.

In addition, you can combine thread specifiers with frame specifiers to write fully qualified environment specifiers. The following shows the syntax for fully qualified environment specifiers:

```
`thread(thread_id)\`main(frame_number)
`thread(thread_id)\`run(frame_number)
`thread(thread_id)\`env(+|-n)
```

The `thread(*thread_id*) notation may also be used wherever another environment qualifier can be used. The command print `thread(5)\x, for example, prints the variable x in the environment of the run frame of Thread 5.

Thread-Specific Debugger Commands

The two commands that apply specifically to debugging threaded applications are list threads and thread.

Use the list threads command to obtain a list of thread identifiers for a target program. You can also use the Threads dialog box which is described in "Viewing and Manipulating Threads".

The thread command changes the state of program threads. The following briefly describes the thread command with its options:

- thread -select alters the scheduler so that the specified thread runs when the program executes. This option is similar to the goto command in that

it alters the point of execution of the program. The debugger cannot ensure that the program state is valid following this transfer of control.

The selected thread is the only one operated on by a `step` command.

- `thread -disable` prevents specified threads from running until they are enabled again using `thread -enable`.
- `thread -enable` allows the specified thread to run when you issue a `go` command. By default, all threads are enabled; ordinarily you use `thread -enable` to reactivate a disabled thread.
- `thread -kill` marks the specified thread for termination. The killed thread persists until the thread scheduler actually terminates it.

In addition, the `intercept` command can take the following arguments when you are debugging multi-threaded applications:

`thread_create`	Stops program execution when a thread is created.
`thread_exit`	Stops program execution when a thread is terminated.
`thread_switch`	Stops program execution when the context changes from one thread to another.

Finally, some debugger commands allow you to specify a thread by using the `-thread` option. Those commands are:

```
activate breakpoints
breakpoint
delete breakpoints
suspend breakpoints
tb
```

For more information on the commands described in this section, see the online command reference.

Assembly Level Debugging

Although HP/DDE is intended as a high-level (source language) debugging tool, it also supports low-level (assembly language) debugging by using the machine code generated by the compiler.

You can examine the machine code produced by the compiler for your program, and step your program at the assembly code level. You can set breakpoints on individual machine instructions, and you can also monitor virtual address ranges.

Assembly level debugging is particularly useful for debugging optimized code. See "Debugging Optimized Code".

Using the Assembly Instructions Dialog Box

The `Show:Assembly Instructions` menu choice brings up the Assembly Instructions dialog box. (See Figure 8-3.) This dialog box contains a scrollable view of the assembly language code for the current procedure of your program. The format of the code depends on the system on which you are running. The window normally shows the following:

- Source line number
- Memory address
- Disassembly listing of the actual machine code

To set a breakpoint at the assembly level:

1. Select the address (in the Assembly Instructions dialog box). This copies the address into the `( ):` Input Box.
2. Choose `Break:Set At Hex Address ( )`.

You can also click the left mouse button in the Annotation Margin of the Assembly Instructions dialog box, or press the right mouse button on the desired assembly line, to set or clear breakpoints.

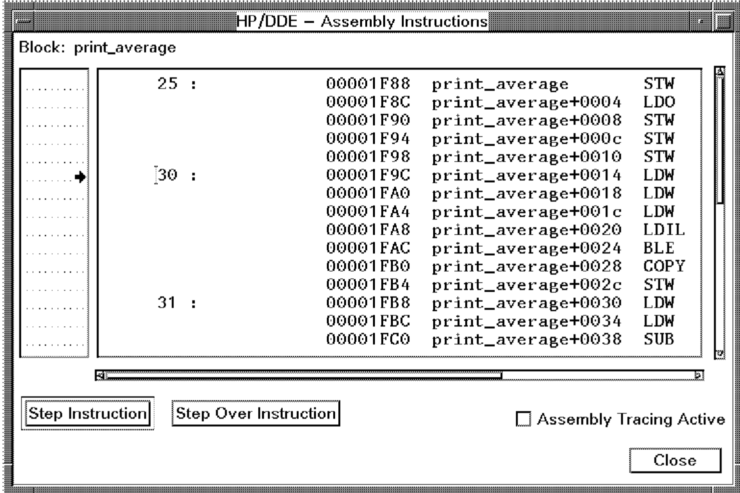


Figure 8-3. Assembly Instructions Dialog Box

Use the (Step Instruction) button to step through your program one assembly instruction at a time. Use (Step Over Instruction) to step over routines called from the assembly code.

The `Assembly Tracing Active` button enables updating of the Assembly Instructions window. If assembly tracing is disabled until you need it, your program might run faster.

Using Debugger Commands

You can display ranges of memory with the `dump` command and watch a virtual address range with the `watchpoint` command. Refer to the `dump` and `watchpoint` command descriptions in the online command reference for details. The `dump` command remembers the last location that was dumped and continues dumping from the next address when you issue another `dump` command using `*` as the `-from` address. The `dump` command also prints the previously specified number of bits by default.

For example:

```
dump -from 2260 -to 227E -bits 16 -hex
00002260: 6BD8  3F29  2B7F  FFFF  4821  0F58  8C20  2012
00002270: E800  0148  0800  025C  E800  0122  23E1  0000

dump -from * -to 2284 -binary
00002280: 1110011111100000  0010011010101000  0000100000011111
```

The target manager startup file defines aliases that dump the state of the machine registers and that monitor registers for changes. See the "Startup File" section in Appendix C.

You can use the debugger's `'va` reserved identifier to print the contents of any machine location or to store a value to any machine location. This function takes a machine address (in hexadecimal format) or register name as an argument and acts like an untyped variable at that address. You can use `'va` in combination with type casting to print in any format. For example:

`print (long)'va(2F5A)`	Prints the contents of 2F5A as a long integer (C syntax).
`print integer('va(2F5A))`	Prints the contents of 2F5A as a long integer (Pascal syntax).
`print (my_struct)'va(2F5A)`	Prints the value starting at 2F5A as a record (C syntax).
`set integer32('va(31CF450)):=99`	Assigns a long integer to an address (Pascal syntax).
`watch (long)'va(2F5A)`	Watches the contents of 2F5A as a long integer (C syntax).

You can use the `'va` function to refer to a specific code location in any command that calls for a location specifier. Consider the following examples:

`break 'va(00001FCC)`	Sets a breakpoint on a specific machine address.
`goto 'va(1FCC)`	Transfers control to a specific machine address.

You can also use the `describe` command with the `'va` function. For example:

`describe -location 'va(1FCC)`	Finds the source statement that corresponds to a code location.

Saving Assembly Code in a File

To save assembly code in a file for printing or viewing, redirect output of the `dump` command using the following syntax:

`dump -from` *address* `-to` *address* `-instruction >`*filename*

You can determine *address* values from the Assembly Display window.

Debugging Optimized Code

HP/DDE supports debugging of code compiled at optimization levels 2 and below. The following is a brief description of the compiler optimization options that are compatible with debugging:

`+O0` Minimal optimization. This is the default.

`+O1` Basic block level optimization.

`+O2` Full optimization within each procedure in a file. (Can also be invoked with the compiler option `-O`.)

For more information about optimization levels, consult your compiler documentation.

Ordinarily, you first compile and debug your program without optimization. All or nearly all of the bugs in your program will show up in the unoptimized version.

After eliminating all the bugs that you can find, turn on optimization (compile with `-O`). If the program behaves incorrectly, scan the source code for the most common kinds of bugs that appear for the first time in optimized code:

- Uninitialized variables
- Out-of-bounds array references
- Variable references based on the assumption that two variables are adjacent in memory

These kinds of problems, however, are often very difficult to find by examining the source code. If you cannot determine the reason for the program's misbehavior, you need to debug the optimized code.

This section provides background information on the differences between optimized code and unoptimized code.

For tutorial and task-oriented information on how to debug optimized code using the debugger, see the online help.

Optimized Code and Unoptimized Code

Source-level debugging of unoptimized code is relatively easy because there is a simple correspondence between source-code statements and the assembly-code instructions into which they are translated. Each statement is translated into a contiguous series of instructions, which are executed in sequence. The source and object code are isomorphic: they have essentially the same form. Also, program variables are stored in memory and are therefore easy to access.

Optimization destroys isomorphism. Optimization is a series of transformations performed on the object code in order to make the program run faster. An optimized program performs the same tasks and produces the same results that the source code specifies. However, the order in which these tasks are performed and the way in which they are performed can change drastically.

In effect, optimization transforms a program into a different program.

The executable program you are debugging is actually not the same program as the source program.

In addition, program variables are stored in registers instead of memory and are therefore more difficult to access.

The following sections describe these problems in detail.

What Optimization Does to Program Logic

Figure 8-4 shows how source-code statements map to object-code instructions in unoptimized code. Every instruction in the object code corresponds to a single statement in the source code. And, in general, every statement in the source corresponds to a sequential group of instructions in the object code. (There are a few exceptions; a loop, for example, may be broken up into two groups of instructions, one at the beginning and one at the end of the loop.)

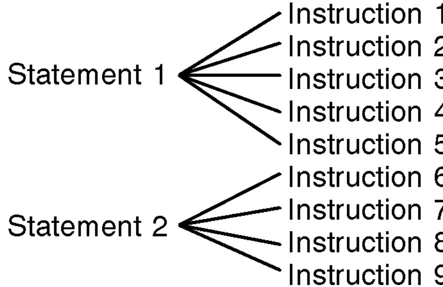


Figure 8-4. Unoptimized Code: Statement-to-Instruction Mapping

This means that even though it is the object code that is being executed and not the source code, a view of the source code in the debugger can still give you an accurate view of what the object code is doing. For example, when you step from statement 1 to statement 2, instructions 1 through 5 are executed in order, and the current location is now instruction 6, corresponding to statement 2.

Figure 8-5 shows the several things that happen to source-code statements in optimized code.

- As before, a source-code statement corresponds to several instructions. But these instructions are no longer contiguous; instead, there may be several groups of instructions, called fragments because they represent only part of a statement. (A fragment is formally defined as a maximal set of contiguous instructions corresponding to the same source statement.) In the example, the statement on line 11 corresponds to three fragments: instruction 4, instructions 9 and 10, and instruction 12.
- Instructions in the object code can now correspond to more than one source statement. In the example, instruction 4 is associated with the statements on lines 11, 12, 13, 16, 17, and 18.
- The order of statement execution can change. In the example, the instructions from line 12 both begin and end execution before the instructions from line 10, though some of the instructions from those lines are interleaved.

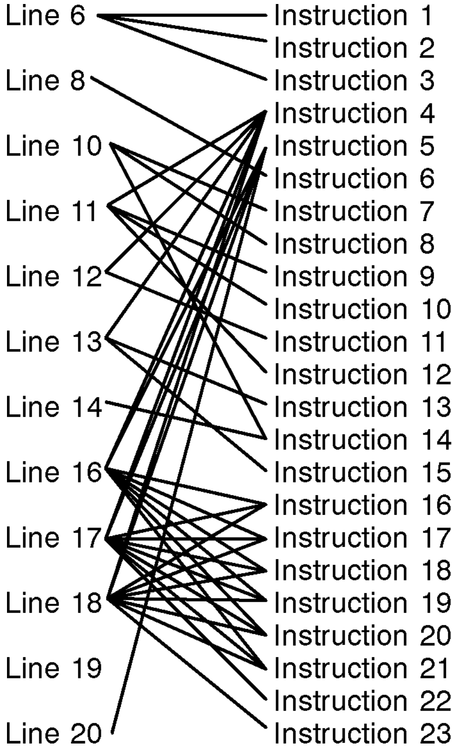


Figure 8-5. Optimized Code: Statement-to-Instruction Mapping

- For all of these reasons, the order in which instructions are executed no longer corresponds to the ordering of the source-code statements. In the example, instructions 9 through 14 come from lines 11, 12, 10, 13, 10, 14, and 13.
- Some source-code statements, such as line 19 in the example, have no corresponding object code at all. This can happen for several reasons, such as the elimination of code that is never executed (dead code).

When you debug unoptimized code,

- The source display gives you a good idea of where you are in the program. An arrow points to your current location; you know that the statements (and instructions) before the arrow have executed, and that the statement (and instructions) after the arrow have not yet executed.

- You can examine the current state of program data, because variables are stored in memory and their values are always accessible.

The reason for this is that in unoptimized code, when you step from one statement to the next in the source code, what the debugger actually does is to step from one group of assembly instructions to the next; but because the assembly instruction groups correspond exactly to the source statements, it looks as if it is the source code itself that is executing.

When you debug optimized code, however, this correspondence breaks down:

- An arrow cannot accurately represent your current location in the source code. When an arrow points to a given statement in the source code, it is likely that not all the statements before that statement have finished executing, and that some of the statements after that statement have at least partly executed.
- It is more difficult to determine the current state of program data, because variables may be stored in registers and access to them may be unreliable. Also, the order of assignments may be changed from the order in the source code.

In fact, it isn't really meaningful to talk of the current location in the source program with optimized code—only of the instructions that are actually executing.

What Optimization Does to Data

In unoptimized code, the values of all program variables are kept in memory. Every time the value of a variable changes, the new value is stored in memory. This means that at any time, the debugger can determine the current value of a variable.

In optimized code, the values of variables are kept in registers instead of memory as much as possible, because register access is much faster than memory access. In addition, some variables may be eliminated and replaced by constants. Therefore, it is much more difficult for the debugger to track the current value associated with a variable.

Sometimes, too, there may be multiple copies of a variable—in loop optimizations, for instance. In such cases it is often impossible to obtain a meaningful value for the variable.

Debugging Parent and Child Processes

Select `Options:Fork Behavior` or use the `property fork` command to specify the process to be debugged when the target program forks.

You have the following options when debugging programs that fork child processes:

- Ignore the child process and continue to debug the parent (`property fork -parent`).
- Free the parent process and debug the child (`property fork -child`).
- Debug both the child and the parent process (`property fork -parent -child`).

When you ask the debugger to debug both processes, it automatically creates another `dde` invocation for the child when the fork occurs. (You cannot debug both the parent and the child processes from the line-mode user interface.)

Choose `Options:Save All Settings` to retain the fork behavior for future sessions.

See the `property fork` command description in the online reference for more information.

Debugging Applications That Use ioctl or curses

Programs that use *ioctl*(2) or *curses*(3x) do not run in the User Program I/O Area as they would in a terminal window. The User Program I/O Area is not designed as a fully functional terminal emulator. However, you can still debug these types of programs with HP/DDE.

One method is to run the program in a terminal window and then attach the debugger to the process. See "Attaching the Debugger to a Running Process" in Chapter 2 for more information.

Another method is to use I/O redirection, as follows:

1. Start an `hpterm` or `xterm` terminal window with a running shell.
2. Run the `tty` command in the terminal window to get its device name (for example, `/dev/pty/ttyp5`).
3. Run the `sleep` command in the terminal window to keep the shell from intercepting any input (for example, `sleep 100000`).
4. Load the target program, but redirect standard I/O to the terminal window, using the device name obtained in Step 2.

   If you use the Load/Rerun dialog box, enter the device name (which is `/dev/pty/ttyp5` in this example) in the `stdin`, `stdout`, and `stderr` redirection input boxes.

   When the target program starts, the User Program I/O area disappears since all I/O has been redirected.

Running the Debugger Remotely

You can use resources provided by the X Window System to run the debugger on one machine while displaying the user interface on another. To run a debugging session over two machines in a network, you must:

1. Provide display access to the machine that will run the user interface. Enter the following command at a shell prompt on the machine that will display the user interface:

   `xhost` *debugger_machine*

   where *debugger_machine* is the hostname of the machine that will run the debugger. The hostnames are usually listed in `/etc/hosts`.

2. Log in to the machine that will run the debugger from the machine that will display the user interface. Typically, at a shell prompt, you would enter:

   `rlogin` *debugger_machine*

   where *debugger_machine* is the name of the machine that will run the debugger.

3. Invoke HP/DDE with the `-display` option from the shell where you logged into the *debugger_machine*:

   `dde -display` *display_machine*`:0`

   where *display_machine* is the name of the machine that will display the user interface. If you get a message stating that the `dde` command cannot be found, add `/opt/langtools/bin` to your `PATH` variable.

   Note that instead of using the debugger `-display` option, you can also set the `DISPLAY` environment variable to *display_machine*`:0`. You may get a `Broken pipe` error message if you forget to specify the `-display` option or set the `DISPLAY` variable.

Line-Mode User Interface

This appendix describes the components and operation of the debugger's line-mode user interface. The user interface manages all input and output and displays information about the target program during a debugging session.

You need to use this interface if your system does not have OSF/Motif installed. You may also use it if you prefer a simple line-based interface.

You can run the debugger's line-mode user interface under Emacs Version 18.*x* by using the HP/DDE-Emacs interface file:

```
/opt/langtools/dde/contrib/dde.el
```

Invoking the Line-Mode User Interface

You invoke the line-mode user interface by specifying the option `-ui line` to the `dde` command. When the debugger starts, a `dde>` prompt appears.

At the prompt, you can enter most of the commands described in the online command reference. However, note the following exceptions and recommendations:

- The following commands do not apply to the line-mode interface:
  - `property highlight`
  - `unhighlight`
  - `property echo -graphic`
- Use the `property transcript` command to obtain multiple-line displays of the Source File Area, the Assembly Instructions window, the Data Watchpoints window, and the Stack View window.
- Use the following macros to monitor registers:

`reg_update`	general registers
`fsreg_update`	single-precision floating-point registers
`fdreg_update`	double-precision floating-point registers

- The `help` command pipes the contents of the help file through the paging command defined by the `PAGER` environment variable. If `PAGER` is not defined, it uses *more*(1).

You can change the default user interface to the line-mode interface for all users of your system. See Appendix E for details.

The User Interface Startup File

The file `/opt/langtools/dde/ui/nls/C/ui_line.startup` contains several macros that may save you some typing:

- Use `w` to display the 10 lines immediately surrounding the current source line. The `w` command is an alias for `view -5 -lines 10`.
- Use `l` to display the 10 lines immediately following the current source line. The `l` command is an alias for `view -0 -lines 10`.
- The following macros execute common commands and allow you to repeat the command on subsequent lines by pressing the <RETURN> key:

```
g          go
gr         go -return
sa         search
s          step
so         step -over
si         step -instruction
up         env ‘env(-1)
down       env ‘env(+1)
```

- The `edit` macro invokes `vi` to edit the source file.

Screen Display Conventions

Letters and symbols in the margin of the transcript area indicate the following:

`>` Current point of execution (the equivalent of the arrow)

`B` Breakpoint (the equivalent of the stop sign)

`b` Suspended breakpoint

`*` Watched variable whose value has changed

`=` Current environment (the equivalent of the horizontal bar)

`V` Location requested by the `view` command

The following symbols are used when debugging optimized code:

`)` Midline PC location marker

`#` Variable whose value is unsure because of optimization

`C` Critical point breakpoint

`F` Fragment breakpoint

`c` Suspended critical point breakpoint

`f` Suspended fragment breakpoint

`!` Critical point location marker

`]` Midline critical point marker

`}` Fragment location marker

`|` Midline fragment marker

Examples

The following examples illustrate some of the capabilities of the line-mode user interface.

Invoke the debugger to debug the program `average`.

```
$ dde -ui ui_line average
Executing image in process 23640: "/home/smith/average".
Break at: \\average\main\44
Source File: /home/smith/average.c
   44 B>     print_average (my_list, first, last);
```

Print source lines and assembly code in the transcript area.

```
dde> property transcript -source -asm
dde> step -instruction
Stepped to: \\average\main\44 (0000202C)
   44 B)     print_average (my_list, first, last);
asm:   0000202C  main+000c  LDO            R'6d8(%r1),%r26
```

Set a watchpoint and the print source and the watched variable in the transcript area.

```
dde> watchpoint \\average\print_average\total
(Warning) The location of variable 'total' is not in an active frame.
dde> property transcript -source -variable
dde> step -over
The initial value of \\`main(4)\average\print_average\total is 0
var:    33) \\`main(4)\average\print_average\total: 0
The value of \\`main(4)\average\print_average\total has changed from 0 to 36.
var:    33) \\`main(4)\average\print_average\total: 36
   31  >         num_elements = high - low;/* note this is an off-by-one bug */
asm:         31:  00001FBC  print_average+0030  LDW            -108(%r30),%r1
Stopped at: \\average\print_average\31
```

Use the predefined alias `w` to print the 10 source lines surrounding the current line.

```
dde> w
   26  V int list[], low, high;
   27    #endif
   28        {
   29            int total, num_elements, average;
   30            total = sum(list, low, high);
   31  >         num_elements = high - low;
   32
   33            average = total / num_elements;
   34            printf("%10.d\n", average);
   35        }
```

Print source, assembly, watched variables, and traceback in the transcript area.

```
dde> property transcript -source -asm -variable -tb

dde> step
Stepped to: \\average\print_average\33
   33  >         average = total / num_elements;
asm:  > 33:  00001FCC  print_average+0040  LDW            -64(%r30),%r26
tb:     1    $START$+0094 (0000192C)
tb:     2    _start+0068 (80041D9C)
tb:     3    \\average\main\44 (0000203C)
tb:     4 >  \\average\print_average\33
```

Single step one statement using the predefined alias `s`.

```
dde> property transcript -source

dde> s
Stepped to: \\average\print_average\34
   34  >         printf("%10.d\n", average);
```

Repeat the last command (single step). (The 4 is the output of the `printf` call on the previous line.)

```
dde> <RETURN>
        4
Stepped to: \\average\print_average\35
   35  >     }
```

Use the predefined alias up to walk up the stack.

```
dde> property transcript -source -asm -tb
dde> up
Environment: \\average\main\44 (0000203C) (frame ‘main(3))
Stopped at: \\average\print_average\35
   44 B=     print_average (my_list, first, last);
asm:  =          0000203C  main+001c  BLE            R’78c(%sr4,%r31)
tb:     1    $START$+0094 (0000192C)
tb:     2    _start+0068 (80041D9C)
tb:     3 >  \\average\main\44 (0000203C)
tb:     4    \\average\print_average\35
```

B

Language Managers

Whenever the debugger performs a language-specific operation, a language manager provides the necessary language-specific information. This chapter describes the language managers that enable the debugger to evaluate expressions and declarations in different languages. Consult this chapter for information specific to the language managers, such as the value of the *language_type* argument to the `print` command's `-language` option.

On HP-UX systems, the debugger uses HP C, HP C++, HP FORTRAN, and HP Pascal high-level language managers and an HP-UX PA-RISC assembly language manager. On Solaris systems, the debugger uses a SPARC assembly language manager.

This chapter provides the following information about each manager:

Type Name
Identifies the name used for the manager as installed.

Title
Contains the name that the manager uses to identify itself. To list the titles of the managers currently loaded, use the `version` command. The following is an example of the output:

```
dde, version 4.0
User interface manager ui_gui: GUI-Mode UI, version 4.0
Target manager tgt_hpux_pa: HP-UX PA-RISC, version 4.0 DBGK 4.0
Object manager obj_som_som: HP SOM, version 4.0
Language manager lang_c: ANSI C, version 4.0
```

In the output of the preceding `version` command, the language manager title is `ANSI C`. The language type name is `lang_c`.

Description
Briefly describes what the manager does and does not support.

Syntax Describes the syntax of debugger command arguments such as *expression*, *declaration*, and *address* that vary among the managers. The *manager_option* argument represents options you can specify with the `property flags` command to change the behavior of a manager.

Currently, the language managers do not offer any options through the *manager_option* argument.

You can use type names or synonyms wherever the syntax of a debugger command calls for a type, such as *language_type*. For example, the following `property` command specifies `pc` (a synonym for the type name `lang_pas_hp`) as the *language_type*:

```
property language pc
```

You can create a new synonym for a language type name by creating a link to the language type name in the directory `/opt/langtools/dde/lang`. For example, to make `p` a valid language type, enter the following commands (as superuser):

```
cd /opt/langtools/dde/lang
ln -s lang_pas_hp p
```

Startup File Describes the startup file that executes when the debugger loads the manager. The language managers do not use startup files.

Related Managers Lists other types of managers related to this manager.

Names, titles, and syntax specific to the language managers are described in the following sections.

C Language Manager

Type Name

`lang_c`

Title

ANSI C

Description

In general, this manager supports the C expressions and declarations described in the C language reference manual for your system, plus the debugging extensions listed under "Syntax."

Syntax

- *declaration*

  Declaration of a type or variable, restricted to the following syntax:

  type_ref id, ...

  type_ref id[*bounds*]

  `typedef` *type_ref id*

  type_ref:
  ```
  char
  wchar_t
  int
  long
  long long
  short
  float
  double
  long double
  void
  struct id
  union id
  ```

enum *id*
type_ref *
typeof(*identifier*)
unsigned
signed
id

- *expression*

A C expression constructed from the comments, operators, identifiers, constants, type cast, and extensions listed in this section.

Comments

/* *comment* */

Operators

```
Arithmetic              +, -, *, /, %
Increment/decrement     ++, --
Relational              <, <=, >, >=, ==, !=
Logical                 &&, ||, !
Bitwise logical         &, ^, |, ~
Shift                   <<, >>
Assignment              =, +=, -=, *=, /=, %=, |=, ^=, &=, >>=, <<=
Address                 *, &
Conditional             ?:
Sequential evaluation   ,
Size                    sizeof
Type conversion         (type)
Array indexing          [,]
Member reference        ->, .
Grouping                ()
```

Identifiers

Identifiers are case sensitive and start with a dollar sign ($), an underscore (_), or a letter (ISO Latin-1 decimal values 65-90 and 97-122); subsequent characters can be a dollar sign ($), an underscore (_), a letter, or a digit (0-9). Use of the dollar sign is an extension to C.

Constants

Integer	*digits*[L | l | U | u] (decimal) 0x*digits*[L | l | U | u] (hexadecimal) 0*digits*[L | l | U | u] (octal) For example, 0xFu specifies an unsigned hexadecimal number. For specifying a long long, any combination of two L's is acceptable (LL, ll, Ll, lL).
Float	*digits*.*digits*[{E |e}[+ | -]*digits*][F | f | L | l] *digits*.[{E | e}[+ | -]*digits*][F | f | L | l] .*digits*[{E | e}[+ | -]*digits*][F | f | L | l] For example, 2.0e4L specifies a number of type long double.
String	"*string*" '*character*' '*nnn*'
Wide string	L"*string*" L'*character*' L'*nnn*'

Type Cast

(*type_name*)*expression*

Debugger Extensions

*location**identifier*	Name *identifier* visible from the scope *location*
`va(*address*)	Virtual address *address*
`long(*expression*)	Cast the resulting value of *expression* to type long
`longlong(*expression*)	Cast the resulting value of *expression* to type long long
`short(*expression*)	Cast the resulting value of *expression* to type short

`typeof(`*identifier*`)`	The type of *identifier*; can be used to refer to anonymous types
Array slices	In place of an array subscript, you can give a range of elements:

`*`	Lower bound to upper bound
expression`..`*expression*	Given range
`*..`*expression*	Lower bound to given *expression*
expression`..*`	Given *expression* to upper bound

Example: `print table[*, 100..110]`

- *language_type*

  { `lang_c` | `c` | `cc` }

- *manager_option*

  None.

Startup File

None.

Related Managers

None.

C++ Language Manager

Type Name

`lang_c++`

Title

C++

Description

In general, this manager supports the C++ expressions and declarations described in your C++ reference manual, and the debugger extensions listed under "Syntax."

Function overloading is permitted, and modules may contain identically named functions. Identically named functions are specified by name and parameters that identify them using the syntax

`amb(*function_name*,*parameter_type*,*parameter_type*,...)

where *function_name* is the name of the function and *parameter_type* is a list of the data types of the function's parameters, separated by commas. You may list all of the parameters; alternatively, you may list as many *parameter_type* arguments as you need to identify the function or functions you are referring to, and use ellipses (...) to indicate that the function may take additional parameters. Use

`amb(*function_name*)

to specify a version of the overloaded function that takes no parameters. Use

`amb(*function_name*, ...)

to specify all versions of the overloaded function.

For example,

```
`amb(sum,int,int)
```

refers to a function `sum` in the current environment that takes two integer parameters.

```
`amb(sum,int,int,...)
```

refers to all functions named `sum` in the current environment that take at least two parameters, the first two being integers.

Also note that, in the examples above, the syntax relates to the use of `` `amb `` to specify a location. You can also use `` `amb `` as a part of a qualified name to identify objects or variables. An example of the syntax is:

module\\`amb(*function_name*,*parameter_type*,...)*object_name*

The language manager implements overload resolution rules appropriate to C++ to resolve ambiguity. Overload resolution considers built-in type conversions but not user-defined type conversions.

The debugger does not execute implicit references to user-defined type-conversion operators.

The debugger supports standard operator syntax for user-defined operators (this does *not* include the `new`, `delete`, `++`,`--`,`()`, `->`, and `->*` operators). For example, if the user defines a `+` operator for a class, the command

```
print a + b
```

on two operands of that class invokes the user-defined `+` operator. However, you can also invoke the operator (including the `new`, `delete`, `++`,`--`,`()`, `->`, and `->*` operators) using the following member function call syntax:

```
print a.operator+(b)
```

Syntax

- *declaration*

  A declaration of a type or variable, restricted to the following syntax:

  type_ref id , ...

  type_ref id [*bounds*]

  `typedef` *type_ref id*

  type_ref: `char`
  `wchar_t`
  `int`

`long`
`short`
`float`
`double`
`long double`
`void`
`class` *id*
`struct` *id*
`union` *id*
`enum` *id*
type_ref `*`
`typeof(`*identifier*`)`
`unsigned`
`signed`
id

The following are also available with the HP ANSI CC compiler:

`bool`
`long long`

- *expression*

  A C++ expression constructed from the comments, operators, identifiers, constants, and extensions listed in this section.

  Comments

  `/*` *comment* `*/`
  `//` *comment*

  Operators

Arithmetic	`+`, `-`, `*`, `/`, `%`
Increment/decrement	`++`, `--`
Relational	`<`, `<=`, `>`, `>=`, `==`, `!=`
Logical	`&&`, `||`, `!`
Bitwise logical	`&`, `^`, `|`, `~`
Shift	`<<`, `>>`
Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `|=`, `^=`, `&=`, `>>=`, `<<=`
Address	`*`, `&`

Conditional	`?:`
Sequential evaluation	`,`
Size	`sizeof`
Type conversion	(*type*) or *simple_type*(*expr*)
Array indexing	`[,]`
Member reference	`->`, `.`, `.*`, `->*`
Grouping	`()`
Scope resolution	`::`*identifier*

The following operators are available with the HP ANSI CC compiler:

Dynamic cast	`dynamic_cast`<*type*>(*expr*)
Type identification	`typeid`(*expr*)

Identifiers

Identifiers are case sensitive and start with a letter (ISO Latin-1 decimal values 65-90 and 97-122), a dollar sign (`$`), a tilde (`~`), a colon (`:`), or an underscore (`_`); any additional characters can be a letter, digit (`0-9`), dollar sign (`$`), underscore (`_`), a tilde (`~`), a colon (`:`), or an operator symbol. Use of the dollar sign, tilde, and colon are extensions to C++.

Constants

Integer

digits[`L` | `l` | `U` | `u`] (decimal)
`0x`*digits*[`L` | `l` | `U` | `u`] (hexadecimal)
`0`*digits*[`L` | `l` | `U` | `u`] (octal)

For example, `0xFu` specifies an unsigned hexadecimal number.

Float

digits`.`*digits*[{`E` | `e`}[`+` | `-`]*digits*][`F` | `f` | `L` | `l`]
digits`.`[{`E` | `e`}[`+` | `-`]*digits*][`F` | `f` | `L` | `l`]
`.`*digits*[{`E` | `e`}[`+` | `-`]*digits*][`F` | `f` | `L` | `l`]

For example, `2.0e4L` specifies a number of type `long double`.

String

`"`*string*`"`
`'`*character*`'`
`'\`*nnn*`'`

Wide string	`L"`*string*`"`
	`L'`*character*`'`
	`L'\`*nnn*`'`

Debugger Extensions

location`\`*identifier*	Name *identifier* visible from the scope *location*
`` `va( ``*address*`)`	Virtual address *address*
`` `amb( ``*function_name*`,` *parameter_type*`,` *parameter_type*`, ... )`	Use the list of parameter types (*parameter_type*) to uniquely identify the overloaded function *function_name*
`` `long( ``*expression*`)`	Cast the resulting value of *expression* to type `long`
`` `short( ``*expression*`)`	Cast the resulting value of *expression* to type `short`
`typeof (`*identifier*`)`	The type of *identifier*; can be used to refer to anonymous types
Array slices	In place of an array subscript, you can give a range of elements:

`*`	Lower bound to upper bound
expression`..`*expression*	Given range
`*..`*expression*	Lower bound to given *expression*
expression`..*`	Given *expression* to upper bound

Example: `print table[*, 100..110]`

- *language_type*

{ `lang_c++` | `c++` | `CC` | `ccxx` }

- *manager_option*

None.

Startup File

None.

Related Managers

None.

FORTRAN Language Manager

Type Name

`lang_ftn`

Title

FORTRAN

Description

In general, this manager supports the FORTRAN expressions and declarations described in the FORTRAN language reference manual for your system, plus the debugging extensions listed under "Syntax."

Syntax

- *declaration*

  A declaration of a variable, restricted to the following syntax:

  [`external`] *type_name* {*identifier*[(*integer*)]}, ...

  type_name:
  ```
  INTEGER
  INTEGER*2
  INTEGER*4
  INTEGER*8
  REAL
  REAL*4
  REAL*8
  REAL*16
  DOUBLE PRECISION
  COMPLEX
  COMPLEX*8
  DOUBLE COMPLEX
  COMPLEX*16
  LOGICAL
  ```

```
LOGICAL*1
LOGICAL*2
LOGICAL*4
BYTE
CHARACTER
CHARACTER*nnn
```

A declaration of a record variable whose form was previously declared in a `STRUCTURE` statement:

`RECORD/`*struc_name*`/[`*variable_name* `|` *array_name* `|` *array_declarator*`]`

A declaration of a derived type whose form was previously declared in a `TYPE` definition:

`TYPE(`*dtype_name*`)[,` *attribute_list* `::]` *entity_list*

- *expression*

A FORTRAN expression constructed from the operators, identifiers, constants, intrinsic functions, and extensions listed in this section.

Operators

Arithmetic	`+`, `-`, `*`, `/`, `**`
Relational	`.eq.`, `.ne.`, `.lt.`, `.le.`, `.gt.`, `.ge.`
Logical	`.not.`, `.or.`, `.and.`, `.eqv.`, `.neqv.`
String	`//`, `(:)`
Assignment	`=`
Array indexing	`(,)`
Grouping	`()`
Member reference	`.`, `%`
Declaration	`::`

Identifiers

Identifiers are not case sensitive and must start with a letter (ISO Latin-1 decimal values 65-90 and 97-122); any additional characters can be a letter, digit (`0`-`9`), dollar sign (`$`), or underscore (`_`).

Constants

Integer	*digits* (decimal) [*radix*#]*digits*[_*KS*](*radix* = 2, 8, 10, 16; *KS* = 1, 2, 4, 8) For example, `16#1c6` specifies a hexadecimal integer.
Real	*digits* . *digits*[_*KS*][E[+ | -]*digits*](*KS* = 4,8,16) *digits* . *digits*[{D | Q}[+ | -]*digits*] *digits* .[_*KS*][E[+ | -]*digits*](*KS* = 4,8,16) *digits* .[{D | Q}[+ | -]*digits*] . *digits*[_*KS*][E[+ | -]*digits*] . *digits*[{D | Q}[+ | -]*digits*] For example, `3.5D0` specifies a double-precision number.
Complex	(*number* , *number*)
Logical	.TRUE. .FALSE.
Character string	' *characters* '

Intrinsic Functions

```
ABS
AIMAG
AINT
AMOD
ANINT
CHAR
CMPLX
DABS
DBLE
DINT
DMOD
DNINT
EXP
FLOAT
IABS
ICHAR
IDINT
```

```
IDNINT
IFIX
INT
LOC
MOD
NINT
REAL
SNGL
```

Debugger Extensions

`iaddr()`	Address
*location**identifier*	Name *identifier* visible from the scope *location*
‘`va(`*address*`)`	Virtual address *address*
`(`*typename*`)` *expression*	Type conversion
‘`long(`*expression*`)`	Cast the resulting value of *expression* to type `INTEGER*4`
‘`short(`*expression*`)`	Cast the resulting value of *expression* to type `INTEGER*2`
Array slices	In place of an array subscript, you can give a range of elements:

:	Lower bound to upper bound
expr1 : *expr2*	Given range
: *expr*	Lower bound to given *expr*
expr :	Given *expr* to upper bound
: :	Lower bound to upper bound, stride of 1
: : *expr*	Lower bound to upper bound, stride of *expr*
expr1 : *expr2* :	Given range, stride of 1

expr1 : *expr2* : *expr3*	Given range, stride of *expr3*
: *expr2* : *expr3*	Lower bound to *expr2*, stride of *expr3*
expr1 : : *expr3*	Given *expr1* to upper bound, stride of *expr3*

Example: `print table(::, 30:60:5)`

- *language_type*

  { `lang_ftn` | `fortran` | `ftn` | `f77` | `f90` }

- *manager_option*

  None.

Startup File

None.

Related Managers

None.

HP Pascal Language Manager

Type Name

`lang_pas_hp`

Title

HP Pascal

Description

In general, this manager supports a subset of the Pascal expressions and declarations described in the HP Pascal reference manual for your system, plus the debugger extensions listed under "Syntax."

Syntax

- *declaration*

  A declaration of a type or variable, restricted to the following syntax:

  `type` *identifier*[, *identifier*] ... `=` *type_reference*

  `var` *identifier*[, *identifier*] ... : *type_reference*

type_reference	`[`*location*`\]`*identifier*
	`shortint`
	`integer`
	`longint`
	`bit16`
	`bit32`
	`bit52`
	`bit64`
	`real`
	`single`
	`double`
	`longreal`
	`boolean`

```
char
string
globalanyptr
^type_reference
array [bound..bound, ...] of type_reference
set of type_reference
typeof([location\]identifier)
```

- *expression*

  A Pascal expression constructed from the comments, operators, identifiers, literals, predeclared functions, type cast, and extensions listed in this section.

  Comments

  `{` *comment* `}`
  `(*` *comment* `*)`
  `"`*comment*`"`

  Operators

Arithmetic	`+`, `-`, `*`, `/`, `div`, `mod`
Relational	`=`, `<>`, `<`, `>`, `<=`, `>=`, `in`
Boolean	`and`, `or`, `not`
Set	`+`, `*`, `-`
Assignment	`:=`
Array indexing	`[ , ]`
Field selection	`.`
Dereference	`^`
Set construction	`[ ... ]`
Grouping	`( )`

  Identifiers

  Identifiers are case-insensitive and must start with a letter (ISO Latin-1 decimal values 65-90 and 97-122); any additional characters can be letters, digits (`0-9`), or underscore (`_`).

Literals

Integer	*digits* (decimal)
Real	*digits* . *digits* *digits*[. *digits*]E[+ | -]*digits* For example, `5.48E-11` is a valid real number.
Character	'*character*'
String	'*characters*'
Boolean	`TRUE` `FALSE`
Pointer	`NIL`

Predeclared Functions

```
abs
addr
ord
round
sizeof
```

Type Cast

type_name(*expression*)

Debugger Extensions

*location**identifier*	Name *identifier* visible from the scope *location*
`va(`*address*`)	Virtual address *address*
`long(`*expression*`)	Cast the resulting value of *expression* to type `integer`
`longlong(`*expression*`)	Cast the resulting value of *expression* to type `longint`
`short(`*expression*`)	Cast the resulting value of *expression* to type `shortint`
^*type_name*(*expression*)	Cast the resulting value of *expression* to type ^*type_name*

`typeof` (*identifier*)	The type of *identifier*; can be used to refer to anonymous types
Array slices	In place of an array subscript, you can give a range of elements:

`*`	Lower bound to upper bound
expression..*expression*	Given range
`*`..*expression*	Lower bound to given *expression*
expression..`*`	Given *expression* to upper bound

Example: `print table[*, 100..110]`

- *language_type*

  { `lang_pas_hp` | `lang_pas` | `pascal` | `pc` | `hppas` | `hppascal` }

- *manager_option*

  None.

Startup File

None.

Related Managers

None.

HP-UX PA-RISC Assembly Language Manager

Type Name

`lang_asm_pa`

Title

HP-UX PA-RISC assembler

Description

This manager supports simple PA-RISC assembly language expressions as described in "Syntax."

Syntax

- *declaration*

  None.

- *expression*

  An assembly language expression constructed from the reserved names, constants, operators, and address expressions listed in this section.

  Reserved Names

`r0` - `r31`	General registers
`fr0l`, `fr0r`, `fr1l`, ... , `fr31l`, `fr31r`	Single-precision floating-point registers; these 64 registers correlate to the left and right halves of the double-precision floating-point registers having the corresponding numbers
`fr0` - `fr31`	Double-precision floating-point registers
`sr0` - `sr7`	Space registers
`cr0`, `cr8` - `cr31`	Control registers
`rctr`	Recovery counter
`pidr1` - `pidr4`	Protection identifiers
`ccr`	Coprocessor configuration register
`sar`	Shift amount register

`iva`	Interruption vector address
`eiem`	External interrupt enable mask
`itmr`	Interval timer
`pcsq`, `pcoq` (or `pcspace`, `npcspace`)	Interruption instruction address space and offset queues
`iir`, `isr`, `ior`	Interruption parameter registers
`ipsw`	Interruption processor status word
`eirr`	External interrupt request register
`tr0` - `tr7`, `ppda`, `hta`	Temporary registers (usable only by code executing at the most privileged level)
`rp`	Return link
`t1` - `t4`	Temporary registers
`arg0` - `arg3`	Argument words
`dp`	Data pointer
`ret0`	Return value
`ret1`, `sl`	Return value, static link
`sp`	Stack pointer
`mrp`	Millicode return link
`sret`, `sarg`	Return value, argument
`farg0` - `farg3`	Floating arguments
`fret`	Return value
`sflags`	Status flags

Constants

Integer
`B'`*digits* (binary) (for example, `B'100001111`)
`Q'`*digits* (octal) (for example, `Q'71035`)
digits (decimal) (for example, `9876`)
`D'`*digits* (decimal) (for example, `D'9876`)
`H'`*digits* (hexadecimal) (for example, `H'2F`)
`R'`*digits* (hexadecimal right) (for example, `R'2F` is `H'2F`)
`L'`*digits* (hexadecimal left shifted 11) (for example, `L'2F` is `H'17800`)

Float
digits[{`E` | `e`}[`+` | `-`]*digits*]
digits`.`[*digits*][{`E` | `e`}[`+` | `-`]*digits*]

For example, `5.48E-11` is a valid number.

Character
`'`*char*[*char*[*char*[*char*]]]`'`

Operators

<table>
<tr><td>Assignment</td><td>=</td></tr>
<tr><td>Arithmetic</td><td>+, -, *, /</td></tr>
<tr><td>Mod</td><td>%</td></tr>
<tr><td>Shift</td><td><<, >></td></tr>
<tr><td>Logical</td><td>&, |, ^, ~ (and, or, xor, not)</td></tr>
<tr><td>Grouping</td><td>{}</td></tr>
</table>

Address Expressions

[*location*]*identifier*	Name from source program
R*n*	Register direct
[*expression*]	Indirect
(*expression*)	Indirect
(*expression*, *expression*)	Add, then indirect
expression1(*expression2*)	Indirect, plus displacement (*expression1* must be a register or a constant)
expression.B	8-bit byte
expression.W	16-bit word
expression.L	32-bit longword

Hexadecimal is the default input and output radix.

- *language_type*

  { `lang_asm_pa` | `asm_pa` }

- *manager_option*

  None.

Startup File

None.

Related Managers

None.

Solaris SPARC Assembly Language Manager

Type Name

`lang_asm_sparc`

Title

SPARC assembler

Description

This manager supports simple assembly language expressions as described in "Syntax".

Syntax

- *declaration*

  None.

- *expression*

  An assembly language expression constructed from the reserved names, constants, operators, and address expressions listed in this section.

  Reserved Names

`r0` - `r31`	General-purpose registers
`g0` - `g7`	Global registers
`o0` - `o7`	Output registers
`i0` - `i7`	Input registers
`sp`	Stack pointer (`o6`)
`fp`	Frame pointer (`i6`)
`f0` - `f31`	Floating-point registers
`fsr`	Floating-point status register
`fq`	Floating-point deferred-trap queue
`pc`	Program counter
`npc`	Next program counter
`psw`	Processor status word

`y` Multiply/divide register
`wim` Window invalid mask
`tbr` Trap base register

Constants

Integer
digits (decimal) (for example, `9876`)
`0`*digits* (octal) (for example, `05376`)
`0x`*digits* (hexadecimal) (for example, `0x2F`)
`B'`*digits* (binary) (for example, `B'10001111`)
`Q'`*digits* (octal) (for example, `Q'71035`)
`O'`*digits* (octal) (for example, `O'71035`)
`D'`*digits* (decimal) (for example, `D'9876`)
`H'`*digits* (hexadecimal) (for example, `H'2F`)
`X'`*digits* (hexadecimal) (for example, `X'2F`)
`R'`*digits* (hexadecimal right) (for example, `R'2F` is `H'2F`)
`L'`*digits* (hexadecimal left shifted 10) (for example, `L'2F` is `H'BC00`)

Float
digits[{`E` | `e`}[`+` | `-`]*digits*]
digits`.`[*digits*][{`E` | `e`}[`+` | `-`]*digits*]

For example, `5.48E-11` is a valid number.

Character
`'`*char*[*char*[*char*[*char*]]]`'`

Operators

Assignment `=`
Arithmetic `+`,`-`, `*`, `/`
Mod `%`
Shift `<<`, `>>`
Logical `&`, `|`, `^`,`~` (and, or, xor, not)
Grouping `()`, `{}`
Unary `lo` (least significant 10 bits)
Unary `hi` (most significant 22 bits)

Address Expressions

<*identifier*> Name from source program
<*register_name*> Register direct
`[`*expression*`]` Indirect
expression`.B` 8 bits

expression.C	8 bits as a char
expression.H	16 bits
expression.L	32 bits

Hexadecimal is the default input and output radix.

- *language_type*

  { `lang_asm_sparc` | `asm_sparc` }

- *manager_option*

  None.

Startup File

None.

Related Managers

None.

C

Target Managers

This chapter describes the debugger target managers, which perform all operations that depend on the target hardware, operating system, or run-time environment.

On HP-UX systems, the HP-UX PA-RISC target manager is installed. On Solaris systems, the Solaris SPARC target manager is installed.

This chapter provides the following information about each manager:

Type Name
Identifies the name used for the manager as installed.

Title
Contains the name that the manager uses to identify itself. To list the titles of the managers currently loaded, use the `version` command. The following is an example of the output:

```
dde, version 4.0
User interface manager ui_gui: GUI-Mode UI, version 4.0
Target manager tgt_hpux_pa: HP-UX PA-RISC, version 4.0 DBGK 4.0
Object manager obj_som_som: HP SOM, version 4.0
Language manager lang_c: ANSI C, version 4.0
```

In the output of the preceding `version` command, the target manager title is `HP-UX PA-RISC`. The target type name is `tgt_hpux_pa`.

Description
Briefly describes what the manager does and does not support.

Syntax
Describes the syntax of debugger command arguments such as *expression*, *declaration*, and *address* that vary among the managers. The *manager_option* argument represents options you can specify with the `property flags` command to change the behavior of the manager.

You can create a new synonym for a target type name by creating a link to the target type name in the directory /opt/langtools/dde/tgt. For example, to make hppa a valid target type, enter the following commands (as superuser):

```
cd /opt/langtools/dde/tgt
ln -s tgt_hpux_pa hppa
```

Startup File Describes the startup file that executes when the debugger loads the manager.

Related Managers Lists other types of managers related to this manager.

Names, titles, and syntax specific to the target managers are described in the following sections.

HP-UX PA-RISC Target Manager

Type Name

tgt_hpux_pa

Title

HP-UX PA-RISC

Description

This manager supports the HP PA-RISC processors running HP-UX and the HP SOM (Spectrum Object Module) object file format.

Syntax

- *address*

  A hexadecimal integer or one of the following register names:

r0 - r31	General registers
fr0l, fr0r, fr1l, ... , fr31l, fr31r	Single-precision floating-point registers; these 64 registers correlate to the left and right halves of the double-precision floating-point registers having the corresponding numbers
fr0 - fr31	Double-precision floating-point registers
sr0 - sr7	Space registers
cr0 - cr31	Control registers
rctr	Recovery counter
pidr1 - pidr4	Protection identifiers
ccr	Coprocessor configuration register
sar	Shift amount register
iva	Interruption vector address
eiem	External interrupt enable mask
itmr	Interval timer
pcsq, pcoq (or pcspace, npcspace)	Interruption instruction address space and offset queues
iir, isr, ior	Interruption parameter registers

`ipsw`	Interruption processor status word
`eirr`	External interrupt request register
`tt0` - `tr7`, `ppda`, `hta`	Temporary registers (usable only by code executing at the most privileged level)
`rp`	Return link
`t1` - `t4`	Temporary registers
`arg0` - `arg3`	Argument words
`dp`	Data pointer
`ret0`	Return value
`ret1`, `sl`	Return value, static link
`sp`	Stack pointer
`mrp`	Millicode return link
`sret`, `sarg`	Return value, argument
`farg0` - `farg3`	Floating arguments
`fret`	Return value
`sflags`	Status flag

The register indirection operator, parentheses (`()`), allows you to refer to the contents of the contents of a register. This operator is particularly useful with the `dump` and `watchpoint` commands.

- *manager_option*

  The `tgt_shlib_debug` option to the `property flags` command allows you to set breakpoints in shared library routines. It even allows you to set breakpoints when the routines were compiled without debug information (in system libraries, for example). For more information about using this option and about debugging shared libraries, see "Debugging Shared Libraries" in Chapter 8.

- *process_id*

  A UNIX `pid` (integer).

- *program_invocation*

  A program invocation of the form *program_pathname* [*program_arguments*].

- *signal*

  A UNIX signal number (*integer* ≤ 32) or one of the following UNIX signal names (the numeric equivalent of each is also shown):

```
SIGHUP      01
SIGINT      02
SIGQUIT     03
SIGILL      04
SIGTRAP     05
SIGABRT     06
SIGEMT      07
SIGFPE      08
SIGKILL     09
SIGBUS      10
SIGSEGV     11
SIGSYS      12
SIGPIPE     13
SIGALRM     14
SIGTERM     15
SIGUSR1     16
SIGUSR2     17
SIGCHLD     18
SIGPWR      19
SIGVTALRM   20
SIGPROF     21
SIGIO       22
SIGWINCH    23
SIGSTOP     24
SIGTSTP     25
SIGCONT     26
SIGTTIN     27
SIGTTOU     28
SIGURG      29
SIGLOST     30
SIGRESERVE  31
SIGDIL      32
```

- *target_type*

  { `tgt_hpux_pa` | `pa` }

- *target_command*

  One of the following arguments to the command `target command`:

`dump proc` [`full`]	Prints a list of known target processes
`help`	Lists and describes the supported target commands

Startup File

The startup file `/opt/langtools/dde/tgt/tgt_hpux_pa.startup` defines the following macros:

`regs`	Dumps general registers
`cregs`	Dumps control registers
`fregs`	Dumps double-precision floating-point registers
`fsregs`	Dumps single-precision floating-point registers
`fdregs`	Dumps double-precision floating-point registers
`sregs`	Dumps space registers
`reg_update`	Monitors general registers for changes
`freg_update`	Monitors double-precision floating-point registers for changes
`fsreg_update`	Monitors single-precision floating-point registers for changes
`fdreg_update`	Monitors double-precision floating-point registers for changes
`‘asm`	Identifies the appropriate assembly language manager
`‘r0` - `‘r31`	Identify general registers
`‘pc`	Identifies the program counter
`‘sp`	Identifies the stack pointer register
`‘dp`	Identifies the data pointer register
`‘arg0` - `‘arg3`	Identify the argument word registers
`‘ret0`, `‘ret1`	Identify the return value registers

Related Managers

`lang_asm_pa`	Use to evaluate expressions using assembler syntax.
`obj_som`	Use to debug programs compiled in HP SOM format.

Solaris SPARC Target Manager

Type Name

`tgt_solaris_sparc`

Title

Solaris 2.*x* SPARC

Description

This manager supports SPARC processors running Solaris 2.*x* using the ELF object file format.

Syntax

- *address*

  A hexadecimal integer or one of the following register names:

`r0` - `r31`	General purpose registers
`g0` - `g7`	Global registers
`o0` - `o7`	Output registers
`i0` - `i7`	Input registers
`sp`	Stack pointer (`o6`)
`fp`	Frame pointer (`i6`)
`f0` - `f31`	Floating-point registers
`fsr`	Floating-point status register
`fq`	Floating-point deferred-trap queue
`pc`	Program counter
`npc`	Next program counter
`psw`	Processor status word
`y`	Multiply/divide register
`wim`	Window invalid mask
`tbr`	Trap base register

- *manager_option*

  None.

- *process_id*

  A UNIX `pid` (integer).

- *program_invocation*

  A program invocation of the form *program_path* [*program_arguments*].

- *signal*

  A UNIX signal number (integer) or one of the following UNIX signal names:

SIGHUP	SIGINT	SIGQUIT	SIGILL	SIGTRAP
SIGABRT	SIGEMT	SIGFPE	SIGKILL	SIGBUS
SIGSEGV	SIGSYS	SIGPIPE	SIGALRM	SIGTERM
SIGUSR1	SIGUSR2	SIGCHLD	SIGPWR	SIGWINCH
SIGURG	SIGIO	SIGSTOP	SIGTSTP	SIGCONT
SIGTTIN	SIGTTOU	SIGVTALRM	SIGPROF	SIGXCPU
SIGXFSZ	SIGWAITING	SIGLWP		

- *target_type*

  { `tgt_solaris_sparc` }

- *target_command*

  None.

Startup File

The startup file *dde_install_dir*`/tgt/tgt_solaris_sparc.startup` defines the following macros:

`regs`	Dumps registers
`fregs`	Dumps single-precision floating-point registers
`‘asm`	Identifies appropriate assembly language manager

Related Managers

`lang_asm_sparc`	Use to support simple assembly language expressions.
`obj_elfstabs`	Use to support ELF object format and STAB style debug information.

D

Object Managers

This chapter describes the debugger object managers, which allow the debugger to handle object and debug formats. The object managers process the debugging information in executable files.

On HP-UX systems, the debugger uses the HP SOM object manager. On Solaris systems, the debugger uses the Solaris SPARC object manager.

This chapter provides the following information about each manager:

Type Name
: Identifies the name used for the manager as installed.

Title
: Contains the name that the manager uses to identify itself. To list the titles of the managers currently loaded, use the `version` command. The following is an example of the output:

```
dde, version 4.0
User interface manager ui_gui: GUI-Mode UI, version 4.0
Target manager tgt_hpux_pa: HP-UX PA-RISC, version 4.0 DBGK 4.0
Object manager obj_som_som: HP SOM, version 4.0
Language manager lang_c: ANSI C, version 4.0
```

In the output of the preceding `version` command, the object manager title is `HP SOM`. The type name is `obj_som_som`.

Description
: Briefly describes what the manager does and does not support.

Syntax
: Describes the syntax of debugger command arguments such as *expression*, *declaration*, and *address* that vary among the managers. The *manager_option* argument represents options you can specify with the `property flags` command to change the behavior of a manager.

Currently, the object managers do not offer any options through the *manager_option* argument.

Startup File Describes the startup file that executes when the debugger loads the manager. The object managers do not use a startup file.

Related Managers Lists other types of managers related to this manager.

Names, titles, and syntax specific to the object managers are described in the following section.

HP SOM Object Manager

Type Name

`obj_som_som`

Title

HP SOM

Description

This manager supports the object format produced by the HP C, HP C++, HP FORTRAN, and HP Pascal, and the HP-UX PA-RISC assembler. The object and debug format is the Spectrum Object Module (SOM) format.

Syntax

- *object_type*

  { `obj_som_som` | `som` }

- *manager_option*

  None.

Startup File

None.

Related Manager

`tgt_hpux_pa`	Use to debug programs on HP PA-RISC machines running HP-UX.

D

Solaris SPARC Object Manager

Type Name

`obj_elfstabs`

Title

Solaris ELF with STABs

Description

This manager supports the ELF object format and STAB style debug information produced by the SunPro C and C++ compilers. Both the SC2.X and SC3.X versions are supported.

Syntax

- *object_type*

  { `obj_elfstabs` | `elfstabs` }

- *manager_option*

  None.

Startup File

None.

Related Manager

`tgt_solaris_sparc`	Use to support SPARC processors using the ELF object file format.

E

User Interface Managers

This chapter describes the managers that enable the debugger to provide different user interfaces. Consult this chapter for information specific to the user interface managers, such as the value of the *ui_type* argument to the `dde` command option `-ui`.

The debugger supports the graphical user interface, line-mode user interface, and SoftBench managers.

This chapter provides the following information about each manager:

Type Name Identifies the name used for the manager as installed.

Title Contains the name that the manager uses to identify itself. To list the titles of the managers currently loaded, use the `version` command. The following is an example of the output:

```
dde, version 4.0
User interface manager ui_gui: GUI-Mode UI, version 4.0
Target manager tgt_hpux_pa: HP-UX PA-RISC, version 4.0 DBGK 4.0
Object manager obj_som_som: HP SOM, version 4.0
Language manager lang_c: ANSI C, version 4.0
```

In the output of the preceding `version` command, the user interface manager title is `GUI-Mode UI`. The type name is `ui_gui`.

Description Briefly describes what the manager does and does not support.

Syntax Describes the syntax of debugger command arguments such as *expression*, *declaration*, and *address* that vary among the managers. The *manager_option* argument represents options you can specify with the

`property flags` command to change the behavior of the manager.

Currently, the user interface managers do not offer any options through the *manager_option* argument.

You can use type names or synonyms wherever the syntax of a debugger command calls for a type, such as *ui_type*. For example, the following `dde` command specifies `gui` as the *ui_type*:

```
dde -ui gui a.out
```

You can create a new synonym for a user interface type name by creating a link to the user interface type name in the directory `/opt/langtools/dde/ui`. For example, to make `mtf` a valid user interface type, enter the following commands (as superuser):

```
cd /opt/langtools/dde/ui
ln -s ui_gui mtf
```

You can change the default user interface for all users of your system by changing the link `/opt/langtools/dde/ui/default`. The default user interface is ordinarily set to `ui_gui`. For example, to make the line-mode interface the default, enter the following commands (as superuser):

```
cd /opt/langtools/dde/ui
rm default
ln -s ui_line default
```

Startup File Describes the startup file that executes when the debugger loads the manager.

Related Managers Lists other types of managers related to this manager.

Names, titles, and syntax specific to the user interface managers are described in the following sections.

Graphical User Interface Manager

Type Name

`ui_gui`

Title

GUI-Mode UI

Description

This manager supports the graphical user interface and the HP/DDE command language, both of which are described in the HP/DDE online help. In addition, this manager provides compatibility with the `dbx` command syntax through the macros in `/opt/langtools/dde/contrib/dbx_macros`.

Syntax

- *ui_type*

  { `ui_gui` | `gui` }

- *manager_option*

  None.

Startup File

The startup file `/opt/langtools/dde/ui/nls/C/ui_gui.startup` contains predefined key bindings, pop-up menus, and command buttons.

Related Managers

None.

Line-Mode User Interface Manager

Type Name

`ui_line`

Title

Line-Mode UI

Description

This manager supports the line-mode user interface (described in Appendix A) and the HP/DDE command language (described in the HP/DDE online help). In addition, this manager provides compatibility with the `dbx` command syntax through the macros in `/opt/langtools/dde/contrib/dbx_macros`.

Syntax

- *ui_type*

  { `ui_line` | `line` }

- *manager_option*

  None.

Startup File

The startup file `/opt/langtools/dde/ui/nls/C/ui_line.startup` creates predefined macros.

Related Managers

None.

SoftBench User Interface Manager

Type Name

`ui_softdde`

Title

Softbench Softdebug UI

Description

This manager supports the Softbench Softdebug tool. When using this UI manager, all communication (input, output, error) with HP/DDE is done via Softbench messaging technology. All communication (input, output, error) with the target program is also done via Softbench messaging. The message protocol for this manager can be found in the *softdebug*(1) man page.

The manager provides compatibility with the `dbx` command syntax through the macros in `/opt/langtools/dde/contrib/dbx_macros`.

Syntax

- *ui_type*

  { `ui_softdde` | `softdde` }

- *manager_option*

  None.

Startup File

The startup file `/opt/langtools/dde/ui/nls/C/ui_softdde.startup` creates predefined macros and sets several properties for softdebug behavior.

Glossary

accelerator

An OSF/Motif shortcut for choosing a selection from the **menu bar** or from a pull-down menu. An accelerator for a menu item is indicated by name after the menu item name. Unlike a **mnemonic**, an accelerator can invoke a **command** when the name of the command is not visible.

action list

A series of debugger **commands** that is associated with a specific **breakpoint**, **intercept**, **watchpoint**, or **trace** request. The debugger executes an action list after executing up to the breakpoint, encountering a program event, detecting a change in the watched value, or encountering a trace request.

activation

An instance of a **block** or variable created by the recursive invocation of a procedure or by multiple instances of a procedure in a multi-threaded application.

alias macro

See **macro**.

Alt

A key that, when used in combination with a **mnemonic**, opens a menu corresponding to a name on the **menu bar**. On a keyboard, the key may be [ALT], [Meta], or [Extend char].

ancestor

C and C++ only. Any function up the call chain from the current function. Ancestors include functions that either directly or indirectly call the current function.

annotation margin
An area next to the **Source File Area** that contains line numbers, breakpoint symbols, and an arrow that indicates the current program counter location.

archive library
A library that contains one or more object files and is created with the `ar` command. When linking an object file with an archive library, `ld` searches the library for global definitions that match up with external references in the object file. If a match is found, `ld` copies the object file containing the global definition from the library into the executable file. See **shared library**.

basic block
A sequence of statements (or their corresponding instructions) that contains no branches. A branch is a change in the flow of control, such as an `if`, `for`, or `do` statement. A branch begins a new basic block.

block
A program unit, such as a module, a main program, a subroutine, or a function. What constitutes a block depends on the language in which the program is written. A block defines and encloses a **scope**. The debugger also defines blocks called `'declared` (for user-declared symbols), `'predefined` (for data types for supported languages) and `'image` (for the program **images**).

block expression
A block name, a file name, a class name, a template name, or a class object reference. For C++, a block expression can be any valid C++ expression that can be evaluated to one or more executable addresses or to a class type.

block qualified name
A format used by the debugger to refer to variables that are not visible from the current **environment**. A block qualified name explicitly identifies the **block** enclosing the object and the object's name; the format is *block**object_name*.

breakpoint
A **monitor** that, when encountered during program execution, stops execution and transfers control back to the debugger. A breakpoint is always associated with a particular address, which is either specified explicitly or implied by its association with a **location**.

buffer
An area in memory used as a temporary storage area.

button
In OSF/Motif, an icon, usually accessible by a mouse pointer, that starts an action. Radio buttons come in sets of two or more, each button representing a mutually exclusive selection.

check box
In OSF/Motif, a square box on a **dialog box**. You may choose any number of items with check boxes.

command
Commands tell the debugger which functions to perform. They can be spelled out or abbreviated. The abbreviation for most commands is the first three letters of each word in its name. Commands are terminated by a newline.

command list
One or more debugger **commands** separated by semicolons (;) and enclosed in square brackets, braces, or parentheses. A command list lets you combine commands that require an EOL as a terminator with commands that are terminated with a semicolon on the command line.

continuation character
A backslash character (\), which, when placed at the end of a command line, allows you to continue the **command** on the next line.

core file
A memory image of an abnormally terminated process (see *core*(4)). This file contains sufficient information to determine what the process was doing at the time of its termination. It can be examined using the debugger to determine why the program failed.

critical point
The instruction in the object code at which the most important action accomplished by a given source statement takes place. The critical point instruction is usually the instruction where a data value may change. Not every source code statement has a critical point; some statements have more than one critical point. Statements likely to have a critical point include assignment statements and function calls.

critical points location mapping
In optimized code, a **location mapping** that maps a source statement to the **critical point** for that statement.

current environment
See **environment**.

Current Location
The location at which the debugger's attention is currently focused. The debugger uses the Current Location in the source code to determine, among other things, what source code to display and what **block** to use as the debugger's **scope**. The Current Location is also called the current **environment**.

current point of execution
Program **location** where execution stopped. The statement at this location will, by default, be the next statement to be executed when execution resumes. Same as **PC Location**.

cursor
The insertion point for text. The default cursor is a blinking vertical bar (on color displays) or a double wedge (on monochrome displays).

Also called the text cursor.

dead code
One or more lines of source code that the optimizer has eliminated from the executable program because it is never actually executed.

Debugger Output Area
An area of the **main debugger window** that displays debugger commands and debugger messages.

debugging information

Name, type, source file, and source-line-to-address mapping information generated by the compiler for use by the debugger. This information can significantly increase the size of an executable file. All debugger information is preprocessed (and reduced in size) when the program is linked.

‘declared block

Outer debugger **block** used to contain definitions of user-declared symbol names. You may explicitly refer to symbol names created using the `declare` command by using a **block qualified name**.

define macro

See **macro**.

delay slot

The instruction following a branch instruction. In PA-RISC assembly language, a branch instruction is paired with the instruction that follows it, and both are executed simultaneously. When you step through assembly code, this means that when you reach the branch instruction (the `BL` instruction in the following example), you do not immediately step into the routine `read_sequence`. Instead, you step to the next instruction (`LDO`), and the next time you step, both the branch and the load occur.

```
BL              read_sequence,%r2
LDO             -272(%r30),%r24
```

dialog box

In the OSF/Motif user interface, an interactive pop-up containing command options. Generated when a **command** is selected from a menu, a dialog box provides **check boxes**, input fields, and **buttons** that you use to select options, enter arguments, and execute or cancel the command.

double-click

Click the mouse button twice in rapid succession.

drag

To press and hold down a mouse button while moving the mouse.

environment
A concept used by the debugger to determine what source code to display, what language to use when parsing **expressions**, and what **blocks** to search for objects. The run environment is the block containing the **current point of execution**. The current environment (also called the **Current Location**) is the run environment by default, but you can use the `environment` command to change the current environment to any block within the program.

environment variable
A named variable that is passed to all processes created by the current shell. See your shell reference page *sh*(1), *ksh*(1) or *csh*(1) for information on setting and reading environment variables.

explicitly loaded shared library
A **shared library** that is loaded by a call within your program to *shl_load*(3X) (on HP-UX systems) or *dlopen*(3X) (on Solaris systems).

expression
A valid combination of data object names, language operators, and constant numeric values. Every expression is evaluated and reduced to a single value.

fragment
In optimized code, a set of contiguous instructions in the object code that all derive from the same statement in the source code. See **location mapping**.

fragments location mapping
In optimized code, a **location mapping** that maps a source statement to the **fragment** or fragments in the object code that correspond to that statement.

frame block qualified name
A format used by the debugger to identify dynamically activated symbols within recursive procedures, by using both a **block** and a frame to specify the current **environment**.

fully qualified name
A format used by the debugger to identify all **blocks** that enclose the object. A fully qualified name explicitly identifies the module

and block enclosing the object and the object's name; the format is
*module**block**object_name*.

HP Help System

The online help system provided on HP-UX systems. You can obtain online help for the debugger by selecting the "Help" menu item. If you are not using the debugger, you can obtain help by clicking on the bookshelf icon on the front panel, then clicking on the Top Level icon to bring up the Help Manager window, and then selecting the debugger help.

identifier

A sequence of characters that represents an entity such as a function or data object.

image

A loaded executable file or **shared library**.

image qualified name

A format used by the debugger to eliminate ambiguity among module names when an application consists of more than one executable **image**. An image qualified name explicitly identifies the image, module, and **block** enclosing the object and the object's name; the format is
`image(*image_name*)*module**block**object_name*.

implicitly loaded shared library

A **shared library** that is linked against your program.

indirect pointers

Pointers that address other pointers. Linked lists often use chains of pointers in their list structure.

Input Box

A text area that can accept typed keystrokes.

The "( ):" Input Box is a text area in the **main debugger window** that provides input to some command buttons (such as [Print ()]) and many pull-down menu commands (such as "Break: Set at ( )" and "Visit: Line ( )"). You enter information in this Input Box either by typing or by selecting text in the source or I/O areas for use with these commands. Text can be selected either by dragging the mouse or by double-clicking.

The "Debugger Input:" Input Box is a text area in the **main debugger window** in which you enter commands to the debugger from the keyboard.

intercept
A **monitor** that stops program execution when a specified program event occurs. Events that can be intercepted include operating-system signals, the loading or removing of images from a program's address space, and the termination of the program. By default, all operating-system signals are intercepted.

location
A unique position in the user program. It can be specified as a file name, procedure name, **source line** number, or combination of these. An address can also be used to specify a location for certain **commands**.

location mapping
A method of defining the relationship between the source code and the object code. When you debug optimized code, you can choose any of three location mappings. The **source statement order location mapping** (the default) maps a statement to an instruction so as to follow the order of statements in the source. The **critical points location mapping** maps a statement to the instruction at which some data value may change. The **fragments location mapping** maps each source statement to the first set of contiguous instructions that corresponds to it.

macro
An identifier (optionally with arguments) that is defined as a substitute for **commands** or for text strings. The debugger supports both alias macros, which substitute for commands, and define macros, which substitute for any string of text, including commands.

main debugger window
The window that contains the main **menu bar**, the "( ):" **Input Box**, the **Source File Area**, the **Debugger Output Area**, the "Debugger Input:" **Input Box**, and the **User Program I/O Area**.

menu bar
In OSF/Motif, an area at the top of a **window** that contains the titles of the pull-down menus for that application.

mnemonic
In OSF/Motif, an underscored letter in a menu name. You can bring up the associated menu by pressing **Alt** and the mnemonic.

monitor
A software "trigger," such as a breakpoint, that interrupts **target program** execution and optionally describes the state of the target program after the interrupt. See **breakpoint**, **trace**, **watchpoint**, and **intercept**.

mouse button
A key on the mouse that has some action associated with it when clicked. On a two-button mouse, middle button commands can be accessed by pressing both buttons at once.

mouse cursor
See **pointer**.

path map
A specification of the replacement of a previous path argument to the `property sdir` command by a new path argument. The new path argument replaces the old one as a directory prefix in the source file directory search path.

PC Location
Program **location** where execution stopped. The statement at this location will, by default, be the next statement to be executed when execution resumes. Same as **current point of execution**.

pointer
The screen object that tracks mouse movement. This is usually an arrow, but it can take other forms (such as `?`) to indicate specific applicable areas. It may appear as a small hourglass to indicate that an application is busy or as something resembling a sonar wave to indicate that a request message is being processed.

Also called the mouse cursor.

process ID (pid)
A unique identification number assigned to all processes by the operating system.

qualified name
The name of a variable, specified in a format that allows you to refer to variables in any **scope**. See **block qualified name**, **frame block qualified name**, **fully qualified name**, and **image qualified name**.

radio buttons
An X construct consisting of several buttons representing several choices. Only one button may be selected at a time. When a button is selected, all other buttons are automatically deselected.

registers
Hardware registers. Most of these are directly accessible by the debugger through symbolic names (for example, `fr0`). Many registers have special meaning; some cannot be modified by the debugger user. Actual modification of hardware registers should not normally be necessary while debugging. Correct program execution depends highly on registers and their contents.

resource
A component of the X Window System resource data base. Resources control the appearance and behavior of parts of the system.

run environment
See **environment**.

scope
The region of source code over which a name's declaration is active.

scroll bar
In OSF/Motif or the X Window System, a graphical device used to scroll data displayed in a **window**. A scroll bar consists of a slider, scroll area, and scroll arrows.

shared library
Like an **archive library**, a shared library contains relocatable object code. However, `ld` treats shared libraries quite differently from archive libraries. When linking an object file with a shared library, `ld` does not copy object code from the library into the executable file; instead, the linker simply notes in the executable file that the code calls a routine in the shared

library. The actual linkage does not occur until the program is run. Shared libraries can be **implicitly** or **explicitly** loaded.

shell
An HP-UX command interpreter (Bourne, Korn, Key, Posix or C), providing a working environment interface for the user. The shell takes command input from the keyboard and interprets it for the operating system.

signal
A software interrupt sent from the operating system to a program. This can inform the program of any asynchronous event. Signals are used for segment violation, divide by zero, or other hardware problems; they can also be sent as a job control mechanism (stop, continue, kill).

source
Source text (files) used to compile the user program. Source files can be in any of the programming languages supported by the debugger.

Source File Area
A text area in the **main debugger window** where a program source file is displayed.

source line
A single line of text in a source file, denoted by a line number. A source line may or may not contain actual executable statements. Conversely, more than one statement can occur on a single line.

source statement order location mapping
In optimized code, a **location mapping** that maps statements to instructions in a way that follows the order of the statements in the source code as closely as possible, even though the actual order of instruction execution does not follow that order. By default, the debugger uses this location mapping.

stack
A linear data structure maintained by the user program for management of local data and flow of control during procedure calls. Each sequential region on the stack embodies information about a particular procedure. The preceding region (frame) describes its caller. At any point during execution,

a stack trace (generated by the `tb` command) displays information contained in each stack frame.

standard input
The source of input data for a program. The standard input file is often called `stdin`, and is automatically opened by the **shell** for reading on file descriptor 0 for every command invoked.

standard output
The destination of output data from a program. The standard output file is often called `stdout`, and is automatically opened by the **shell** for writing on file descriptor 1 for every command invoked. Standard output appears on the display unless it is redirected otherwise.

startup files
Files containing **commands** that specify the user interface, the target manager, and the debugging environment. At invocation, the debugger reads three startup files: a user interface startup file, a personal startup file, and a target manager startup file.

stub
Stubs are short code segments that may be inserted into procedure calling sequences by the PA-RISC linker. Stubs are used for very specific purposes, such as inter-space (for example, **shared library**) calls, long branches, and preserving calling interfaces across modules (for example, parameter relocation). For more information on stubs, see the *Procedure Calling Conventions Reference Manual*.

target program
The program that is currently being debugged.

text cursor
See **cursor**.

thread
A single flow of control in a process. A process may have multiple threads capable of executing at any time.

The HP/DDE debugger distinguishes between the current thread and the selected thread. The current thread is the thread that the debugger refers

to when evaluating expressions. The selected thread is the thread that was running when execution stopped. the selected thread differs from the current thread when you change environments with the thread specifier `` `thread(n) ``.

trace
A **monitor** that stops execution momentarily, reports the current program location, then continues **target program** execution. You can specify whether a trace is in effect at every source statement, at every instruction, or only at routine entry and/or exit points.

typed pointers
Pointers that have been declared with a specific type; for example, a pointer in C may be declared as a pointer to `char` and used for characters and strings.

unnamed block
A **block** generated by the compiler, usually to enclose a new lexical **scope** that has no associated name. In a C++ program, it usually holds variables with declaration statement scope rather than function or class scope. In a C program, it usually holds variables defined in a block below that of the function.

user interface
The medium through which users communicate with their workstations or with an application.

User Program I/O Area
A text area in the **main debugger window**. Programs being debugged send their output to this area and take their input from this area. Move the mouse pointer into this area to enter text into the standard input of your program.

watchpoint
A **monitor** that monitors a selected variable or address range and reports the value of the variable or address range only when that value changes. As with **traces**, you can specify whether a watchpoint is in effect at every source statement, at every instruction, or only at routine entry and/or exit points.

window

A frame-defined, rectangular area of the screen used by the X Window System to contain a particular application or a command line. Windows can be moved, resized, iconized, and manipulated.

Index

Special characters

"", 5-10
(), 5-2, 5-3, 5-10
():, 1-2
,, 5-3
., 5-10, 6-9
;, 5-2
<, 5-7
>, 5-8
>>, 5-8
2>, 5-8
2>>, 5-8
>>?, 5-8
[ ], 5-2, 5-3, 5-10
\, 5-3, 7-8
\\, 7-8
‘, 5-10
{}, 5-2, 5-3, 5-10

A

abbreviating debugger commands, 5-2
action lists, 5-15
  conditional, 5-16
  creating, 5-15, 5-16
  errors in, 5-16
  executing, 5-16
  multiple, 5-16
`activate intercepts` command, 3-20
`Add Source Directories`
  `File` menu choice, 2-12
`‘after_debug` macro, 5-11, 6-4
`‘after_fault` macro, 5-11
`alias` command, 5-9
aliases. *See* macros
`‘amb` reserved identifier, 5-11, B-7
angle brackets
  2>> (appending standard error), 5-8
  2> (redirecting standard error), 5-8
  >>? (appending standard error), 5-8
  >> (appending standard output), 5-8
  < (redirecting standard input), 5-7
  > (redirecting standard output), 5-8
annotation margin, 1-4, 8-13
`‘arg0 - ‘arg3` macros, C-6
`args` command, 4-4
arguments
  displaying, 4-4
argument word registers, C-6
arrays, 4-5
`‘asm` macro, 5-11, C-6, C-8
`Assembly Instructions`
  `Show` menu choice, 2-16, 8-13
Assembly Instructions dialog box, 8-13, 8-14
assembly language code
  debugging, 8-13
  saving to a file, 8-16
assembly language manager
  HP-UX PA-RISC, B-22
  identifying, C-8
  SPARC, B-25
  specifying, C-6
attaching to a running process, 2-7
`average`, sample program, 2-2

B

backslash
  double (\\), 7-8
  single (\), 5-3, 7-8
basic debugging tasks, 1-6
block qualified names, 7-6
blocks, 7-2
  inactive, 7-10
  scope and visibility, 7-4
braces ({}), 5-2, 5-3, 5-10
brackets ([]), 5-2, 5-3, 5-10
`Break` menu, 3-5
  `Set`, 3-6
  `Show`, 3-8
`breakpoint` command, 3-3
  `-after` option, 3-5
  `-do` option, 3-4
  `-in` option, 3-4
breakpoints, 3-2. *See also* monitors
  action lists and, 5-15
  in alternate source files , 3-4
  assembly instructions, 8-13
  at blocks or routines , 3-4
  `Break` menu, 3-5
  listing, 3-8
  in loops , 3-5
  preserving, 2-9
  setting, 3-3
  setting with the mouse, 3-3
  specifying actions, 3-4
  specifying locations, 3-4
  symbol, 3-3
  threads, 8-9
Breakpoint Set/Change dialog box, 3-6
broken pipe errors, 8-24
buffers, 4-8
buttons
  command, 1-4
  interrupt, 1-2
  location, 1-2

C

call/return stack
  examining, 2-18
case sensitivity, 5-4
changing current environment, 7-3
changing current language, 7-4
changing environment variables, 2-5
child processes, 8-22
C++ language manager, B-7
C language manager, B-3
code, assembly language, 8-13
comma (`,`), 5-3
command buttons, 1-4
  changing, 6-2
command input box, 1-4
`Command Input Mode`, 5-5
command line
  conventions, 5-1
  editors, 5-5
  syntax conflicts, 5-3
command lists, 5-2
`‘command` macro, 5-11
command reference, 1-6
commands
  abbreviating, 5-2
  action lists, 5-15
  `activate intercepts`, 3-20
  `alias`, 5-9
  `args`, 4-4
  `breakpoint`, 3-3
  command lists, 5-2
  continuing on next line, 5-3
  `dde`, 2-3, 2-4
  `debug`, 2-6, 2-7, 8-2
  `declare`, 4-4, 7-10
  `define`, 5-9
  `delete intercepts`, 3-20
  `describe`, 4-8
  `dump`, 4-8, 4-12, 8-15
  `environment`, 2-7, 2-12, 2-19, 7-3, 7-12

- executing after invoking target program, 6-4
- `free`, 2-9
- `go`, 2-15, 5-17
- history, 5-5
- `if`, 5-16
- `initialize -altdbinfo`, 2-2, 8-4
- `input`, 5-7
- `intercept`, 3-20, 8-12
- `intercept load`, 8-4
- `kill`, 2-9
- `list blocks`, 7-9
- `list images`, 8-5
- `list intercepts`, 3-20
- `list threads`, 8-9
- multiple on one line, 5-2
- `print`, 4-4, 4-8
- `property abort`, 5-7
- `property array_dim_max`, 4-6
- `property flags tgt_shlib_debug`, 8-4, C-4
- `property fork`, 8-22
- `property language`, 5-4, 7-4
- `property libraries`, 8-4
- `property qual_max`, 4-2, 7-8
- `property record`, 5-6
- `property sdir`, 2-2, 2-12
- `pxdb`, 2-7
- `quit`, 2-3
- register commands, 4-12
- `restart`, 2-9
- `shell`, 5-7
- `step`, 2-16, 5-17
- `suspend intercepts`, 3-20
- `tb`, 2-7, 2-19
- terminating, 5-2
- `thread`, 8-11
- `trace`, 3-14
- `use source`, 2-2
- `watchpoint`, 3-13

common debugging tasks, 1-6
compatibility with `dbx`. *See* `dbx` debugger
compatibility with `xdb`. *See* `xdb` debugger
compiling code for debugging, 2-2
conditional action lists, 5-16
continuation characters, 5-3
`Continue` command button, 2-14
`Continue Out` command button, 2-10, 2-14
continuing
- commands on next line, 5-3
- program execution, 2-15

control registers
- displaying, 4-12, C-6

conventions
- command line, 5-1

core files
- debugging, 8-2

creating
- action lists, 5-15, 5-16
- alias and define macros, 5-9

`cregs` macro, 4-12, C-6
`‘cr` macro, 5-11
`Current Location:`, 4-2
curses(3x), 8-23
customizable command buttons, 1-4
customizing the debugger, 1-6, 6-1

D

data pointer
- specifying, C-6

`Data Value` menu, 4-4
Data Watchpoints dialog box, 3-9
Data Watchpoint Set/Change dialog box, 3-12
`dbx` debugger
- compatibility, 6-9
- `dbx_macros` startup file, 6-9

`dde` command, 2-3, 2-4
- man page, 2-3

`.dderc` startup file, 6-4
- loading `dbx` macros, 6-9

loading `xdb` macros, 6-8
`dderc_xdb` startup file, 6-8
`debug` command, 2-6, 8-2
`-attach` option, 2-7
debugger
attaching to a running process, 2-7
basic tasks, 1-6
command input box, 1-4
command line conventions, 5-1
command reference, 1-6
customizing, 1-6, 6-1
declaring temporary variables, 4-4
executing a target program, 2-14
information generated by compiler, 2-2
invoking, 2-3
invoking remotely, 8-24
language managers, B-1
line-mode user interface, A-2
looking at call/return stack, 2-18
man page, 2-3
`MANPATH` variable, 2-3
object managers, D-1
output area, 1-4
overview, 1-1
`PATH` variable, 2-3
preparing target program for, 2-2
quick start guide, 1-6
running remotely, 8-24
scope and visibility of objects, 7-1
target managers, C-1
target program I/O area, 1-4
tutorial, 1-6
user interface managers, E-1
using monitors, 3-2
viewing program data, 4-1
debugger commands. *See* commands
debugging
assembly language code, 8-13
child and parent processes, 8-22
code compiled without `-g`, 2-2
core files, 8-2
curses(3x), 8-23
dynamically loaded code, 8-4
forked processes, 8-22
ioctl(2), 8-23
loops, 3-5
multiprocess programs, 8-22
multi-threaded applications, 8-6
optimized code, 8-17
shared libraries, 8-4
threaded applications, 8-6
debugging session
ending, 2-3
starting, 2-3
`Debug Running Process()`
`File` menu choice, 2-7
`declare` command, 4-4, 7-10
`‘declared`
block search, 7-4
`‘declared` reserved identifier, 5-11, 7-9, 7-10
declaring
temporary variables, 4-4
`define` command, 5-9
`delete intercepts` command, 3-20
delimiters
macro names, 5-10
dereferencing pointers, 4-6
`describe` command, 4-8
discarding `go` commands, 5-17
display, accessing remote, 8-24
displaying
routine arguments, 4-4
source files, 2-7, 2-12
tracebacks, 2-19
dlopen(3X), 8-4
documentation, online. *See* online help
`-do` option
`breakpoint` command, 3-4
specifying action lists, 5-15
`‘dp` macro, C-6

`dump` command, 4-8, 4-12
  and assembly language code, 8-15
dumping
  memory, 8-15
  registers, 4-12, 8-15, C-6, C-8
dynamically loaded code, 8-4

E

editing
  changing modes, 6-2
  Debugger Input command line, 5-5
`EDITOR` environment variable, 5-5
ELF object manager, D-4
eliminating qualifiers, 4-2
`Enable Images/Libraries`
  `Execution` menu choice, 2-16, 8-4
ending a debugging session, 2-3
environment
  changing, 7-3
  current, 7-2
  overriding current language, 7-4
  run, 7-2
`environment` command, 2-7, 2-12, 2-19, 7-3, 7-12
`‘env` reserved identifier, 2-19, 5-13, 7-3, 7-12
errors
  action lists, in, 5-16
  broken pipe, 8-24
  continuing after, 5-7
evaluating
  expressions, 4-4
examining
  arrays, 4-5
  buffers, 4-8
  call/return stack, 2-18
  linked lists, 4-7
  objects referenced by pointers, 4-6
  registers, 4-10
  variables and expressions, 4-2
executing
  action lists, 5-16
  target program, 2-14
`Execution` menu
  `Enable Images/Libraries`, 2-16, 8-4
  `Signals/Intercepts`, 3-18, 8-4
  `Threads`, 8-8
exiting
  debugger, 2-3
  target program, 2-9
expressions
  case sensitivity, 5-4
  examining, 4-2
  printing values of, 4-4
  syntax, 5-3
external variables, 7-10

F

faults. *See* intercepts
`fdregs` macro, 4-12, C-6
`fdreg_update` macro, C-6
`File()`
  `Visit` menu choice, 2-12
`File` menu
  `Add Source Directories`, 2-12
  `Debug Running Process()`, 2-7
  `Load Corefile`, 8-2
  `Load Executable`, 2-5
  `Quit`, 2-3, 2-9
  `Rerun`, 2-9
  `Unload Executable`, 2-9
files
  `average`, 2-2
  `dbx_macros`, 6-9
  `.dderc`, 6-4
  startup, 6-3
  `xdb_macros`, 6-8
floating-point registers
  displaying, 4-12, C-6, C-8
  monitoring for changes, C-6
`Fork Behavior`

Options menu choice, 8-22
forked processes, 8-22
specifying debugger behavior, 6-2
FORTRAN language manager, B-13
frame block qualifiers, 7-10
environment relative, 7-12
main relative, 7-12
run relative, 7-12
free command, 2-9
fregs macro, 4-12, C-6, C-8
freg_update macro, C-6
fsregs macro, 4-12, C-6
fsreg_update macro, C-6
fully qualified names, 7-7

G

-g compiler option, 2-2
general registers
displaying, 4-12, C-6
monitoring for changes, C-6
specifying, C-6
global symbol search, 7-4
go command, 2-15
action lists, 5-17
discarding, 5-17
granularity
traces, 3-14
watchpoints, 3-8, 3-13
graphical user interface, 1-2. *See also* user interface
grave accent (‘), 5-10

H

help, online. *See* online help
HP Distributed Debugging Environment (HP/DDE). *See* debugger
HP Pascal language manager, B-18
HP SOM object manager, D-3
HP-UX PA-RISC
assembly language manager, B-22
target manager, C-3
HP-UX Symbolic Debugger. *See* xdb debugger

I

identifiers, reserved, 5-11
if command, 5-16
image qualified names, 7-9
‘image reserved identifier, 5-13, 7-9
inactive
blocks, 7-10
modules, 7-6
inhibiting macro expansion, 5-10, 6-9
initialize -altdbinfo command, 2-2, 8-4
input and output operators, 5-8
input box, ():, 1-2
input command, 5-7
input, redirecting, 5-7
intercept command, 3-20
in multi-threaded applications, 8-12
intercept load command, 8-4
intercepts, 3-2, 3-18. *See also* monitors
signal names, C-5, C-8
Intercepts dialog box, 3-18
interrupt button, 1-2, 2-10
interrupting a program, 2-10
invoking
debugger, 2-3
line-mode user interface, A-2
shell commands, 5-7
target program, 2-5
ioctl(2), 8-23

K

kernel code
debugging in, 2-10
kill command, 2-9
Ksh Mode
command line editor, 5-5

L

'label reserved identifier, 5-13
language, changing, 7-4
language managers, B-1
  C, B-3
  C++, B-7
  FORTRAN, B-13
  HP Pascal, B-18
  HP-UX PA-RISC assembly language, B-22
  SPARC assembly language manager, B-25
-language option, 5-4, 7-4
lexical block search, 7-4
libdce.sl, 8-7
libraries
  dynamically loaded, 8-4
  shared, 8-4
  system, 2-10
line continuation, 5-3
line-mode user interface, A-1
  example of use, A-5
  invoking, A-2
  manager, E-4
  screen display conventions, A-4
  startup file, A-3
linked lists
  examining, 4-7
list blocks command, 7-9
list images command, 8-5
list intercepts command, 3-20
list threads command, 8-9
Load Corefile
  File menu choice, 8-2
Load Executable
  File menu choice, 2-5
loading target programs, 2-4, 2-5
Load/Rerun dialog box, 2-5
location buttons, 1-2
'long reserved identifier, 5-13
lookups, 7-4

M

macros
  'after_debug, 6-4
  'arg0 - 'arg3, C-6
  'asm, C-6, C-8
  creating, 5-9
  cregs, 4-12, C-6
  dbx compatibility, 6-9
  delimiting names, 5-10
  'dp, C-6
  expansion, 5-9
  fdregs, 4-12, C-6
  fdreg_update, C-6
  fregs, 4-12, C-6, C-8
  freg_update, C-6
  fsregs, 4-12, C-6
  fsreg_update, C-6
  inhibiting expansion, 5-10, 6-9
  names of, 5-10
  parameters, 5-10
  'pc, C-6
  'r0 - 'r31, C-6
  register name, C-6
  regs, 4-12, C-6, C-8
  reg_update, C-6
  'ret0, 'ret1, C-6
  'sp, C-6
  special, 5-11
  sregs, 4-12, C-6
  xdb compatibility, 6-8
'main reserved identifier, 5-13, 7-12
managers
  language, B-1
  object, D-1
  target, C-1
  user interface, E-1
MANPATH variable, 2-3
memory dumping, 4-8, 8-15
menu bar, 1-2

monitors. *See* breakpoints, intercepts, traces, watchpoints
  action lists and, 5-15
  using, 3-2
`Motif Mode`
  command line editor, 5-5
mounting file systems, 2-8
mouse
  execution commands, 2-17
multiple action lists, 5-16
multiprocess debugging, 8-22
multi-threaded applications. *See* threads
  debugger commands, 8-11
  debugging, 8-6

N

names
  block qualified, 7-6
  frame block qualified, 7-10
  fully qualified, 7-7
  image qualified, 7-9
  symbols, 7-4, 7-6
nondebuggable code, 2-2, 2-10

O

object managers
  HP SOM, D-3
  Solaris SPARC, D-4
online help
  commenting on, 1-6
  overview, 1-6
  using, 1-7
operators
  input and output, 5-8
optimized code, 8-17
  compared with unoptimized code, 8-18
  instruction fragments, 8-19
  program logic, 8-18
  statement-to-instruction mapping, 8-19
  values of variables, 8-21
`Options` menu, 6-2
  `Command Input Mode`, 5-5
  `Fork Behavior`, 8-22
  `User Configurable Buttons`, 2-15
`/opt/langtools/bin`, 2-3
`/opt/langtools/dde/examples`, 2-2
`/opt/langtools/share/man`, 2-3
OSF/Motif
  command line editor, 5-5
  user interface manager, E-3
output area, 1-4
output, redirecting, 5-7
overriding current language, 7-4

P

`PAGER` variable, A-2
parameter input box, `():`, 1-2
parameters
  and input files, 5-8
  and macros, 5-10
parentheses (`()`), 5-2, 5-3, 5-10
parent processes, 8-22
PA-RISC
  assembly language manager, B-22
  target manager, C-3
Pascal language managers, B-18
`PATH` variable, 2-3
PC location, 7-2
`‘pc` macro, C-6
PC (Program Counter) arrow, 2-10, 2-14, 4-2
period (`.`), 5-10, 6-9
`Permission denied` error, 2-8
personal startup file
  creating, 6-4
  loading `dbx` macros, 6-9
  loading `xdb` macros, 6-8
  sample file, 6-5
playing back command sequences, 5-6
pointers, 4-6

'predefined
block search, 7-4
'predefined reserved identifier, 5-14, 7-9
preparing target program, 2-2
preserving breakpoints and watchpoints, 2-9
print command, 4-4, 4-8
Print() command button, 4-3
Print*() command button, 4-3
Procedure()
Visit menu choice, 4-2
process, attaching to, 2-7
program counter
arrow, 2-10, 2-14, 4-2
location, 2-10
specifying, C-6
program environment
specifying defaults, 6-2
program execution
beginning or continuing, 2-15
stepping, 2-16
property abort command, 5-7
property array_dim_max command , 4-6
property flags tgt_shlib_debug command, 8-4, 8-7, C-4
property fork command, 8-22
property language command, 5-4, 7-4
property libraries command, 8-4
property qual_max command, 4-2, 7-8
property record command, 5-6
property sdir command, 2-2, 2-12
pxdb command, 2-7

Q

quick start guide, 1-6
Quit
File menu choice, 2-3, 2-9
quit command, 2-3
quotation marks (""), 5-10

R

recording command sequences, 5-6
recursive procedures
identifying symbols in, 7-10
redirecting stdin, stdout, and stderr, 2-5, 2-6, 5-7
register name macros, C-6
registers
commands, 4-12
displaying, 4-12, 8-15, C-6, C-8
examining, 4-10
monitoring for changes, C-6
specifying, C-6
Registers
Show menu choice, 4-10
regs macro, 4-12, C-6, C-8
reg_update macro, C-6
remote operation of the debugger, 8-24
accessing the display, 8-24
Rerun
File menu choice, 2-9
reserved identifiers, 5-11
'amb, B-7
'declared, 7-9, 7-10
'env, 7-3, 7-12
'image, 7-9
'main, 7-12
'predefined, 7-9
'run, 7-12
'va, 8-15
resolving syntax conflicts, 5-3
restart command, 2-9
restarting
target program, 2-9
'ret0 and 'ret1 macros, C-6
return value registers
specifying, C-6
routines
displaying arguments, 4-4
running target programs, 2-4, 2-5
'run reserved identifier, 5-14, 7-12

S

sample program, 2-2
scope, 7-2
  rules, 7-4
`Search`
  `Visit` menu choice, 2-12
searching for text strings, 2-12
search rules, 7-4
semicolon (`;`), 5-2
`Set`
  `Break` menu choice, 3-6
  `Trace` menu choice, 3-16
  `Watch` menu choice, 3-12
setting
  breakpoints, 3-3
  intercepts, 3-18
  traces, 3-14
  watchpoints, 3-8, 3-13
shared libraries
  debugging, 8-4
`shell` command, 5-7
shl_load (3X), 8-4
`‘short` reserved identifier, 5-14
`Show`
  `Break` menu choice, 3-8
  `Trace` menu choice, 3-16
  `Watch` menu choice, 3-13
`Show` menu
  `Assembly Instructions`, 2-16, 8-13
  `Registers`, 4-10
  `Stack`, 2-18, 4-2, 7-3
Show Registers dialog box, 4-10
signal
  names, C-8
signals
  intercepting, 3-18
  names, C-5
`Signals/Intercepts`
  `Execution` menu choice, 3-18, 8-4
simultaneous action lists, 5-17
SoftBench
  user interface manager, E-5
SoftBench Program Debugger, 1-1
Solaris
  SPARC assembly language manager, B-25
  SPARC target manager, C-7
Solaris SPARC object manager, D-4
SOM object file format
  object manager, D-3
`Source Actions` popup menu, 2-17, 4-3
source files
  displaying, 1-4, 2-7, 2-12
  searching for text strings, 2-12
space registers, displaying, 4-12, C-6
SPARC
  assembly language manager, B-25
  target manager, C-7
special macros, 5-11
`‘sp` macro, C-6
`sregs` macro, 4-12, C-6
stack, 2-18
  modifying display, 6-2
`Stack`
  `Show` menu choice, 2-18, 4-2, 7-3
`Stack Frame:`, 4-2
`Stack Frame` buttons, 7-3
Stack Options dialog box, 2-18
stack pointer, specifying, C-6
starting
  debugging session, 2-3
  program execution, 2-15
  target program, 2-5
startup files, 6-3
  `dbx_macros`, 6-9
  `.dderc`, 6-4
  `dderc_xdb`, 6-8
  line-mode user interface, A-3
‘stdin’’, `stdout`, and `stderr`, 2-5, 2-6
`stdin`, `stdout`, and `stderr`, 5-7
`step` command, 2-16

action lists, 5-17
Step command button, 2-14
Step Over command button, 2-14
stepping
into system or library calls, 2-16
program execution, 2-14, 2-16
stopping
debugger, 2-3
target program, 2-9
suspend intercepts command, 3-20
symbols
naming, 7-4, 7-6
scope of, 7-4
user-declared, 7-10
visibility, 7-4
syntax
command line, 5-3
expression, 5-3
resolving conflicts, 5-3
system libraries, 2-10

T

target managers
HP-UX PA-RISC, C-3
Solaris SPARC, C-7
target program
changing environment variables, 2-5
executing, 2-14
exiting, 2-9
loading during debugger startup, 2-4
loading from the debugger, 2-5
redirecting stdin, stdout, and stderr, 2-5, 2-6, 5-7
restarting, 2-9
target program I/O area, 1-4
tb command, 2-7, 2-19
temporary variables, declaring, 4-4
terminating
commands, 5-2
target programs, 2-9
tgt_shlib_debug manager option
property flags command, C-4
thread command, 8-11
threaded applications
using libdce.sl, 8-7
'thread reserved identifier, 5-14, 8-11
threads
breakpoints, 8-9
current thread, 8-11
debugger commands, 8-11
debugging multi-threaded applications, 8-6
environment, 8-10
list threads command, 8-9
'thread specifier, 8-11
Threads
Execution menu choice, 8-8
Threads dialog box, 8-8
command buttons, 8-9
tracebacks
displaying, 2-19
trace command, 3-14
Trace menu, 3-15
Set, 3-16
Show, 3-16
traces, 3-2. *See also* monitors
action lists and, 5-15
granularity, 3-14
listing, 3-16
setting, 3-14
Trace menu, 3-15
Trace Set/Change dialog box, 3-16
tutorial, 1-6

U

unbalanced brackets, braces, or parentheses, 5-3
undebuggable code, 2-2, 2-10
Unload Executable
File menu choice, 2-9
User Configurable Buttons
Options menu choice, 2-15

user interface
  graphical, 1-2
  line-mode, A-1
  remotely invoking, 8-24
user interface managers, E-1
  line-mode, E-4
  OSF/Motif, E-3
  SoftBench, E-5
`use source` command, 2-2

V

`Values Display`
  `Watch` menu choice, 3-9
`'va` reserved identifier, 5-14, 8-15
variables
  assigning values to, 4-4
  declaring, 4-4
  examining, 4-2
  external, 7-10
  printing values of, 4-4
visibility
  rules, 7-4
`Visit` menu, 7-3
  `File()`, 2-12
  `Procedure()`, 4-2
  `Search`, 2-12

W

`Watch()` command button, 3-10
`Watch*()` command button, 3-10
`Watch` menu, 3-11
  `Set`, 3-12
  `Show`, 3-13
  `Values Display`, 3-9
`watchpoint` command, 3-13
watchpoints, 3-2. *See also* monitors
  action lists and, 5-15
  granularity, 3-8, 3-13
  listing, 3-13
  modifying display, 6-2
  on registers, C-6
  preserving, 2-9
  setting, 3-8, 3-13
  setting with command buttons, 3-10
  viewing and modifying, 3-9
  `Watch` menu, 3-11
windows
  changing display, 6-2

X

`xdb` debugger
  compatibility, 6-8
  `xdb_macros` startup file, 6-8