

HP 9000  
Computers

**HP-UX Symbolic Debugger  
User's Manual**

# **HP-UX Symbolic Debugger User's Guide**

**HP 9000 Computers**



**HP Part No. B2355-90044  
Printed in USA 08/92**

**First Edition  
E0892**

---

## Legal Notices

The information contained 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 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 © Hewlett-Packard Company 1990, 1991, 1992

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 sub-paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company  
3000 Hanover Street  
Palo Alto, CA 94304 U.S.A.

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

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

Copyright © UNIX System Laboratories, Inc. 1980, 1984, 1986  
Copyright © The Regents of the Univ. of California 1979, 1980,1983,  
1985-1990

This software and documentation is based in part on materials licensed from The Regents of the University of California. We acknowledge the role of the Computer Systems Research Group and the Electrical Engineering and Computer Sciences Department of the University of California at Berkeley and the other named Contributors in their development.



---

## Printing History

New editions of this manual will incorporate all material updated since the previous edition. The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

August 1992 ... Edition 1.

This edition includes information on how to debug C++ parameterized types, exceptions, and nested classes. There is also information on how the debugger provides for debugging shared libraries, source file mapping, viewing of the execution stack, and necessary resources for window mode. This manual replaces HP part number B1864-90005.

---

## Preface

The *HP-UX Symbolic Debugger User's Guide* explains how to debug computer programs on HP 9000 computer systems. The manual assumes that you are an experienced programmer familiar with symbolic debuggers on other systems.

This manual contains the following chapters:

- |            |  |
|------------|--|
| Chapter 1  | Introduces the HP Symbolic Debugger - what it is and who can use it. This chapter also explains how to prepare a program for use with the symbolic debugger.                                   |
| Chapter 2  | Contains listings of sample debugger programs which are used in sample debugger sessions online. Use these listings for reference to the online programs when experimenting with the debugger. |
| Chapter 3  | Describes how to use the HP Symbolic Debugger to debug programs.   |
| Chapter 4  | Discusses the HP Symbolic Debugger commands.   |
| Chapter 5  | Covers information that is specific to the use of the symbolic debugger for debugging C++ programs.  |
| Chapter 6  | Covers information that is specific to the use of the symbolic debugger for debugging shared libraries.  |
| Appendix A | Lists warning and error messages, along with their remedial actions.   |
| Appendix B | Lists the language operators for HP C and HP C++.  |
| Appendix C | Lists the language operators for HP FORTRAN 77 and explains FORTRAN VMS record support.  |
| Appendix D | Lists the language operators for HP Pascal.  |
| Appendix E | Lists the special variables and environment variables used by the HP Symbolic Debugger.  |
| Appendix F | Lists some limitations of the HP Symbolic Debugger and gives some usage hints.   |

Appendix G	Lists installed HP Symbolic Debugger files.
Appendix H	Provides a brief description of all of the HP Symbolic Debugger commands.
Appendix I	Gives a comparison between the <code>xdb</code> and <code>cdb</code> HP Symbolic Debuggers.
Appendix J	Lists registers displayed by the HP Symbolic Debugger in disassembly mode.
Glossary	Lists new terms and their definitions.

## Additional Documentation

This manual does not discuss the HP-UX operating system in detail. Only those aspects relevant to the HP Symbolic Debugger are mentioned. Similarly, details about compiling a program using HP FORTRAN 77, HP Pascal, HP C, and HP C++ are only discussed to the extent that they affect how you use the HP Symbolic Debugger. See the appropriate operating system or language manual for complete information about those subjects. The following is a partial list of the operating system and language manuals:

Series 300/400 Computer Manual Title	Number Used to Order Manual
<i>Programming on HP-UX</i>	B2355-90026
<i>HP-UX Portability Guide</i>	B2355-90025
<i>HP FORTRAN/9000 Programmer's Reference</i>	B2408-90010
<i>HP FORTRAN/9000 Programmer's Guide</i>	B2408-90009
<i>HP Pascal Reference</i>	B2373-90000
<i>C Programmer's Guide</i>	B1864-90008
<i>C Programming Tools</i>	B1864-90009
<i>C: A Reference Manual (a Prentice Hall book)</i>	
<i>HP C++ Programming Language</i>	B2402-90001
<i>HP C++ Programmer's Guide</i>	92501-90005
<i>HP C++ Developer User's Guide</i>	B1697-90000
<i>HP-UX Assembler and Tools</i>	B1864-90014

<b>Series 600 Computer Manual Title</b>	<b>Number Used to Order Manual</b>
<i>Programming on HP-UX</i>	B2355-90026
<i>HP FORTRAN/9000 Programmer's Reference</i>	B2408-90010
<i>HP FORTRAN/9000 Programmer's Guide</i>	B2408-90009
<i>Assembly Language Reference Manual</i>	92432-90001

<b>Series 700/800 Computer Manual Title</b>	<b>Number Used to Order Manual</b>
<i>Programming on HP-UX</i>	B2355-90026
<i>HP-UX Portability Guide</i>	B2355-90025
<i>HP FORTRAN/9000 Programmer's Reference</i>	B2408-90010
<i>HP FORTRAN/9000 Programmer's Guide</i>	B2408-90009
<i>HP Pascal/HP-UX Reference Manual</i>	92431-90005
<i>HP Pascal/HP-UX Programmer's Guide</i>	92431-90006
<i>HP C/HP-UX Reference Manual</i>	92453-90024
<i>HP C Programmer's Guide</i>	92434-90002
<i>HP C++ Programming Language</i>	B2402-90001
<i>HP C++ Programmer's Guide</i>	92501-90005
<i>HP C++ Developer User's Guide</i>	B1697-90000
<i>Assembly Language Reference Manual</i>	92432-90001
<i>HP-UX Floating-Point Guide</i>	B2355-90024

---

## Conventions

### CASE

In a syntax statement, commands and keywords are shown in uppercase and lowercase characters. The characters must be entered exactly as shown. For example:

```
breakpoint
```

cannot be entered as any of the following:

```
Breakpoint  BreakPoint  break_point
```

### *italics*

In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with an actual value. In the following example, you must replace *filename* with the name of the file:

```
view filename
```

Italics font is also used to emphasize a *word* or *words*.

A name in italics followed by a number in parentheses (e.g., *more(1)*) is a reference to an entry in the *HP-UX Reference*.

### punctuation

In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the colon must be entered:

```
file:proc
```

### underlining

Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, “yes” is the user’s response to the prompt:

```
Really quit? >> y
```

---

## Conventions (continued)

{ } In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either `db` or `delete breakpoint`:

```
{ db
  delete breakpoint }
```

Note that the debugger use braces (`{ }`) to group commands. These groups are called *command lists* throughout this manual. This usage can be distinguished from the notation above because the enclosed entries are not stacked, but sequential.

[ ] In a syntax statement, brackets enclose optional elements. In the following example, *count* can be omitted:

```
s [count]
```

[ ... ] In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces.

... In an example, horizontal ellipses indicate where portions of the example have been omitted.

---

## Conventions (continued)

<input type="text"/>	The symbol <input type="text"/> indicates a key on the keyboard. For example, <input type="text"/> represents the carriage return key.
<input type="text"/> <i>char</i>	<input type="text"/> <i>char</i> indicates a control character. For example, <input type="text"/> S means you press the control key and the S key simultaneously.
>	The HP Symbolic Debugger prompt.
	Represents “or”.
;	Separates commands in a command list.





# Contents

---

<b>1. Introducing the HP Symbolic Debugger</b>	
Who Can Use the HP Symbolic Debugger . . . . .	1-4
Special Considerations for the Series 300/400 7.40 and 8.0 Releases . . . . .	1-4
Special Considerations for the Series 600/700/800 8.0 Release . . . . .	1-5
Special Considerations for the Series 300/400 9.0 Release . . . . .	1-6
Special Considerations for the Series 600/700/800 9.0 Release . . . . .	1-6
Creating a Program with Debugger Information . . . . .	1-7
Terminal Support . . . . .	1-8
Command History . . . . .	1-10
Where To Go from Here . . . . .	1-11
<b>2. Getting Started</b>	
Debugger Session Scenario One . . . . .	2-2
Running the Sample Sessions . . . . .	2-2
Debugger Session Scenario Two . . . . .	2-4
Running Sample Sessions Two . . . . .	2-4
Where To Go from Here . . . . .	2-5
Sample Program Listings . . . . .	2-6
Sample HP FORTRAN 77 Program . . . . .	2-7
Sample HP Pascal Program . . . . .	2-8
Sample HP C Program . . . . .	2-10
Sample HP C++ Program . . . . .	2-12

<b>3. Using the HP Symbolic Debugger</b>	
Preparing the Program . . . . .	3-3
Preparing Shared Libraries . . . . .	3-4
Starting the HP Symbolic Debugger . . . . .	3-5
Customizing the Symbolic Debugger Environment . . . . .	3-9
Toggling the Case Sensitivity . . . . .	3-9
Setting Up the Screen . . . . .	3-10
Setting Up the Locale . . . . .	3-11
Once You Start the HP Symbolic Debugger ... . . . .	3-13
Starting the Program . . . . .	3-16
Ending the Program . . . . .	3-17
Ending the HP Symbolic Debugger . . . . .	3-17
Displaying Lines in the Program . . . . .	3-18
Controlling the Command Window Display . . . . .	3-20
Changing the Source Window Size . . . . .	3-21
Displaying Assembly Code . . . . .	3-22
Displaying Source and Assembly Code . . . . .	3-24
Stepping through the Program . . . . .	3-26
Searching for a String in the Current File . . . . .	3-27
Pausing during Execution . . . . .	3-28
Setting Breakpoints . . . . .	3-28
Resuming Execution After a Breakpoint . . . . .	3-30
Listing Breakpoints . . . . .	3-30
Deleting Breakpoints . . . . .	3-31
Displaying Data . . . . .	3-32
Modifying Data . . . . .	3-35
Tracing Function and Procedure Calls . . . . .	3-36
Navigating the Execution Stack . . . . .	3-37
Using the down Command . . . . .	3-38
Using the up Command . . . . .	3-39
Using the top Command . . . . .	3-39
Using the View Command . . . . .	3-40
Capturing and Rerunning a Debugger Session . . . . .	3-41
Saving and Restoring the Debugger State . . . . .	3-42
Displaying Character Data and Using NLS . . . . .	3-43
Wide Characters . . . . .	3-45
Separate Interfaces (Debugging Screen Applications) . . . . .	3-47
Separate Environments by way of Adoption . . . . .	3-50

Executing Commands At Each Instruction . . . . .	3-51
Using Macros . . . . .	3-52
Altering the Execution Sequence . . . . .	3-53
Getting Help . . . . .	3-54
Adopting a Running Process . . . . .	3-55
Debugging a Program that Caused a Coredump . . . . .	3-57
Generating a Coredump . . . . .	3-57
Debugging the test_prog Program . . . . .	3-59
Mapping of Source Directories . . . . .	3-62
A Scenario for Using the apm Command . . . . .	3-63
Example 1: Both Old Path and New Path are Provided . . . . .	3-65
Example 2: Stripping Part of an Old Path . . . . .	3-65
Example 3: Prefixing a Path . . . . .	3-66

#### 4. HP Symbolic Debugger Commands

Entering Commands . . . . .	4-1
Using Uppercase and Lowercase . . . . .	4-5
Abbreviating Commands . . . . .	4-6
Entering Variable Names . . . . .	4-6
Special Variables . . . . .	4-9
Entering Expressions . . . . .	4-15
Character and String Expressions . . . . .	4-16
Symbolic Constants . . . . .	4-17
Numeric Constants . . . . .	4-18
Promotion of Operands . . . . .	4-19
Assignment . . . . .	4-19
Pointers, Casts, and Composite Types . . . . .	4-20
Arrays . . . . .	4-20
Entering Procedure Calls in an Expression . . . . .	4-21
Window Mode Commands . . . . .	4-23
File Viewing Commands . . . . .	4-28
Source Directory Mapping Commands . . . . .	4-35
Data Viewing and Modification Commands . . . . .	4-36
Stack Viewing Commands . . . . .	4-56
Status Viewing Command . . . . .	4-63
Job Control Commands . . . . .	4-64
Breakpoint Commands . . . . .	4-70
Overall Breakpoint Commands . . . . .	4-77

Breakpoint Creation Commands . . . . .	4-79
Breakpoint Status Commands . . . . .	4-86
All-Procedures Breakpoint Commands . . . . .	4-88
Global Breakpoint Commands . . . . .	4-92
Auxiliary Breakpoint Commands . . . . .	4-93
Exception Handling Commands . . . . .	4-95
Assertion Control Commands . . . . .	4-97
Record and Playback Commands . . . . .	4-102
Macro Facility Commands . . . . .	4-105
Signal Control Commands . . . . .	4-108
Miscellaneous Commands . . . . .	4-112

**5. C++ and the Symbolic Debugger**

Summary of Debugger Support for C++ . . . . .	5-2
How the Debugger Deals with C++ Scopes . . . . .	5-5
What Does Scope Mean . . . . .	5-5
Class Scope . . . . .	5-5
Declaration Statement Scope . . . . .	5-7
Setting Breakpoints at the End of a Scope . . . . .	5-9
C++ Expressions . . . . .	5-10
Variables . . . . .	5-10
Global Variables . . . . .	5-11
Reference Types . . . . .	5-12
Function Calls . . . . .	5-12
Operators . . . . .	5-16
Class Objects . . . . .	5-18
Displaying Type Information for an Object . . . . .	5-19
Displaying the Contents of an Object . . . . .	5-23
Object Identification . . . . .	5-26
Class Members . . . . .	5-28
Data Members . . . . .	5-28
Member Functions . . . . .	5-30
Object Pointers . . . . .	5-33
Member Pointers . . . . .	5-35
Casts . . . . .	5-36
Anonymous Unions . . . . .	5-40
Displaying Static Data Members . . . . .	5-42
Listing Local Variables . . . . .	5-44

Listing Functions . . . . .	5-45
Listing Functions . . . . .	5-45
Listing Overloaded Functions . . . . .	5-46
Viewing Functions with the Debugger . . . . .	5-47
Example . . . . .	5-47
Breakpoint Commands . . . . .	5-50
Setting a Breakpoint on a Function . . . . .	5-50
Setting a Breakpoint on Overloaded Functions . . . . .	5-52
Setting a Breakpoint at all Member Functions of a Class . . . . .	5-53
Setting an Instance Breakpoint . . . . .	5-54
Handling Exceptions . . . . .	5-57
Using throw and catch . . . . .	5-57
Stopping on a throw Statement . . . . .	5-59
Executing a throw Command List . . . . .	5-60
Stopping on a catch Statement . . . . .	5-61
Executing a catch Command List . . . . .	5-62
Listing Exceptions . . . . .	5-63
Exception Command's Effect on Other Commands . . . . .	5-64
Step-Into (s) . . . . .	5-64
Step-Over (S) . . . . .	5-64
Debugging Parameterized Types . . . . .	5-65
Using Parameterized Types . . . . .	5-66
Setting Breakpoints in Templates . . . . .	5-67
All Member Functions of a Class Template . . . . .	5-67
All Member Functions of a Template Class . . . . .	5-68
A Single Class Template Member Function . . . . .	5-69
A Single Class Template Member Function Instance . . . . .	5-69
Function Templates . . . . .	5-70
Displaying Template Data . . . . .	5-70
Data Member Values in a Template Class . . . . .	5-70
Calling a Template Function . . . . .	5-70
The Type of An Object Declared as a Template Class . . . . .	5-71
The Template Type of an Object . . . . .	5-71
Listing Templates . . . . .	5-71
Classes . . . . .	5-72
Class Templates . . . . .	5-72
Function Templates . . . . .	5-72
Template Functions . . . . .	5-73

Using Nested Classes . . . . .	5-74
References to Static Members . . . . .	5-74
References to Class Names of Enclosed Classes . . . . .	5-75
Customizing Default Debugger Behavior . . . . .	5-76
Sample C++ Debugging Sessions . . . . .	5-77
Session One . . . . .	5-77
Session Two . . . . .	5-88
<b>6. Debugging Shared Libraries</b>	
Enabling the Debugging of Shared Libraries . . . . .	6-2
Creating the Library . . . . .	6-2
Naming a Shared Library . . . . .	6-2
Locating Shared Libraries . . . . .	6-3
Invoking the Debugger . . . . .	6-4
The Debugger Environment (Symbol Binding) . . . . .	6-6
Shared Library Symbols . . . . .	6-8
Explicit Library References . . . . .	6-9
Debugging Shared Libraries in Disassembly Mode . . . . .	6-10
Summary of Extended Debugger Commands . . . . .	6-11
Debugging Shared Libraries in an Adopted Process (xdb -P) . . . . .	6-12
Special Considerations . . . . .	6-14
<b>A. Messages</b>	
User Errors (UE42 - UE2031) . . . . .	A-3
<b>B. HP C and C++ Language Operators</b>	
HP C and C++ Language Operators . . . . .	B-1
HP C and C++ Language Operators . . . . .	B-2
<b>C. HP FORTRAN 77 Language Operators and VMS Record Support</b>	
HP FORTRAN 77 Language Operators . . . . .	C-1
HP FORTRAN 77 Language Operators . . . . .	C-2
VMS FORTRAN Records . . . . .	C-4

<b>D. HP Pascal Language Operators</b>	
HP Pascal Language Operators . . . . .	D-1
HP Pascal Language Operators . . . . .	D-2
<b>E. Special and Environment Variables Used by the Symbolic Debugger</b>	
Special Variables . . . . .	E-1
<b>F. Limitations and Hints</b>	
Limitations and Hints . . . . .	F-1
Source Limitations . . . . .	F-1
Process Limitations . . . . .	F-2
Single Step Limitations . . . . .	F-3
Signals Restrictions . . . . .	F-3
Operators Limitations . . . . .	F-3
Object Type Limitations . . . . .	F-4
Files Restrictions . . . . .	F-6
Naming Restrictions . . . . .	F-6
Command-Line Procedure Call Limitations . . . . .	F-7
Shared Library Limitations . . . . .	F-8
Disassembly Mode Limitations . . . . .	F-10
Save State Limitations . . . . .	F-11
Pointer Limitation . . . . .	F-12
Address Format Restriction . . . . .	F-12
Hints for Using Assertions . . . . .	F-12
Window Mode Requirements . . . . .	F-13
<b>G. Installed Files</b>	
Debugger Installation . . . . .	G-1
<b>H. HP Symbolic Debugger Commands</b>	
Invocation Options . . . . .	H-1
Window Mode Commands . . . . .	H-5
File Viewing Commands . . . . .	H-7
Data Viewing and Modification Commands . . . . .	H-10
Source Directory Mapping Commands . . . . .	H-18
Stack Viewing Commands . . . . .	H-19
Status Viewing Command . . . . .	H-21
Job Control Commands . . . . .	H-22



Breakpoint Commands . . . . .	H-25
Exception Handling Commands . . . . .	H-39
Assertion Control Commands . . . . .	H-40
Record and Playback Commands . . . . .	H-42
Macro Facility Commands . . . . .	H-44
Miscellaneous Commands . . . . .	H-45
Signal Control Commands . . . . .	H-51

## **I. Comparison between the xdb and cdb Symbolic Debuggers**

Startup Command File . . . . .	I-1
Basic Command Form . . . . .	I-2
Basic Command Form for xdb . . . . .	I-2
Basic Command Form for cdb . . . . .	I-2
Variable Name Conventions . . . . .	I-2
Special Variables . . . . .	I-3
Expression Conventions . . . . .	I-3
The Debugger Special Variable \$lang . . . . .	I-4
Division Operator . . . . .	I-4
Command-Line Editing Environment Variables . . . . .	I-4
Split-Screen Mode . . . . .	I-5
Single-Stepping Commands . . . . .	I-5
File Viewing Commands . . . . .	I-6
Data Viewing Commands . . . . .	I-8
Stack Viewing Commands . . . . .	I-10
Job Control Commands . . . . .	I-11
Breakpoint Counts . . . . .	I-12
Breakpoint Commands . . . . .	I-13
Assertion Evaluation . . . . .	I-14
Assertion Commands . . . . .	I-15
Signal Command . . . . .	I-15
Toggle Recording . . . . .	I-16
Toggle Case Sensitivity . . . . .	I-16
Save-State . . . . .	I-16

**J. Registers Displayed by the HP Symbolic Debugger in Disassembly Mode**

Register Names for Series 600/700/800 Computers . . . . .	J-1
Special Variables Names Used for Registers . . . . .	J-1
Special Variables Names Used for Registers (Continued) . . . . .	J-2
Registers Displayed in the General or Floating-Point Register Windows . . . . .	J-2
Registers Displayed in the Special Register Window . . . . .	J-3
Register Names for Series 300/400 Computers . . . . .	J-4
Special Variable Names Used for Registers . . . . .	J-4
Registers Displayed in the General and Floating-Point Register Window . . . . .	J-4

**Glossary**

**Index**

## Figures

---

1-1. Creating an Executable Program File . . . . .	1-7
2-1. HP FORTRAN 77 Main Source File, Fortran_demo . . . . .	2-7
2-2. HP Pascal Main Source File, Pascal_demo . . . . .	2-8
2-2. HP Pascal Main Source File, Pascal_demo (Continued) . . . . .	2-9
2-3. C Main Source File, C_demo . . . . .	2-10
2-3. C Main Source File, C_demo (Continued) . . . . .	2-11
2-4. HP C++ Main Source File, C++_demo . . . . .	2-12
2-4. C++ Main Source File, C++_demo (Continued) . . . . .	2-13
2-4. C++ Main Source File, C++_demo (Continued) . . . . .	2-14
2-4. C++ Main Source File, C++_demo (Continued) . . . . .	2-15
2-4. C++ Main Source File, C++_demo (Continued) . . . . .	2-16
2-4. C++ Main Source File, C++_demo (Continued) . . . . .	2-17
2-4. C++ Main Source File, C++_demo (Continued) . . . . .	2-18
3-1. The HP Symbolic Debugger Screen (Source Mode) . . . . .	3-13
3-2. The HP Symbolic Debugger Screen (Disassembly Mode) . . . . .	3-23
3-3. The HP Symbolic Debugger Screen (Source and Disassembly Mode) . . . . .	3-24
3-4. Debugging the Program test_prog . . . . .	3-59
3-5. Viewing the Procedure that Called set_to . . . . .	3-61
4-1. Stack Depth . . . . .	4-56
4-2. Listing a Breakpoint . . . . .	4-77
4-3. Signal Numbers for the z Command . . . . .	4-108
4-4. A View of Object and Core Address Maps . . . . .	4-119
5-1. Nested Classes . . . . .	5-74

## Tables

---

1-1. Examples of Terminals that Do and Do Not Support Window Mode . . . . .	1-8
4-1. Methods for Specifying Variables . . . . .	4-7
4-2. Escape Sequences . . . . .	4-17
4-3. Symbolic Constants . . . . .	4-18
4-4. Data Viewing Formats . . . . .	4-53
4-5. Shorthand Notation for Size . . . . .	4-54
4-6. Record and Playback Commands . . . . .	4-103
5-1. Debugger Support for C++ . . . . .	5-2
5-2. Bits Contained in the \$cplplus Variable . . . . .	5-76
B-1. Language Operators for HP C and C++ . . . . .	B-2
C-1. Language Operators for HP FORTRAN 77 . . . . .	C-2
D-1. Language Operators for HP Pascal . . . . .	D-2
E-1. Special Variables . . . . .	E-1
E-2. Environment Variables . . . . .	E-4
H-1. Window Mode Commands . . . . .	H-5
H-2. File Viewing Commands . . . . .	H-7
H-3. Data Viewing and Modification Commands . . . . .	H-10
H-4. Source Directory Mapping Commands . . . . .	H-18
H-5. Stack Viewing Commands . . . . .	H-19
H-6. Status Viewing Command . . . . .	H-21
H-7. Job Control Commands . . . . .	H-22
H-8. Overall Breakpoint Commands . . . . .	H-25
H-9. Breakpoint Creation Commands . . . . .	H-26
H-10. Breakpoint Status Commands . . . . .	H-32
H-11. All-Procedures Breakpoint Commands . . . . .	H-33
H-12. Global Breakpoint Commands . . . . .	H-37
H-13. Auxiliary Breakpoint Commands . . . . .	H-38
H-14. Exception Handling Commands . . . . .	H-39
H-15. Assertion Control Commands . . . . .	H-40

H-16. Record and Playback Commands . . . . .	H-42
H-17. Macro Facility Commands . . . . .	H-44
H-18. Miscellaneous Commands . . . . .	H-45
H-19. Signal Control Commands . . . . .	H-51
I-1. Startup Command File . . . . .	I-1
I-2. Variable Name Conventions . . . . .	I-2
I-3. Special Variables . . . . .	I-3
I-4. Division Operator . . . . .	I-4
I-5. Command-Line Editing Environment Variables . . . . .	I-4
I-6. Single-Stepping Commands . . . . .	I-5
I-7. File Viewing Commands . . . . .	I-6
I-8. Data Viewing Commands . . . . .	I-8
I-9. Stack Viewing Commands . . . . .	I-10
I-10. Job Control Commands . . . . .	I-11
I-11. Breakpoint Counts . . . . .	I-12
I-12. Breakpoint Commands . . . . .	I-13
I-13. Assertion Commands . . . . .	I-15
I-14. Signal Command . . . . .	I-15
I-15. Toggle Recording . . . . .	I-16
I-16. Toggle Case Sensitivity . . . . .	I-16
I-17. Save-State . . . . .	I-16

## Introducing the HP Symbolic Debugger

---

This manual describes the operation of the HP Symbolic Debugger called `xdb`. Related debuggers are `cdb`, `fdb`, and `pdb`. They are only available on Series 300/400 computers and have syntax and functionality similar to `xdb`.

If you are a first-time user of HP Symbolic Debuggers, it is recommended that you use `xdb`. If you are familiar with either `cdb`, `fdb`, or `pdb` and want to compare them to `xdb`, read the appendix “Comparison between the `xdb` and `cdb` Symbolic Debuggers.”

The HP Symbolic Debugger is an interactive tool that assists you in finding errors in programs written in high-level programming languages.

On most terminals, the HP Symbolic Debugger uses the full screen. The screen is divided into an area for viewing source code, and an area for entering commands and command and program output. When you work with the debugger, you use the same language constructs that are used in the program you're debugging.

The HP Symbolic Debugger lets you:

View source code	You can view any program source line readily.
Display and modify variables	You can view the value of any type of data item in the program and you can display it in the format that is most appropriate. When necessary, you can change the value of a data item.
Trace program flow	You can execute one or more statements at one time, allowing you to closely examine program flow and data areas. If the program is large, you might prefer to set breakpoints at certain statements in the program. When the breakpoints occur, you can examine data areas and alter them if necessary. If your program contains several procedure calls, you might want to display the program stack to trace those calls. You can also trace execution at the machine-instruction level.
Capture and rerun a debugger session	If you think you might need to retrace your steps during a debugger session, you can have the debugger automatically record your session commands in a file. Then, at a later time, you can replay those commands. This playback feature can save you time because it contains the “trail” of commands that led to a given program state.
Execute debugger commands before each machine instruction	You can have the debugger execute one or more commands before it executes each instruction in the program. These commands, called assertions, can save you time when you need to examine execution progress one operation at a time.
View machine instructions	You can view disassembled-machine code with symbolic addresses at any address in your program. Register display and access are also provided. Associated source line numbers are also shown where possible, and source can also be displayed simultaneously.

## Examine core files

If the program failed in a way which caused the kernel to write a file, named `core`, containing a dump of the program state at the time of the failure, you can use the debugger to examine that file. Usually, you will be able to look at the stack, registers, and variables from debuggable portions of the program.



---

## Who Can Use the HP Symbolic Debugger

The HP Symbolic Debugger can be used by programmers who program in HP C++, HP C, HP Pascal, and HP FORTRAN 77 on HP 9000 Series 300/400 and Series 600/700/800 computers.

This manual describes the 9.0 release of the HP Symbolic Debugger (`xdb`). The `xdb` command exists in earlier releases of HP-UX, but not all features described here are present in those releases.

### Special Considerations for the Series 300/400 7.40 and 8.0 Releases

As of the 7.40 and 8.0 releases, the Series 300/400 compilers generate a different format for debugger information that is incompatible with debugger information generated by previous releases of the compilers. Therefore, if you compile with the `-g` compile-line option, you will not be able to debug this code with pre-7.40/8.0 debuggers. Similarly, the 7.40 and later debuggers cannot be used to debug code produced by pre-7.40/8.0 Series 300/400 compilers. Note that object files or libraries previously compiled with the `-g` option using these compilers can be linked to a program being debugged on 7.40 and 8.0 releases, but the symbolic information in these modules will not be used.

If you have a debuggable C++ program which was compiled with the A.02.00 revision of C++, it cannot be debugged with debuggers with a revision number of A.07.40 or later. If you attempt to do so, the debugger will issue an error message and abort. To remedy this situation, the `pxdb++` preprocessor must be updated on your system and then your program must be relinked and preprocessed once again. To update `pxdb++`, have your system administrator execute the following command after the 7.40 or 8.0 release has been installed:

```
ln /usr/bin/pxdb /usr/bin/pxdb++
```

The Series 300/400 8.0 release of the symbolic debugger is fully compatible with the compilers in the 7.40 language release, as well as the A.02.00 revision of C++ when the proper `pxdb` is used. Since this debugger is otherwise fully compatible with the A.02.00 C++ product, it can be used in place of the other debuggers provided with it (that is, `cdb++` and `xdb++`).

## Special Considerations for the Series 600/700/800 8.0 Release

The Series 600/700/800 8.0 symbolic debugger (`xdb`) is compatible with 7.0 compilers. Programs linked (but not already debugged) on 7.0 systems can be debugged on 8.0 systems. This also applies to programs linked against 7.0 libraries. Programs that have already been debugged on 7.0 systems will not be debuggable on 8.0 systems due to changes in the debug-information preprocessor (`pxdb`). In these cases the debugger will issue a message indicating that the user should re-link his application.

In certain cases where 7.0 compilers produce incorrect symbolic debug information, the preprocessor will detect this and issue an internal error. In such cases, you should re-compile an application to insure that correct debug information is generated and available to the new debugger.

If you have a debuggable C++ program which was compiled with the A.02.00 revision of C++, it cannot be debugged with the 8.0 debuggers. If you attempt to do so, the debugger will issue an error message and abort. To remedy this situation, the `pxdb++` preprocessor must be updated on your system and then your program must be relinked and preprocessed once again. To update `pxdb++`, have your system administrator execute the following command after the 8.0 release has been installed:

```
ln /usr/bin/pxdb /usr/bin/pxdb++
```

The Series 600/700/800 8.0 symbolic debugger (`xdb`) is otherwise fully compatible with the A.02.00 revision of the C++ product and can be used in place of the `xdb++` debugger.

Shared libraries compiled and linked on 8.0 or earlier releases are not debuggable at the source level.

## **Special Considerations for the Series 300/400 9.0 Release**

Shared libraries compiled and linked on the 8.0 release are not debuggable at the source level.

Programs compiled and linked on the 8.0 release can support only assembly-level debugging of shared libraries; however, if shared libraries are loaded with *shl\_load(3X)* they cannot be debugged.

Some features, including support for source-level debugging of shared-libraries, analyzing shared-library core files, and C++ exception handling, require current versions of the files `/usr/lib/nd.o` and `/lib/crt0.o` to be linked with the program (these files are normally linked as part of a debuggable compilation). The debugger will issue a warning if the version used is lacking support for a requested feature.

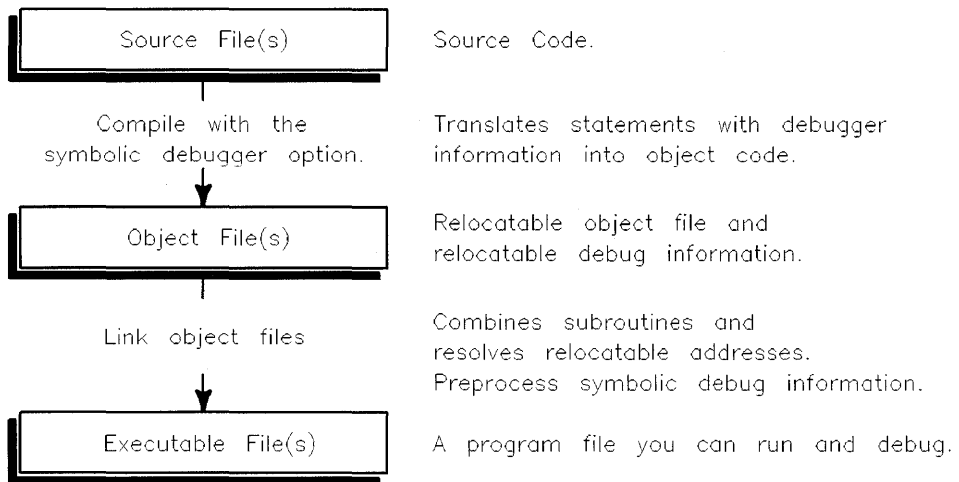
## **Special Considerations for the Series 600/700/800 9.0 Release**

Shared libraries that have been compiled and linked on any 8.0x release are not debuggable at the source level. Programs compiled and linked on the Series 700 8.05 or 8.07 releases can support only assembly-level debugging of shared libraries.

Some features, including support for source-level debugging of shared libraries, analyzing shared-library core files, and C++ exception handling, require current versions of the files `/usr/lib/nd.o` and `/lib/crt0.o` to be linked with the program (these files are normally linked as part of a debuggable compile). The debugger will issue a warning if the version used is lacking support for a requested feature.

## Creating a Program with Debugger Information

To debug a program on the symbolic level, you must compile and link the source program with debugger information to create an executable program file. The figure “Creating an Executable Program File” illustrates the process of creating an executable program file containing debugger information. If you do not compile and link your program with debugger information, the debugger can only display disassembled code, register values, absolute addresses, and linker symbols (unless it has been stripped of its linker symbol table (see *strip(1)* in the *HP-UX Reference*)). If you optimize the program using either compile-line options or optimization directives, the program can be debugged only in disassembly mode.



**Figure 1-1. Creating an Executable Program File**

## Terminal Support

Hewlett-Packard terminals with memory lock and various terminals with scrolling region capability support the command windows (window mode) used by the debugger. Other terminals operate in line mode only. Use the `-L` command line option when invoking the debugger to force operation in line mode.

The debugger uses the environment variable `TERM` to determine from the `terminfo` database if window mode is supported. For more information on environment variables and `terminfo`, read the section “Setting Up the Screen” in the chapter “Using the HP Symbolic Debugger” and the manual page `terminfo(4)` in the *HP-UX Reference*.

Some examples of `TERM` types that *do* and *do not* support window mode are given in the following table.

**Table 1-1.**  
**Examples of Terminals that Do and Do Not Support Window Mode**

Does Support Window Mode	Does Not Support Window Mode
hp2622	hp2621
hp2392	300h
hpterm <sup>1</sup>	98720
xterm <sup>1</sup>	98550
vt102	vt100

<sup>1</sup> X Window terminal emulators.

Note that the built-in console terminal (ITE) on HP workstations does not support window mode.

Programs that use escape sequences, `tty ioctls`, or screen handling packages such as the `curses(3X)` library to do special display or input handling may be hard to debug due to interactions with the debugger’s user interface. In these cases, using a separate window or terminal for the program can help.

### 1-8 Introducing the HP Symbolic Debugger

For example, to place a program's user interface in one X window and the debugger's in another X window so they will not interact, follow these steps:

1. Execute the command

```
tty Return
```

in the `hpterm` or `xterm` window chosen for the program's use.

2. Execute the command

```
sleep 10000000 Return
```

in the program's window (to keep the shell from competing with the program for input).

3. Start the debugger in another window and use the response from the `tty` command as the argument for the `-i`, `-e`, and `-o` options. For example, if the `tty` command returned `/dev/pty/ttyp4`, invoke the debugger with the following command:

```
xdb -i /dev/pty/ttyp4 -o /dev/pty/ttyp4 -e /dev/pty/ttyp4 [args]
```

For a more complete example, see the section "Separate Interfaces (Debugging Screen Applications)" in the chapter "Using the HP Symbolic Debugger."

---

## Command History

The symbolic debugger has a command history mechanism modeled after the *ksh(1)* command editing facility. The environment variables `XDBEDIT`, `VISUAL`, or `EDITOR` are checked (in that order) to determine which of the three available editing modes (`vi`, `emacs`, or `gmacs`) is to be used. For more information on the `vi`, `emacs`, and `gmacs` modes, read the *ksh(1)* man page for an explanation of these modes.

The command history file is specified by the `XDBHIST` environment variable and its size is derived from `HISTSIZ`. If any of these environment variables is not set, the default is the same as with *ksh(1)* except that `XDBHIST` defaults to `$HOME/.xdbhist`.

---

## Where To Go from Here

To get hands-on practice in running the debugger, continue on with the next chapter. It steps you through the debugging of the same sample program for C, Pascal, and FORTRAN.

If you don't have time to debug the sample program, but want to start debugging a program right away, skip to Chapter 3. Chapter 3 introduces you to the most common ways to use the debugger and should give you enough information to begin using it.

Use Chapter 4 as a reference chapter. It lists details about each of the HP Symbolic Debugger commands.

Chapter 5 covers information that is specific to the use of the symbolic debugger for debugging C++ programs.

Chapter 6 covers information that is specific to the use of the symbolic debugger for debugging shared libraries.

See appendix A for error message information.

See appendix B to find out the language operators for HP C and C++.

See appendix C to find out the language operators for HP FORTRAN 77 and VMS FORTRAN record support.

See appendix D to find out the language operators for HP Pascal.

See appendix E for a list of special variables and environment variables used by the HP Symbolic Debugger.

See appendix F for a list of HP Symbolic Debugger limitations and hints.

See appendix G for a list of installed files for the HP Symbolic Debugger.

See appendix H for a brief description of all the HP Symbolic Debugger commands.

See appendix I for a comparison between the `xdb` and `cdb` HP Symbolic Debuggers.

See appendix J for a list of registers displayed by the HP Symbolic Debugger in disassembly mode.

See the glossary for definitions of new terms.





# 2

## Getting Started

---

The HP Symbolic Debugger comes with a sample debugger session for each of the supported languages, HP FORTRAN 77, HP Pascal, HP C, and HP C++. The debugger session scenario for HP FORTRAN 77, HP Pascal, and HP C can be found in the section “Debugger Session Scenario One,” and the debugger session scenario for HP C++ can be found in the section “Debugger Session Scenario Two.” You can run these sample debugger sessions without knowing anything about the debugger; the debugger guides you through each step. The sessions take only a few minutes to run. When you’re finished, you will have a good overview of how the debugger works and some important ways it can be used.

When running the sample session, follow the instructions explained at the beginning of the session. The programs used in the debugger sessions are listed at the end of this chapter.

---

## Debugger Session Scenario One

This debugger session scenario is for HP FORTRAN 77, HP C, and HP Pascal. Suppose you're developing a program to read and process rainfall data. Proceeding in stages, you're developing the user input section and the portion that fills in an array with data from the rainfall file.

During tests, your program aborts with messages indicating that access to memory outside your program's allotment has occurred. This type of error most frequently results from bad pointer arithmetic or bad array subscripts, especially in a loop. This program does no explicit pointer arithmetic, so you've decided to use the HP Symbolic Debugger to check the loops in your program.

### Running the Sample Sessions

The directory `/usr/lib/xdb_demos` contains the source files for the HP FORTRAN 77, HP C and HP Pascal sample programs and playback files to be used in this quick overview of the capabilities of the symbolic debugger (`xdb`).

To run the sample sessions, you must first compile the source programs. To do so, `cd` to the directory where your executable files are to be built. For example, your home directory or `/tmp` are good places.

To make the executable files, type:

```
make -f /usr/lib/xdb_demos/Makefile C_demo   
make -f /usr/lib/xdb_demos/Makefile Pascal_demo   
make -f /usr/lib/xdb_demos/Makefile Fortran_demo 
```

The resulting executables will be called `democ`, `demop`, and `demof` for HP C, HP Pascal, and HP FORTRAN 77, respectively. You can start up the debugger on the executable file of your choice. The executable file chosen in the following example is `democ`. To start the debugger using this file, type:

```
xdb democ 
```

Note that the the directory `/usr/bin` must be in your `PATH`.

Once inside the debugger, you need to start the appropriate playback script. To start the HP C playback script, type:

```
<< /usr/lib/xdb_demos/c.demo 
```

at the `xdb` prompt. The playback scripts for HP FORTRAN 77 and HP Pascal are `p.demo` and `f.demo`, respectively. Note that for `xdb`, the `<` command is different from the `<<` command so be careful to enter `<<`.

---

## Debugger Session Scenario Two

This debugger session scenario is for HP C++. You're going to use a C++ program to learn how the symbolic debugger supports the debugging of C++ programs. There are no built-in errors in this program as were included in the previous scenario.

### Running Sample Sessions Two

The directory `/usr/lib/xdb_demos` contains the source file for the C++ sample program and playback file to be used in this quick overview of the capabilities of the symbolic debugger (`xdb`).

To run the sample session, you must first compile the source program. To do so, `cd` to the directory where your executable file is to be built. For example, your home directory or `/tmp` are good places.

To make the executable file, type:

```
make -f /usr/lib/xdb_demos/Makefile C++_demo 
```

The resulting executable file is given the name `demoC`. You can start up the debugger on this executable file by typing:

```
xdb demoC 
```

Once inside the debugger, you need to start the appropriate playback script. To start the C++ playback script, type:

```
<< /usr/lib/xdb_demos/C.demo 
```

at the `xdb` prompt. Note that for `xdb`, the `<` command is different from the `<<` command so be careful to enter `<<`.

---

## Where To Go from Here

Now that you've completed the sample sessions, you have a good idea about how the HP Symbolic Debugger works. To learn more details about the operations used in the debugger session or to begin debugging your own programs, continue with the chapter "Using the HP Symbolic Debugger." If you want to see the complete listings for the programs you saw in the session, read on.

---

## Sample Program Listings

This section lists the language source files used in the sample debugger sessions. The data file `RAINFALL`, which is not listed here, contains the data for the `C_demo`, `Pascal_demo`, and `Fortran_demo` programs that are listed in the following table.

These source files:	are listed in:
<code>Fortran_demo</code>	Figure 2-1
<code>Pascal_demo</code>	Figure 2-2
<code>C_demo</code>	Figure 2-3
<code>C++_demo</code>	Figure 2-4

## Sample HP FORTRAN 77 Program

```

$CONTROL RANGE, CODE_OFFSETS, TABLES
PROGRAM RAIN_REPORT
INTEGER*2 NUMBER_YEARS,
2     FIRST_YEAR,
3     YEAR_INDEX,
4     NUM_OF_MONTHS
REAL MONTH_TOTALS(60)
100 PRINT *, 'ENTER THE FIRST YEAR YOU WISH TO REPORT ON: '
READ (5,*) FIRST_YEAR
IF ((FIRST_YEAR .LT. 1950).OR.(FIRST_YEAR .GT. 1988)) THEN
GOTO 100
ENDIF
110 PRINT *, 'ENTER THE # OF YEARS YOU WISH TO CONSIDER (1-5): '
READ (5,*) NUMBER_YEARS
IF ((NUMBER_YEARS .LT. 1).OR.(NUMBER_YEARS .GT. 5)) THEN
GOTO 110
ENDIF
YEAR_INDEX = (FIRST_YEAR - 1950) * 12
NUM_OF_MONTHS = NUMBER_YEARS * 122
CALL LOADMT (YEAR_INDEX, NUM_OF_MONTHS, MONTH_TOTALS)
PRINT *, 'PROGRAM ENDS'
STOP
END

SUBROUTINE LOADMT (YEAR_INDEX, NUM_OF_MONTHS, MONTH_TOTALS)
INTEGER*2 YEAR_INDEX,
2     NUM_OF_MONTHS,
3     TABLE_INDEX
REAL MONTH_TOTALS(60),
2     HOLD_RAINFALL
OPEN (UNIT=10, FILE='RAINFALL')
DO I=1, YEAR_INDEX
READ (10,*) HOLD_RAINFALL
END DO
DO TABLE_INDEX = 1, NUM_OF_MONTHS
READ (UNIT=10, FMT=10, END=900) HOLD_RAINFALL
MONTH_TOTALS(TABLE_INDEX) = HOLD_RAINFALL
END DO
900 RETURN
END

```

Figure 2-1. HP FORTRAN 77 Main Source File, Fortran\_demo



## Sample HP Pascal Program

```

$RANGE ON, CODE_OFFSETS ON, TABLES ON$
program RainReport (INPUT, OUTPUT, RainFall);

type
  YearType      = 1900..2000;
  NumYearsType  = 0..200;
  MonthTotalType = REAL;
  ArrayType     = ARRAY [1..60] of MonthTotalType;

var
  NumYears      : NumYearsType;
  FirstYear     : YearType;
  YearIndex     : INTEGER;
  NumOfMonths   : INTEGER;
  MonthTable    : ArrayType;
  RainFall     : TEXT;

procedure GetInput;
{
  This procedure prompts the user for the initial year and number of
  years for the report.  It also checks to see that the year and number
  of years are within range.
}
const
  YearPrompt     = 'Enter the first year on which to report:  ';
  NumYearsPrompt = 'Enter the # of years to consider (1 - 5):  ';

procedure GetFirstYear;
begin {GetFirstYear statements};
  writeln (OUTPUT);
  prompt (OUTPUT, YearPrompt);
  readln (INPUT, FirstYear);
  IF (FirstYear < 1950) or (FirstYear > 1988) THEN
    GetFirstYear;
end {GetFirstYear statements};

```

Figure 2-2. HP Pascal Main Source File, Pascal\_demo

```

procedure GetNumYears;
begin {procedure GetNumYears statements};
  writeln (OUTPUT);
  prompt (OUTPUT, NumYearsPrompt);
  readln (INPUT, NumYears);
  IF (NumYears < 1) or (NumYears > 5) THEN
    GetNumYears;
end;

begin {level 1 procedure};
  GetFirstYear;
  GetNumYears;
  YearIndex := (FirstYear - 1950) * 12;
  NumOfMonths := NumYears * 12;
end {level 1 procedure};

procedure LoadMonthTable;
var
  ArrayIndex : INTEGER;
  HoldRainFall : INTEGER;

begin {LoadMonthTable statements};
  HoldRainFall := 0;
  reset (RainFall, 'RAINFALL');
  FOR ArrayIndex := 1 to YearIndex DO
  {
    This loop will perform dummy reads to get the file to the start
    of the requested data.
  }
  readln (RainFall, HoldRainFall);
  FOR ArrayIndex := 1 to NumOfMonths DO
  begin {FOR loop}
    readln (RainFall, HoldRainFall);
    MonthTable[ArrayIndex] := HoldRainFall / 100
  end {FOR loop}
end {LoadMonthTable statements};

begin {main program}
  GetInput;
  LoadMonthTable
end {of program}.

```

Figure 2-2. HP Pascal Main Source File, Pascal\_demo (Continued)

## Sample HP C Program

```

#include <stdio.h>

#define YEAR_PROMPT      "\nEnter the first year on which to report: "
#define NUM_YEARS_PROMPT "\nEnter the # of years to consider (1 - 5): "

typedef int              year_type;
typedef int              num_years_type;
typedef double          month_total_type;
typedef month_total_type array_type[60];

num_years_type  num_years;
year_type       first_year;
int             year_index;
int             num_of_months;
array_type      month_table;
FILE            *rain_fall,
               *fopen();

void get_first_year()
{
    printf (YEAR_PROMPT);
    scanf ("%d", &first_year);
    if ((first_year < 1950) || (first_year > 1988))
        get_first_year();
}

void get_num_years()
{
    printf (NUM_YEARS_PROMPT);
    scanf ("%d", &num_years);
    if ((num_years < 1) || (num_years > 5))
        get_num_years();
}

```

Figure 2-3. C Main Source File, C\_demo

```
void get_input()
{
    /*
     * This function prompts the user for the initial year and number of
     * years for the report. It also checks to see that the year and number
     * of years are within range.
     */
    get_first_year();
    get_num_years();
    year_index    = (first_year - 1950) * 12;
    num_of_months = num_years * 122;
}

void load_month_table()
{
    int array_index;
    int hold_rain_fall = 0;

    rain_fall = fopen("RAINFALL", "r");
    /* This loop will perform dummy reads to get the file to the start
     * of the requested data.
     */
    for (array_index = 1; array_index <= year_index; array_index++)
        fscanf (rain_fall, "%d", &hold_rain_fall);
    for (array_index = 1; array_index <= num_of_months; array_index++) {
        fscanf (rain_fall, "%d", &hold_rain_fall);
        month_table[array_index] = hold_rain_fall / 100;
    }
}

main()
{
    get_input();
    load_month_table();
}
```

Figure 2-3. C Main Source File, C\_demo (Continued)

## Sample HP C++ Program

```
extern "C"
{
#include <stdio.h>
void *malloc(int);
}

class buffer
{
int size;
int *pointer;

public:
buffer(int);

int buffer_size() { return size; }
int *buffer_pointer() { return pointer; }

int virtual empty() = 0;
int virtual full() = 0;
void virtual dump() = 0; /* all of it */
void virtual identify() = 0;
};

class stack : public buffer
{
int stack_pointer;
public:
stack(int);
int full();
int empty();
void operator+(int);
int operator--();
int operator[] (int);
void dump();
void dump(int); /* last n elements */
void identify();
};
```

Figure 2-4. HP C++ Main Source File, C++\_demo

```
class circular_buffer : public buffer
{
    int head;
    int tail;
public:
    circular_buffer(int);
    int full();
    int empty();
    void operator+(int);
    int operator--();
    void dump();
    void dump(int); /* last n elements */
    void identify();
};

// *****

buffer::buffer(int size)
{
    pointer = (int *) malloc(size);
    buffer::size = (pointer ? size : 0);
}

// *****

circular_buffer::circular_buffer(int size) : buffer(size)
{
    head = 0;
    tail = 0;
}

int circular_buffer::full()
{
    return (head + 1) % buffer_size() == tail;
}

int circular_buffer::empty()
{
    return head == tail;
}
```

**Figure 2-4. C++ Main Source File, C++\_demo (Continued)**

```
void circular_buffer::operator+(int i)
{
    if(full())
    {
        printf("Warning: circular buffer overflow\n");
        return;
    }

    ++head %= buffer_size();
    *(buffer_pointer() + head) = i;
}

int circular_buffer::operator--()
{
    if(empty())
    {
        printf("Warning: circular buffer underflow\n");
        return -1;
    }

    ++tail %= buffer_size();
    return *(buffer_pointer() + tail);
}

void circular_buffer::dump()
{
    int i = (tail + 1) % buffer_size();
    int limit = (head + 1) % buffer_size();
    while(i < limit)
    {
        printf("%d\n", *(buffer_pointer() + i));
        ++i %= buffer_size();
    }
}
```

**Figure 2-4. C++ Main Source File, C++\_demo (Continued)**

```

void circular_buffer::dump(int n)
{
    printf("-----\n");
    if(n <= 0)
        return;
    if(head == tail)
    {
        printf("Buffer is empty\n");
        return;
    }
    if(head > tail)
    {
        if(n > head - tail)
        {
            printf("Buffer not that deep\n");
            return;
        }
    }
    else if( n > buffer_size() - tail + head)
    {
        printf("Buffer not that deep\n");
        return;
    }
    int limit = (head + 1) % buffer_size();
    int i = (head + buffer_size() - n + 1) % buffer_size();
    do
    {
        printf("%d\n", *(buffer_pointer() + i));
        ++i %= buffer_size();
    }
    while(i != limit);
}

void circular_buffer::identify()
{
    printf("Hi, I'm a circular buffer\n");
}

// *****

stack::stack(int size) : buffer(size)
{
}

```

**Figure 2-4. C++ Main Source File, C++\_demo (Continued)**



```
stack::empty()
{
    return stack_pointer == 0;
}

stack::full()
{
    return stack_pointer == buffer_size();
}

void stack::operator+(int i)
{
    if(full())
    {
        printf("Warning: stack overflow\n");
        return;
    }
    *(buffer_pointer() + stack_pointer++) = i;
}

int stack::operator--()
{
    if(empty())
    {
        printf("Warning: stack underflow\n");
        return -1;
    }
    return *(buffer_pointer() + stack_pointer--);
}

int stack::operator[] (int offset)
// return nth element form the top
{
    if(stack_pointer <= offset)
    {
        printf("Warning: stack underflow\n");
        return -1;
    }
    return *(buffer_pointer() + stack_pointer - offset - 1);
}
```

**Figure 2-4. C++ Main Source File, C++\_demo (Continued)**

```
void stack::dump() // dump the entire stack
{
    printf("\n-----\n");
    if(empty())
    {
        printf("Empty stack\n");
        return;
    }
    for(int i = stack_pointer; i;)
        printf("[%d] %d\n", i, *(buffer_pointer() + --i));
}

void stack::dump(int n) // dump the top n elements
{
    printf("-----\n");
    if(n <= 0)
        return;
    if(stack_pointer < n)
    {
        printf("stack not that deep\n");
        return;
    }
    int limit = stack_pointer - n;
    for(int i = stack_pointer; i > limit;)
        printf("[%d] %d\n", --i, *(buffer_pointer() + i));
}

void stack::identify()
{
    printf("Hi, I'm a stack\n");
}

// *****
```

Figure 2-4. C++ Main Source File, C++\_demo (Continued)

```
main()
{
    int i;

    circular_buffer C(5);
    stack          S(5), S2(8);

    int stack::*pm; /* pointer to int member of stack */

    buffer *buffer_pointer;

    S + 1;
    S2 + 2;
    S + 3;
    S2 + 8;

    S + 4;
    i = S--;

    C + 1;
    i = C--;
    C + 2;
    i = C--;

    /* force debug info by using vars */
    pm = 0; buffer_pointer = 0;

}
```

**Figure 2-4. C++ Main Source File, C++\_demo (Continued)**

## Using the HP Symbolic Debugger

---

This chapter shows you how to start the HP Symbolic debugger and how to use its major features. The first sections of the chapter list the steps you must perform to begin using the debugger and familiarize you with the screen display. The last sections of the chapter show you how to perform various tasks. You do not perform these tasks necessarily in the same order as they are listed; pick and choose the tasks depending on your requirements.

To get started with the HP Symbolic Debugger, read and perform these sections in order:

- Preparing the Program
- Starting the HP Symbolic Debugger
- Starting the Program

---

**Note**

All examples of debugger commands in this chapter appear as:

> *command ...*

The > is the debugger's command prompt. Anything appearing after the prompt represents user input.

---

Once you start the program, read and perform the sections below that correspond to the tasks you may want to perform:

**3**

- Ending the Program
- Ending the HP Symbolic Debugger
- Displaying Lines in the Source Program
- Controlling the Command Window Display
- Changing the Source Window Size
- Displaying Assembly Code
- Displaying Source and Assembly Code
- Stepping through the Program
- Searching for a String in the Current File
- Pausing during Execution
- Displaying Data
- Modifying Data
- Tracing Function and Procedure Calls
- Capturing and Rerunning a Debugger Session
- Saving and Restoring the Debugger State
- Displaying Character Data and Using NLS
- Separate Interfaces (Debugging Screen Applications)
- Executing Commands at Each Source Line
- Using Macros
- Altering the Execution Sequence
- Getting Help
- Adopting a Running Process
- Debugging a Program that Caused a Core Dump
- Mapping of Source Directories
- Navigating the Execution Stack

---

## Preparing the Program

Before starting HP Symbolic Debugger, compile your HP C++, HP FORTRAN 77, HP Pascal, or HP C program using the symbolic debug option. If you do not use the symbolic debug option, you can only debug the program in disassembly mode; the debugger can track only register values, absolute addresses and labels.

When you're confident that the program will compile without errors, use the symbolic debug compile option. When you use the symbolic debug option, the compiler generates tables containing the names and addresses of variables, labels and source lines. These tables are the symbolic hooks into your program.

To tell the compiler to add symbolic debugger information to the executable file, use the `-g` command line option. For example, compile the C source file `test1.c` and create the executable file `test1`. This can be done by executing the following command:

```
cc -g -o test1 test1.c
```

Note that the file `/usr/lib/end.o` will automatically be linked with your program. Do not forget to include it in your list of object files to link if you invoke the linker `ld(1)` directly. For example:

```
ld -o myprog t1.o t2.o /usr/lib/end.o -lc
```

When the `-g` option is used, optimization on the generated code is automatically disabled.

On some systems, the linker by default links in shared libraries instead of archive libraries (see *glossary(9)*). To override this default, use the `-a archive` option with `ld(1)` command or the options `-Wl,-a,archive` with the compiler command.

If you optimize the program using either compile-line options or optimization directives, the program can be debugged only in disassembly mode.

## Preparing Shared Libraries

If your program is linked against shared libraries which you wish to debug, you can debug the shared libraries at the source level if they have been prepared as explained in this section. Note that system libraries (for examples, `libc`) are only debuggable in disassembly mode.

Shared libraries are created by first compiling source code with the `+z` compiler option. This creates relocatable (position independent code) object files. If you want to be able to debug the relocatable object file at the source level, you also need to compile the program with the `-g` symbolic debug option. For example, if you have a program `mytest.c` and you want to compile it for use in a shared library and make it debuggable, you would execute this command:

```
cc -g +z mytest.c
```

The linker `-b` option is then used to create the shared library. For example to create a shared library `libmyshare.sl` containing the relocatable object file `mytest.o`, you would execute this command:

```
ld -b -o libmyshare.sl mytest.o
```

Any module included in a shared library is debuggable at the source level if it has been compiled with the `-g` compiler option.

Note that forcing immediate binding with the linker's `-B immediate` option is not required for debugging your program. The default of deferred binding is acceptable.

If you will be adopting a program with the debugger's `-P` command line option and wish to debug any shared libraries used by the program, you will need to use the `pxdb` command on the program before executing it. For more information on this, see the section "Debugging Shared Libraries in an Adopted Process" in Chapter 6.

---

## Starting the HP Symbolic Debugger

When using the HP Symbolic Debugger, the debugger is the parent process and the program that you're debugging becomes a child process. The debugger controls only the child process and can debug only one child process at a time.

3

To start the debugger, enter the following command:

```
xdb [ -d altdir
      -r file
      -R file
      -p file
      -P process-ID
      -L
      -i file
      -o file
      -e file
      -S num
      -s
      -l library
      -l ALL ] [ objectfile [ corefile ] ]
```

The options for the `xdb` command are described as follows:

- |                   |  |
|-------------------|--|
| <i>objectfile</i> | Is an executable program file with zero or more of its components compiled with the <code>-g</code> option. The default for <i>objectfile</i> is <code>a.out</code> .    |
| <i>corefile</i>   | Is a core image from a failed execution of <i>objectfile</i> (see <i>core(4)</i> in the <i>HP-UX Reference</i> ). The default for <i>corefile</i> is <code>core</code> . |



- d** *altdir* Specifies an alternate directory for source files. Alternate directories are searched in the order given. If a source file is not found in any alternate directory, the current directory is searched last. When searching for the source *file* in an alternate directory *altdir*, where *file* is composed of a directory and a base file name (i.e., *dirname/basename*), *xdb* first attempts to open *altdir/dirname/basename*. If this fails, *xdb* attempts to open *altdir/basename* (see *basename (1)* in the *HP-UX Reference*).
- r** *file* Specifies a record *file*, which is invoked immediately for overwrite, rather than for append (see the section “Record and Playback Commands” in the chapter “HP Symbolic Debugger Commands”).
- R** *file* Specifies a restore state *file*, which is processed before the **-p** option (if any) and after the **-r** option (if any). The *file* must have been created previously with the **ss** command while debugging the same *objectfile* (see the section “Save State Command” in the chapter “HP Symbolic Debugger Commands”). The debugger attempts to verify this when the **-R** option is used.
- p** *file* Specifies a playback *file*, which is invoked immediately (see the section “Record and Playback Commands” in the chapter “HP Symbolic Debugger Commands”).
- P** *process-ID* Specifies the *process-ID* of an existing process that the user wishes to debug (see the section “Adopting a Running Process”).
- L** Forces the line-oriented interface, even if *xdb* can support the window-oriented interface on the terminal type specified by the environment variable **TERM**.
- i** *file* Redirects standard input to the child process from the designated file or character device.
- o** *file* Redirects standard output from the child process to the designated file or character device.
- e** *file* Redirects standard error from the child process to the designated file or character device.

- S *num*** Sets the size of the string cache to *num* bytes (default is 1024, which is also the minimum). The string cache holds textual data read from the *objectfile*.
- s** Causes all shared libraries used by an application to be loaded as private (unshared) copies. This option or the **-l** option (which implies **-s**) is required if breakpoints will be set or single stepping will be done in shared libraries.
- l *shared-library*** Pre-loads the symbolic debug information (and linker symbol table) into the debugger so that the user can view code, set breakpoints, and do other debugging operations prior to running the program. If the **-l** option is not used for a given library, no symbolic information concerning the library will be available, and you will not be able to debug that library at the source level, unless
- You explicitly make a reference to a symbol in that library (e.g. *symbol@shared-library* as opposed to just *symbol*), or
  - The debugger stops execution at some location within that library.
- shared-library* may be implicitly loaded by the program (linked in with the *ld(1)* **-l** option), or explicitly loaded by *shl\_load(3X)*.
- If *shared-library* is not a complete path name, it will be searched for using the same search rules used by the dynamic loader (see the section “Locating Shared Libraries” in Chapter 6, the *ld(1)* **+b** and **+s** options, and the section “Library Location and the Dynamic Loader” in the manual *Programming on HP-UX*). If the library is not located, any directories previously specified with the **-d** option will also be searched, followed by the current directory. If it is still not located, the symbolic debug information will still be available once the library has been mapped in (loaded), and an explicit reference to a symbol within it has been made.
- The trailing **.sl** is optional in *shared-library*.
- l ALL** Pre-loads the debug information (and linker symbol table) into the debugger for all shared-libraries used by the program, with the exception of libraries loaded with *shl\_load(3X)*, which the user must list using a separate **-l** option for each.

There can only be one *objectfile* and one *corefile* per debugging session (activation of the debugger). The program (*objectfile*) is not invoked as a child process until you give an appropriate command (see the section “Job Control Commands” in the chapter “HP Symbolic Debugger Commands”). The same program may be restarted, as different child processes, many times during one debugging session.

---

**Note**           Equivalent debugger commands exist for the `-d`, `-p`, and `-r` options. See the `D` (directory) command and the section “Record and Playback Commands” in chapter “HP Symbolic Debugger Commands.”

---

In addition to the command line options, the debugger uses various environment variables. For information on these environment variables and their default values, see the appendix “Special and Environment Variables Used by the Symbolic Debugger.”

At start-up, the debugger executes commands from the file `.xdbrc` (see the section “Customizing the Symbolic Debugger Environment”), if it exists in the `$HOME` directory. The start-up sequence is:

1. Begin recording to `-r` file
2. Read commands from `.xdbrc`
3. Process `-R` file
4. Playback from `-p` file

If you use the `-r` option when invoking the debugger, that record file will record commands from the `.xdbrc` file (if present), the `-R` file (if given), and the `-p` file (if specified).

## Customizing the Symbolic Debugger Environment

The symbolic debugger environment can be customized by setting up different environment variables and symbolic debugger commands in the `.xdbrc` file. This file is located in your `$HOME` directory. The environment variables and their default values can be found in the appendix “Special and Environment Variables Used by the Symbolic Debugger.” The symbolic debugger commands can be found in the chapter “HP Symbolic Debugger Commands.”

The subsequent sections provide examples for customizing the debugger environment to toggle case sensitivity, and set up the screen for the number of lines and columns you need.

### Toggling the Case Sensitivity

You can place special commands in your `.xdbrc` file that you want to be executed when `xdb` is invoked. For example, the symbolic debugger by default starts up case insensitive. If you are debugging C or C++ programs that use a naming convention where case is significant (for example, X Windows programs), you may wish to change this default condition to be case sensitive. This will allow you to search in the debugger for variables and strings that require case sensitivity. For example, if you were using the `p` (`print`) command to display the value of the variable `ValueOne`, the debugger would search for `ValueOne`, not `valueone`, and print that value for you.

To change the default value from case insensitive to case sensitive, when you initially start the debugger, add the following command to your `.xdbrc` file:

```
tc # toggle case command
```

## Setting Up the Screen

Environment variables define how the debugger (`xdb`) will interact with you. For example, the debugger (`xdb`) looks at the environment variables `TERM`, `LINES`, and `COLUMNS` when setting up the screen.

3

If the `TERM` environment variable is set, the debugger will attempt to retrieve information about the specified terminal type from the `terminfo` database. Screen mode is enabled if `terminfo` contains the resources the debugger needs to implement it (see the section “Window Mode Requirements” in Appendix F). If `TERM` is not defined, the debugger will use line mode.

After the debugger has determined that window mode is feasible, it sizes the screen using the first of the following methods (in the order shown) that provides dimensions:

1. If the `LINES` and `COLUMNS` environment variables are set, it uses them.
2. If automatic X window resizing (`sigwinch`) is supported, it uses the values returned by the `sigwinch` support routine.
3. If `terminfo` contains lines (`lines`) and columns (`cols`) values, it uses them.
4. If none of the above conditions exist, it uses the default values 24 and 80.

The debugger will not allow the number of lines to be less than 12, nor the number of columns to be less than 60. It will remain in screen mode, with these values, to allow the window to be resized to acceptable values.

If automatic X window resizing is supported, the debugger will use the `sigwinch` support routine to determine the new screen size, when the X window is resized. However, this is subject to the 12 lines by 60 columns or greater format restriction. If a command is in progress when the X window is resized, the debugger will reformat the screen at the next command prompt. If the command produces output in the command window, you will not be able to read it before it is erased because the screen is reformatted. If the debugger is at a command prompt, the screen should reformat immediately.

Note that the debugger only looks for the `sigwinch` signal while waiting for input. In the unlikely event that the screen is not reformatted immediately, pressing `(Return)` for a new command prompt should cause it to do so.

## Setting Up the Locale

The environment variable `LANG` defines what *locale* (for example, `german`, `english` or `chinese-t`) the debugger will use for displaying messages such as: errors, warnings, and other notices. If a localized version of the debugger's message catalog has been provided for you, it will be found on your system as

```
/usr/lib/nls/locale/xdb.cat
```

and can be enabled by setting the following in your environment:

```
LANG=locale; export LANG
```

The default catalog provided in the debugger's fileset is

```
/usr/lib/nls/C/xdb.cat
```

It will be used if `LANG` is:

1. not set
2. set to `C`
3. set to a *locale* for which a message catalog has not been provided.

In cases where a catalog is not available for a desired *locale*, the following message will be displayed when the debugger is invoked:

```
xdb: Warning! The following language(s) are not available:
      LANG=locale
Continuing processing using the language "C".
```

Debugging localized applications (that is, those that use *setlocale(3)*) is possible under the following conditions:

- The program uses the environment variable `LC_ALL` (or other `LC_xxx` settings) to define its *locale*, or
- If the program uses `LANG` to define its *locale*, you are willing to accept debugger messages in the same *locale* (or in the standard `C` locale if a debugger catalog is not available in the desired *locale*).

These conditions are necessary as the program being debugged inherits its entire environment from the debugger, and it requires a special effort for the same environment variable to have different values in the debugger and in

the program being debugged. For more information on this see the section “Separate Environments by way of Adoption.”

3

The debugger uses the environment variable `LC_CTYPE` to define the *locale* for interpretation of character data within the program (see the section “Displaying Character Data and Using NLS” later in this chapter). Note that it is possible to use a *locale* for debugger messages that is different than the *locale* used for the displaying of the program’s character data.

For more detailed information on *locales* and the environment variables that define or influence them, see the manual pages for *hpnl5(5)* and *environ(5)* in the *HP-UX Reference*.

## Once You Start the HP Symbolic Debugger ...

When you start HP Symbolic Debugger from a terminal that supports windowing, you see a screen similar to the one shown in Figure 3-1.

```

hpterm
62:     fscanf (rain_fall, "%d", &hold_rain_fall);
63:     month_table[array_index] = hold_rain_fall / 100;
64:     }
65: }
66:
67: main()
68: {
> 69:     get_input();
70:     load_month_table();
71: }

File: demo.c  Procedure: main  Line: 69
Copyright Hewlett-Packard Co. 1985.  All Rights Reserved.
<<<< XDB Version A.07.05 HP-UX >>>>
No core file
Procedures:      6
Files: 2
>

```

**Figure 3-1. The HP Symbolic Debugger Screen (Source Mode)**

**Note** The previous screen appears only on terminals that support window mode. If your terminal does not support window mode, the debugger displays information one line at a time (line mode).

The screen has three parts, which are described below. This is the screen you see when debugging in symbolic (*source*) mode.



- Source window      The source window is located at the top of the screen, above the highlighted line. This is the area where you view the source statements. If your terminal has 24 lines, the top 15 are used for the source window. To alter the number of lines in the source window, see the section “Changing the Source Window Size” in this chapter.
- Source statements are displayed one window at a time. See the section “Displaying Lines in the Source Program” for directions on locating and displaying lines in the source window.
- The > marker in the left margin of the source window points to the current line. When you first start the debugger, this is the first executable statement. The marker always points to the current viewing location. Unless a viewing command has been given, this will correspond to where the program is currently suspended.
- Location window      The location window (or location line) is the highlighted line near the middle of the screen. This line shows you the current program file, procedure name, and the source line number of the current line (the marked location currently being viewed in the source window).
- Command window      The command window is the area located below the location window (highlighted line). This window is where the debugger commands that you enter are echoed. The debugger shows its own output in this area. The command window normally also shows output from the child process (program being debugged). The window automatically scrolls up when full, but this does not affect the other windows. A scrolling *more* feature lets you view debugger output one window-full at a time.
- The debugger prompts you to enter a command by displaying >. When you enter a command, enter the entire command on one line (continuation lines are not allowed).
- For information about controlling the display of lines in the command window, see the section “Controlling the Command Window Display.”

At this point, before starting program execution, you might want to set breakpoints in the program, or change the source window size. The remaining sections in this chapter describe how you can accomplish these tasks and others as well (the tasks can also be performed during any execution pause). The sections are not listed in any particular order. You need to determine which are relevant to the debugging session at hand and perform only those.

**3**

---

## Starting the Program

3

Once you start the debugger and you are ready to begin debugging your program, enter either an **r** (**run**), **R** (**Run**), **s** (**step**) or **S** (**Step**) command. The **r** (**run**) command starts execution of the program and allows you to enter arguments with it. Subsequent uses of **r** without arguments repeat the command with the arguments previously given. The **R** (**Run**) command, as shown below, starts executing the program, but does not allow you to enter run-time arguments:

```
>R
```

To execute one statement at a time, enter either the **s** (**step**) or **S** (**Step**) command. The initial **step** command executes up to the first statement of the program. The following **s** (**step**) command allows single-stepping through the program and any procedures that it contains:

```
>s
```

The following **S** (**Step**) command allows single-stepping through the program, stepping over procedure calls—A procedure call is treated as a single statement.

```
>S
```

---

**Note** HP Symbolic Debugger commands are case sensitive regardless of the case sensitivity set with the **tc** command; you must type them exactly as documented. To see command syntax, refer to each command's listing in the chapter "HP Symbolic Debugger Commands" and "Appendix H."

---

---

## Ending the Program

If you want to terminate your program before it normally completes, enter the **k** (**kill**) command:

```
>k
```

You will be prompted to confirm this request. To have the debugger ignore the request, enter **n**; otherwise, enter **y**.

At this time, you can restart the program, quit the debugger, or enter other commands.

---

## Ending the HP Symbolic Debugger

To end your debugging session, enter the **q** (**quit**) command:

```
>q
```

You will be prompted to confirm this request. To have the debugger ignore the request, enter **n**; otherwise, enter **y**.

---

## Displaying Lines in the Program

There are several ways to display program lines in the source program window.

To display a particular source line, enter the **v (view)** command with the line number. For example, to display line 11:

```
>v 11
```

To move your view one or more lines forward in the program, enter the plus sign (+) and the number of lines you want to move. When moving forward or backward in the program, the source and location windows are adjusted accordingly. For example, to move five lines forward, enter:

```
>+5
```

To move your view backward in the program, enter the minus sign (-) and the number of lines you want to move. To move backwards five lines, enter:

```
>-5
```

---

### Note

When you reach the end (or beginning) of the source program using the + and - commands, no further movement may take place.

You can repeat a previous + or - command (see +5 and -5 above) by pressing **Return**. In this case, the previous count is kept and re-used.

---

There are a variety of other ways to specify an argument to the **v (view)** command to change the current viewing location. For example, a procedure can be viewed by executing a command similar to the following:

```
>v my_procedure
```

To view a file, execute a command similar to the following:

```
>v test1.c
```

To view a particular line in another file, execute a command similar to the following:

```
>v test1.c:104
```

To view a label in a procedure, execute a command similar to the following:

```
>v my_procedure#my_label
```

To view a procedure in a debuggable shared library, execute a command similar to the following:

```
>v my_procedure@my_library
```

For more information on viewing by procedure or file name, read the section “File Viewing Commands” in the chapter “HP Symbolic Debugger Commands.”

To display a procedure that has been called but is currently suspended at a given depth in the run-time stack, enter the **V (View)** command. The following example displays the procedure at depth two in the run-time stack. (Stack depth one is the current procedure’s caller, depth two is its caller, etc.)

```
>V 2
```

To view the current point of suspension in the source window, use the **V (View)** command with no arguments:

```
>V
```

---

**Note**

The source window automatically tracks where the program becomes suspended, and the **V (View)** is only needed after using the **v (view)**, **+**, or **-** commands.

The current view is automatically restored to the current point of suspension any time execution is resumed and again suspended.

---

---

## Controlling the Command Window Display

3

Command and program output is displayed one screen at a time in the command window. You can use the terminal keys **↓**, and the **Shift** *arrow* keys (or the equivalent scroll keys on your terminal) to scroll the command window. When you enter a command that requires more than the number of lines in the command window to display, the debugger displays enough lines to fill the command window then displays a **--More--** prompt at the bottom.

Use one of the following commands to continue from this prompt:

- Space Bar**      Displays one more window-full.
- Return**          Displays one more line.
- q**                Quits scrolling and ignores the rest of the pending output until another debugger prompt is issued.

To view command window output in a continuous stream, use the **sm** (**suspend more**) command to suspend the *more* feature. This is useful, for example, if you are using the debugger to dump the contents of data structures into a record-all file. **CTRL****S** may be used to temporarily suspend scrolling when the *more* feature is suspended. Use **CTRL****Q** to continue scrolling.

To return to single-window output, enter the **am** (**activate more**) command.

If you are using a playback file to debug a program and a command in that playback file causes the *more* feature to be used, the debugger automatically provides any carriage return that is required to continue the scrolling of text in the command window. This *is not* the same as suspending the *more* feature although the effect is similar.

---

**Note**              Output from the child process (program being debugged) normally also appears in the command window, but it is *not* controlled by the *more* feature.

---

---

## Changing the Source Window Size

To change the size of the source window, use the `w` (**w**indow) command and specify the number of lines you want for this window. For example, to change the size of the source window to 12 lines enter:

```
>w 12
```

The number of lines for the source window range from one to 21 for a 24-line terminal (the default is 15). Changing the size of the source window also changes the size of the command window.

An `hpterm` window in the X Window System is well suited to running the debugger. The maximum window height is only limited by the display device and font you are using. The debugger will track changes in X Window size at the (next) command prompt (subject to a minimum size of 12 lines by 60 columns).



---

## Displaying Assembly Code

**3** If you didn't use the symbolic debug option (`-g`) when compiling the program, you will be debugging in disassembly mode and will see a screen similar to the one shown in Figure 3-2. Even if you compiled with the symbolic debug option, you can debug in disassembly mode by entering the `td` (**t**oggle **d**isassembly) command as follows:

```
>td
```

In disassembly mode, the program is debuggable at the machine instruction level. The instructions shown are the reverse-assembled machine code for your program. Addresses are shown symbolically, as determined by the external symbols in the program's linker symbol table (see *a.out(4)*). Note that corresponding source-line numbers are displayed along with the absolute and symbolic address of each instruction. The values of all hardware registers are also shown in disassembly mode. A highlighted register value indicates its contents have been modified since the last debugger command.

It is also possible to debug shared libraries in disassembly mode. For information on this see the section "Debugging Shared Libraries in Disassembly Mode" in Chapter 6 of this manual.

```

hpترم
r0 00000000 01a80000 00002693 00000001 r4 68ff31e4 68ff31ec 40015500 00000012 /
r8 4001a800 00002000 40000000 0000007e r12 68ff5524 68ff5470 00000001 00000001
r16 40000000 40001000 00000000 0000a229 r20 01ac2630 68ff3300 00000000 01ac2630
r24 68ff31ec 68ff31e4 00000001 40000000 r28 00000000 00000001 68ff3330 00004197
pc = 0000a229.00002604  priv = 3  psw = jthlnxbCvmrQFD1  sar = 24
0x000025f0  load_mon+00f8  BV  0(2)
0x000025f4  load_mon+00fc  LDD  -56(30),30
0x000025f8  load_mon+0100  OR  0,0,0
0x000025fc  main  STW  2,-20(0,30)
0x00002600  main  +0004  LDD  48(30),30
> 69: 0x00002604  main  +0008  BL  get_input,2
0x00002608  main  +000c  OR  0,0,0
70: 0x0000260c  main  +0010  BL  load_month_table,2
0x00002610  main  +0014  OR  0,0,0
71: 0x00002614  main  +0018  OR  0,0,0
File: democ.c  Procedure: main  Line: 69
<<<< XDB Version R.07.05 HP-UX >>>>
No core file
Procedures: 6
Files: 2
>s
Starting process 2258: "democ"
>td
>

```

**Figure 3-2. The HP Symbolic Debugger Screen (Disassembly Mode)**

To return to source mode, enter `td` again.

Disassembly mode can also be used when only parts of your program were compiled with the symbolic debugger option. If the current viewing location is within non-debuggable code (such as a system library), and the debugger is in source mode, **No Source** will be shown in the source window. This indicates that it would be appropriate to use the disassembly mode.

You should refer to the appendix “Registers Displayed by the HP Symbolic Debugger” to see the registers displayed by the debugger in disassembly mode.

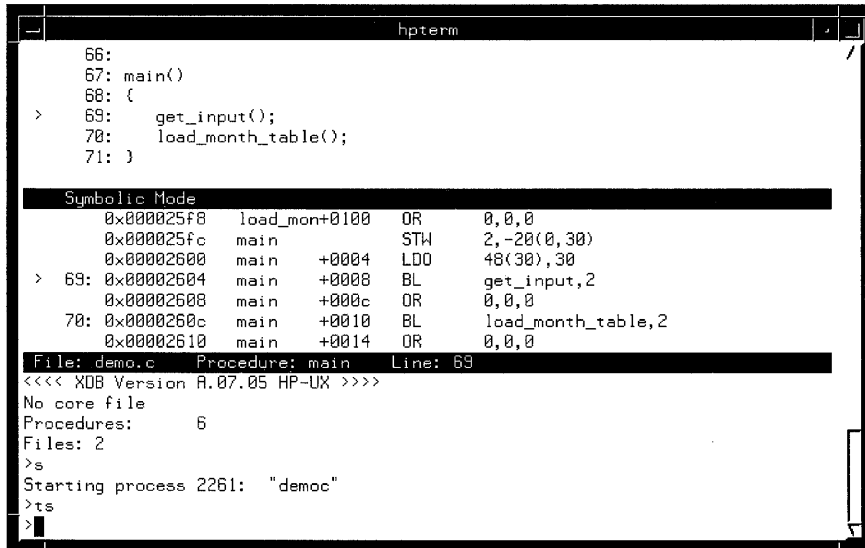
## Displaying Source and Assembly Code

3

To view both the source code and its matching assembly code, enter the `ts` (`toggle screen`) command. When you do this, the source window is divided into two windows, the top for source code and the bottom for assembly code as shown in Figure 3-3.

To view both source and assembly code, enter:

```
>ts
```



```
hpterm
66:
67: main()
68: {
> 69:   get_input();
70:   load_month_table();
71: }
```

Symbolic Mode				
0x000025f8	load_mon+0100	OR	0,0,0	
0x000025fc	main	STW	2,-20(0,30)	
0x00002600	main +0004	LDO	48(30),30	
> 69: 0x00002604	main +0008	BL	get_input,2	
0x00002608	main +000c	OR	0,0,0	
70: 0x0000260c	main +0010	BL	load_month_table,2	
0x00002610	main +0014	OR	0,0,0	

```
File: democ.c Procedure: main Line: 69
<<<< XDB Version A.07.05 HP-UX >>>>
No core file
Procedures: 6
Files: 2
>s
Starting process 2261: "democ"
>ts
>
```

Figure 3-3. The HP Symbolic Debugger Screen (Source and Disassembly Mode)

If the `ts` command is executed when the debugger is in source mode, the new screen will be in **Symbolic Mode**. This means that single stepping through the program will occur at the source line level. If, however, the `ts` command is executed when the debugger is in assembly mode, the new screen will be in the **Assembly Mode** and single stepping will occur at the assembly instruction level. Whether the debugger is in **Symbolic Mode** or **Assembly Mode** is indicated in the line separating the source and assembly screens. This mode may be toggled by executing the `td` (`toggle disassembly`) command.

To return to source mode, enter `ts` again in **Symbolic Mode**. Entering `ts` again in **Assembly Mode** will return the screen to the disassembly mode discussed in the preceding section.

---

## Stepping through the Program

3

The debugger lets you step through a program one (or more) statements at a time. If you're in disassembly mode, you execute one or more machine instructions; if you're in source mode, you execute one or more source statements. If you're in split-screen mode, the single step mode (symbolic or assembly) is indicated on the highlighted line separating the source window from the assembly window.

Stepping lets you closely examine program execution. During stepping, you can display and alter variables or perform other tasks between each statement.

The following command executes the next six statements (or machine instructions) then pauses:

```
>s 6
```

To repeat the step command, press **Return** or type a tilde (~) followed by **Return**. Using **Return** to repeat the step command will only cause one additional step to occur. Any count previously used is discarded.

If the program contains procedure calls and you do not want to step through the code in the procedures themselves, use the **S (Step)** command. The procedure call statements (or instructions) are treated as one step. To single step through a program and to treat procedure calls as one step, enter:

```
>S
```

Note that the **s** command cannot be used to step into system calls (those documented in Section 2 of the *HP-UX Reference*). In such cases, the debugger will always step over the call as if the **S** command was used.

If the program was linked with shared libraries, and the debugger's **-l** or **-s** option is not used to enable shared-library debugging, any single-step command used at a procedure call to a shared library will always step over the call as if the **S** command was used.

---

## Searching for a String in the Current File

This section explains how to locate certain text elements in the current source file. For example, you can search for variables and pointers by name or you can search for arithmetic expressions. You can search forward or backward in the current file for any text string. When you reach the end of the current file, searching starts again at the beginning. Likewise, when searching backwards and you reach the beginning of the current file, searching continues at the end of the file. A match to the search string will reset the current viewing location to the line where the match occurred.

The following example searches forward in the program for the string `r := 0` and stops at the first occurrence of it.

```
>/r:= 0
```

To search backward in a program for the string `const n = 10`, enter:

```
>?const n = 10
```

String searches can be case sensitive or case insensitive. Use the `tc` (**t**oggle **c**ase) command to control case sensitivity.

Search strings will be matched exactly (possibly disregarding case). All characters are significant, including blank spaces. If no match is found, the current viewing location does not change. Note that after locating an occurrence of the search string, the debugger may not always know what procedure the string was found in and will display **Procedure: unknown** in the location window.

To repeat a previous search command searching in the same direction, enter the `n` (**n**ext) command. To repeat the previous search command but search in the opposite direction, enter the `N` (**N**ext) command.

---

## Pausing during Execution

When you want to temporarily suspend the execution of the program to examine some aspect of it, such as a variable's value, set one or more *breakpoints* in the program. This must be done before starting the program, or when it is suspended by an existing breakpoint or an exception condition.

Breakpoints direct the debugger to stop execution at (immediately before executing) the specified line (or instruction). When you resume execution, the program will continue until this or another breakpoint is reached. While the program is suspended, you can enter any debugger command.

For more information on the commands used in this section, see the section "Breakpoint Commands" in the chapter "HP Symbolic Debugger Commands."

### Setting Breakpoints

To set a breakpoint in source mode, enter the line number before which you want execution to pause. There are several ways to specify a line number (see the appendix "HP Symbolic Debugger Commands" for a complete description).

If it is not an executable statement, the debugger sets a breakpoint at the first executable statement following that line. You can set breakpoints before step and run commands or after another breakpoint occurs.

The following example sets a breakpoint at line 10:

```
>b 10
```

When a breakpoint is set, the debugger displays in the command window the procedure and line number where the breakpoint is set and the source statement located at that line. If your terminal supports windowing, the line is marked in the source window with an asterisk (\*). From this point on, the debugger pauses each time line 10 is encountered.

To pause after a specific number of times the breakpoint is encountered during execution, enter the **b (breakpoint)** command followed by the *location* (line number) and *\number*. In the following example, a breakpoint is set at line 10 with a *count* of 2. The debugger pauses every other time line 10 is encountered.

```
>b 10 \2
```

To set a breakpoint at the first executable statement in all debuggable procedures in the program, enter:

```
>bp
```

To execute a series of debugger commands before each procedure is executed, enter the **bp** (**breakpoint procedure**) command with a *command list*. For example, to track the value of a particular variable, the following command sets a breakpoint at the beginning of each procedure and executes three commands (**Q**, **p** and **c**) at each of these breakpoints before continuing execution.

```
>bp {Q; p someglobal; c}
```

In this example, the **Q** (**Quiet**) command suppresses the debugger messages that are normally displayed when any breakpoint is encountered. The **p** (**print**) command displays the current value of the global variable `someglobal`. The **c** (**continue**) command resumes execution of the program. Without a **c** command, the program remains suspended at the breakpoint.

You can also set all-procedure breakpoints with the **bpt** and the **bpx** commands. The **bpt** command sets a trace breakpoint at the beginning and exit of all procedures. The **bpx** command sets a breakpoint at each procedure's exit.

To set procedure breakpoints only on procedures that are within a particular shared library, use a command similar to this:

```
>bp @myshare {Q; L}
```

The procedure breakpoints are set for all procedures. You cannot set individual procedure breakpoints in this manner. The **b** (**breakpoint**) command can be used to set individual procedure breakpoints, which will co-exist with any all-procedure breakpoint that may be set at the same location.

There are also C++ specific breakpoint commands:

- bpc** sets a breakpoint in all the member functions of a particular class.
- bi** sets an "instance" breakpoint (a breakpoint on a member function for a particular object only).
- bpo** lets you set breakpoints on a set of overloaded functions.



## Resuming Execution After a Breakpoint

Once the debugger pauses for a breakpoint and you have finished entering commands at that breakpoint, enter the `c` (`continue`) command:

3

```
>c
```

This causes execution to continue until another breakpoint is encountered, an exception (signal) occurs, or the program terminates.

## Listing Breakpoints

To list the breakpoints that are set in the program, enter the `lb` (`list breakpoints`) command as follows:

```
>lb
```

When the `lb` (`list breakpoints`) command is executed, information about each breakpoint is displayed. For example, two breakpoints are shown below. The first number on each breakpoint line is the debugger-assigned breakpoint number, which you use with other commands (such as `db` (`delete breakpoint`)) to refer to a particular breakpoint. The number following `count` is the number of times the source statement will be encountered before the breakpoint is recognized. The breakpoint's state (active or suspended) is listed next, followed by the line at which the breakpoint is set and the source statement on that line.

```
Overall breakpoints state:  SUSPENDED
    1: count: 1 Active      sortall: 12: abc += 1;
    2: count: 5 Suspended  fixit: 29: def=abc >> 4;
```

To list the breakpoints only within a single shared library, enter a command similar to this:

```
>lb @myshare
```

## Deleting Breakpoints

To delete a breakpoint, enter the debugger-assigned number of the breakpoint (see the previous section “Listing Breakpoints”) with the `db` (delete breakpoint) command.

For example, to delete the breakpoint whose number is 2, enter:

```
>db 2
```

If you do not enter the breakpoint number, the breakpoint at the current line, if any, is deleted. If there is no breakpoint at the current line, the debugger lists all of the breakpoints.

To delete all breakpoints (including all all-procedure breakpoints), enter:

```
>db *
```

To delete only all-procedure breakpoints (only those breakpoints set by the `bp` (breakpoint procedure), `bpt` (breakpoint trace), or `bpx` (breakpoint exit) commands), enter the following respective commands:

```
>dp
```

```
>Dpt
```

```
>Dpx
```

To delete the procedure breakpoints only within a single shared library, enter a command similar to this:

```
>dp @myshare
```

---

## Displaying Data

Whenever program execution pauses, you can display the contents of simple variables, arrays, structures and pointers.

3

To display data, use the **p** (**print**) command. Various options and formats are available for greater control over displaying data.

The example below shows how to display the value of the variable **fob** in a form that is consistent with how it is declared in the language used (if the variable is an integer variable, for example, the value is expressed in decimal form):

```
>p fob
```

To display a variable or expression in a hexadecimal format, enter a print command in a form similar to this:

```
>p fob\x
```

To interpret an expression as a long decimal integer, enter the print command in this form:

```
>p hanoi\D
```

To re-display the variable used with the last command, enter:

```
>p .
```

To display the contents of the location that is 30 bytes ahead of the last displayed data item in memory (using HP C syntax), enter:

```
>p *(&.+30)
```

This assumes the specified location begins a data item of the same type and size as the previously displayed them.

Field members of structures, unions, or records can also be displayed. They are referenced in the same manner as they would be in your program. For example:

```
>p employee_rec.date_of_birth.month  
month = 11
```

If individual fields are not specified, the entire composite object is printed, indented to show its actual structure. In this C example, all nested structures are shown:

```
>p employee_rec
employee_rec = struct {
  name = "Joe Q. Public";
  date_of_birth = struct {
    year = 1960;
    month = 11;
    day = 7;
  }
  ssn = 532892398;
}
```

Pointer variables can be displayed as addresses, or dereferenced to show the object being referenced. For C programs, commands like the following might be used:

```
>p ptr\X
ptr = 0x40042a6c
>p *ptr
0x40042a6c 5
```

Note that the default format used to display the dereferenced object depends on its type. In this example, `ptr` might have been declared as `int *ptr`.

Pointers to records or structures are also easily printed. Fields in the object pointed to are referenced in the same manner as they would be in your program. In this Pascal example, elements of a linked list are examined:

```
>p recptr
RECPTR = 0x40008020
>p recptr^.next
next = 0x40008040
>p *(recptr^.next)
0x40008040 record
  KEY = 'zombie';
  HASHVAL = 349;
  NEXT = 0x40008060;
end
```

The `p+` and `p-` commands are useful for traversing the elements of an array.

To display the next data item in the array using the current format (the format most recently used) and data item size, enter the print command in this form:

**3**

```
>p+
```

This interprets the next sequential data item after the one previously printed, which is assumed to be of the same size and type.

To display the next data item using a format different from the current one, use this form:

```
>p+ \x
```

To display the previous data item in the array using the current format and data item size, enter the print command in this form:

```
>p-
```

To display the previous data item using a format different from the current one, use this form:

```
>p- \x
```

---

## Modifying Data

When you need to alter the value of a variable, array item, field, or pointer, use the `p` (`print`) command followed by an assignment expression. The expression should be entered in the same syntax as the language in which the program is written.

This example changes the value of the variable `A1` to 30 (HP C, HP C++, or HP FORTRAN 77):

```
>p A1=30
```

The following example sets the variable `j` to the value of the expression `j + 17`:

```
>p j = j + 17
```

or

```
>p j += 17
```

In HP Pascal, this same example is:

```
>p j := j + 17
```

If you want to avoid the display of the result of the assignment (for example, because the assignment is in an assertion or a breakpoint command list), use the `pq` command.

---

## Tracing Function and Procedure Calls

When a program contains several functions or procedure calls, you might need to know the sequence of calls that led to the current point of suspension. Displaying this sequence is called “viewing the stack”. To view the stack, enter the **t** (**trace**) command:

3

```
>t
 0 f2 (i = 3)    [t.c: 17]
 1 f1 (i = 2)    [t.c: 11]
 2 main ()      [t.c: 5]
```

In this example, the debugger lists:

- The stack depths: 0, 1, and 2 (0 is always the “top” of the stack).
- The name of the procedure at each depth and their parameter values:
  - f2 (i = 3)
  - f1 (i = 2)
  - main ()
- The source file and line number where it is suspended (within their respective procedures):
  - [t.c: 17]
  - [t.c: 11]
  - [t.c: 5]

If you also want to see the value of local variables at each depth of the stack, use the **T** command.

---

## Navigating the Execution Stack

This section uses the program `navstack.c` to show how to use the View, down, up, and top debug commands.

```
#include <stdio.h>

main()
{
    stack(5);
    exit(0);
}

stack(depth)
    int depth;
{
    int local;

    local = depth;
    if (depth) stack(depth - 1);
    printf("local = %d\n",local);
}
```

Before debugging `navstack.c`, you need to compile it using the `-g` compiler option. For example:

```
cc -g navstack.c -o navstack
```

Once the program has been compiled with debug information, you can execute the following command to run the debugger:

```
xdb navstack
```

Next, set a breakpoint where the procedure `printf` is called in the program, type at the prompt (`>`):

```
>b 16\t
```

You can now run the program by typing:

```
>r
```



The program will stop at the breakpoint you set. You are now ready to use the View, down, up, and top debug commands.

### Using the down Command

3

The debugger **down** command can be used to move down four levels in the execution stack. For example, type:

```
>down 4
```

Result displayed:

```
stack level: 4
```

which means you are currently at level 4 in the execution stack. To verify this, type:

```
>p $depth
```

Result displayed:

```
$depth = 4
```

Note that **\$depth** is a special variable that contains the current execution stack level.

If you type **l** at the prompt, you will get a listing of the values of the local variables for the current procedure. For example:

```
depth    = 4  
local    = 4
```

### Using the up Command

You are currently at level 4 in the execution stack. To move up two levels in the execution stack, type:

```
>up 2
```

Result displayed:

```
stack level: 2
```

If you need to change the value of the variable `local` at stack level 2, you would type:

```
>p local = 15
```

Result displayed:

```
local = 15
```

### Using the top Command

The `top` command is used to get to the top of the stack. To try this command, type:

```
>top
```

Result displayed:

```
stack level: 0
```

To look at the values of the local variables of the current procedure, type:

```
>l
```

Results displayed:

```
depth    = 0  
local    = 0
```

## Using the View Command

3 The **V (View)** command sets the current viewing location to the location of the next instruction that would be executed in the procedure at the specified stack depth. For example, move the current viewing location to line 6 in the program by typing:

```
>v 6
```

The marker (>) points to the current viewing location in the source window (line 6).

Next, to see where execution would continue in the procedure at level 4 of the execution stack, type:

```
>V 4
```

where **V** indicates viewing of the current point of program suspension and **4** indicates viewing is to take place at level 4 of the execution stack. To verify that the current viewing location is stack level 4, type:

```
>p $depth
```

Result displayed:

```
$depth = 4
```

The location shown by typing **V 0** has the next instruction that will be executed when the program is stepped or continued. This is the current point of suspension of program execution.

To allow the program to complete execution, type the continue command:

```
>c
```

The program prints the value of the variable **local** at each stack level, including the one modified above. Results displayed:

```
local = 0  
local = 1  
local = 15  
local = 3  
local = 4  
local = 5
```

---

## Capturing and Rerunning a Debugger Session

If, before a debugging session, you think you might need to retrace your steps, you can capture the debugger commands you used during the session. You can save the debugger commands in a file and “play them back” during a subsequent session.

To write the debugger commands to a file, start the debugger using the `-r` option. The example below invokes the debugger and directs it to echo all commands to the file `acdebug`:

```
xdb -r acdebug test1
```

If you are already in the debugger, execute this command instead:

```
>>acdebug
```

To play back the file in subsequent debugger sessions, invoke the debugger with:

```
xdb -p acdebug test1
```

This file may also be played back from inside the debugger using the `<` command:

```
>< acdebug
```

To interactively play back each command, execute:

```
><<acdebug
```

For more information on capturing and rerunning a debugger session, read the section “Record and Playback Commands” in the chapter “HP Symbolic Debugger Commands.”

---

## Saving and Restoring the Debugger State

3

When you are running the debugger and want to save the current set of breakpoints, macros, and assertions in a file, the debugger provides a way to do this. You can also restore this information when you re-invoke the debugger on the same *objectfile* at some later time.

To save the current set of debugger breakpoints, macros, and assertions in the file `my_cmds`, you would execute a command similar to the following:

```
>ss my_cmds
```

Information saved in the file `my_cmds` can be restored by using the `-R` option with the file name when the debugger is invoked. For example, you would execute:

```
xdb -R my_cmds test1
```

Note that the file `my_cmds` can be used as a playback file; however, this will bypass the verification the debugger provides with the `-R` option (see “Save State Limitations” in the appendix “Limitations and Hints”).

---

## Displaying Character Data and Using NLS

The HP symbolic debugger provides features which aid in the debugging of programs that deal with textual data; that is, characters or strings. A useful feature of `xdb` is its ability to display such data using the semantics described by the HP NLS (Native Language Support) model. For additional information, see the manual *HP-UX Concepts and Tutorials: Native Language Support*. The `hpnl5(5)` entry in the *HP-UX Reference* also provides a good overview of this model. See also the section “Setting Up the Locale” under the section “Customizing the Symbolic Debugger Environment” in this chapter.

When the debugger is invoked, the current setting of the `LC_CTYPE` environment variable determines the *locale* (or language) that character-based operations will be performed in. In the debugger’s case, it determines the “printability” of a given character. Note that if `LC_CTYPE` is not set, it defaults to the current setting of the environment variable `LANG`. If `LANG` is not set, the default C locale is used (see the file `/usr/lib/nls/config` for the *locale* names that may be used).

The debugger provides the `c` formatting character for displaying data bytes as individual characters. The `s` format is used for string types. (Note the definition of *string* depends on the programming language being used.)

For example, assume your program contains the following declaration (in C):

```
char *prompt = "next?"
```

The following debugger command can be used to print the string:

```
>p prompt
prompt = "next?"
```

which, in this case, is equivalent to:

```
>p prompt\s
prompt = "next?"
```

as the `s` format is the default for objects of type “pointer-to-char.”

Individual character elements of the string can be examined with something similar to:

```
>p prompt[3]\c
0x1003  't'
>p *prompt\6c
0x1000      n      e      x      t      ?  \000
```

Here the leading hex number is the address of the object being printed. Note the *null* character that terminates the C string. This is shown as an “octal-escape,” which is an 8-bit value displayed as a backslash followed by three octal digits: `\nnn`.

By default, `xdb` will display all non-ASCII characters as a octal-escapes. However, there is a debugger special variable `$print` that controls this behavior. Initially, it is set to the value `ascii`.

---

**Note** The following examples contain 8-bit characters that require HP terminals to enter and display. The debugger makes no assumptions about the display device or keyboard you are using; all it knows about is the current setting of the NLS environment variable `$LC_CTYPE`. It is up to the user to ensure that the keyboard language (usually set with softkeys) and character-set (or font) are configured properly.

---

To illustrate, suppose you had just executed the following statement in an HP Pascal program:

```
green := 'grün';
```

The 8-bit character `ü` would normally be displayed as an octal-escape:

```
>p green
GREEN = 'gr\317n'
>p green\5c
0x107b0  \004      g      r  \317      n
```

(Note the string size (4) appears as the first element in the Pascal example above. This reflects how Pascal *strings* are stored in memory.)

However, by issuing the following debugger command, only *non-printable* characters in the current locale (LC\_CTYPE) are displayed as octal-escapes:

```
>p $print = native
$print = native
```

Given the above example, ü is now printable, even though it is not an ASCII character. This then gives the following results:

```
>p green
GREEN = 'grün'
>p green\5c
0x107b0  \004      g      r      ü      n
```

This example assumes that the locale was set appropriately from the command shell, as in this *cs*h example:

```
% setenv LC_CTYPE german
```

There is a third possible value for `$print`, the value `raw`. This causes all 8-bit character bytes to be output with no distinction made between printable and non-printable. A word of caution: this may have detrimental effects on your terminal, and is not generally recommended.

## Wide Characters

The ANSI/C language supports a *wide-character* base-type, which represents a universal encoding of character data (see *multibyte(3C)*). Items declared with the ANSI/C type `wchar_t` can also be manipulated as textual data by the debugger, which automatically provides the mapping between the external character set (as determined by the current locale) and its internal representation. This mapping is invoked by the wide-character string formatter `\W`, and the wide-character formatter `\C`.

For example, assume your ANSI/C program contains the following declaration (note the use of the wide-character prefix operator `L`):

```
static wchar_t *wstr = L"Hello, world.";
```



It is irrelevant how individual elements of `wstr` are stored, as they are automatically converted to their external equivalents upon display.

```
>p wstr\W
wstr = L"Hello, world."
>p wstr[0]\C
0x1006 L'H' (0x00000048)
```

3

The `L` prefix is displayed here to indicate the mapping was performed. The formatters used above would be unnecessary if `$print` were set to `native`, as they are the default formats for type `wchar_t` in that case. (When `$print` is set to `ascii`, the default format for `wchar_t` is `X`.)

The mapping from multibyte strings to wide-characters is also performed by the debugger when the `L` prefix is used with character- or string-constant expressions. For example:

```
>p wstr = L"fun"
wstr = L"fun"
>p wstr[0] = L'r'
0x1006 L'r' (0x00000072)
```

Wide-characters are most useful for representing extended (16-bit) character sets, such as *Traditional Chinese*, using the locale `chinese-t`. If you're using a terminal or window that supports input and output of this type of data, you have the ability to display and modify wide-character program variables as easily as if they contained simple ASCII text.

---

## Separate Interfaces (Debugging Screen Applications)

Here is an example program which uses the *curses(3X)* library to read a line of characters from the terminal with no echo. In this example, the line is printed after **Return** is pressed to verify that input has occurred.

3

```
1: #include <stdio.h>
2: #include <curses.h>
3: main()
4: { int i=0; char str[256]; int c;
5:     initscr();
6:     nonl();
7:     cbreak();
8:     noecho();
9:     keypad(stdscr,TRUE);
10:    idlok(stdscr,TRUE);
11:    nodelay(stdscr,TRUE);
12:    do {    c = getch();
13:        if (c == -1) continue;
14:        str[i++] = (char) c;
15:    } while (str[i-1] != '\015'); /* wait for return */
16:    printf("%s\n",str);
17:    endwin();
18: }
```

Suppose you wanted to debug this program by setting a breakpoint on line 14 and observing the processing of each character as it is entered. If the program runs in the same window/terminal as the debugger, as soon as the breakpoint is hit the special *termio(7)* modes are lost and no further interrupts occur until return is pressed.

In order for the program's interface handling to work undisturbed by *xdb*'s interface handling, the program should have its interface directed to another window/terminal as described in the section "Terminal Support" in the chapter "Introducing the HP Symbolic Debugger."

For example, using X windows, one would select two windows: one for the program to use and another for the debugger. In the window selected for the program's use, execute the commands:

3

```
$ tty   
/dev/pty/ttyp7  
$ sleep 10000000 
```

This gives you the device name for the window, and puts the shell in that window to sleep so it does not compete for input with the program you are debugging.

Assume that you have compiled this program with the `-g` option and left the program in `a.out`. In the window selected for the debugger, invoke the debugger with:

```
$ xdb -i /dev/pty/ttyp7 -o /dev/pty/ttyp7 -e /dev/pty/ttyp7 a.out
```

Now a breakpoint at line 14 will not interfere with the terminal state created by calls to routines in the *curses* library, since the debugger and the program are talking to different pseudo-terminals (pty).

While the program's interface is now separate, its controlling terminal is still the debugger's window. This means that keyboard generated signals, such as SIGINT (interrupt), must come from the debugger's window.

If separate windows are often necessary, it may be useful to create special scripts (in an appropriate directory in your PATH). Name the first script `wxdb`. It will contain:

```
/usr/bin/xdb -i $T -o $T -e $T $0
```

---

**Note**

On earlier systems that do not have SIGWINCH support, you should include the following command line:

```
eval '/usr/bin/X11/resize'
```

at the beginning of the `wxdb` script to set the `LINES` and `COLUMNS` environment variables for the new window.

---

**3**

Name the second script `Xxdb`. It will contain:

```
export T='tty'  
/usr/bin/X11/hpterm -name Xxdb -e wxdb $@
```

The `-name Xxdb` is optional; see *hpterm(1)*. While `hpterm` is recommended for use with `xdb`, `xterm` can be used if you prefer.

Executing the script `Xxdb` as shown below will create a new window for `xdb`, but leave the program's interface in the window where you invoked the debugger.

`Xxdb other_xdb_options objfile`

Note that the `sleep` command is not necessary (as in the previous example) since the `hpterm` running in the foreground has the same effect.

The window where you invoke the `Xxdb` debugger script does not need to be the same type of terminal as is invoked in the script. It can be any terminal emulator program that the program being debugged requires. Since the program will inherit its environment variables from `xdb`, it may be necessary to set breakpoints where the program reads them and assign more appropriate values (for example, for `TERM`, `LINES`, or `COLUMNS`).

## **Separate Environments by way of Adoption**

**3**

In some cases, it may be necessary to have completely separate environments for the debugger and the process being debugged. For example, the program may include non-debuggable libraries that use the same environment variables as the debugger for terminal set-up or NLS support. In these cases, adoption can provide a solution. Starting the program in one terminal/window and adopting with a debugger in another will allow each a totally disjoint environment. For more information on how to adopt a program to debug it, see the section “Adopting a Running Process” in this chapter.

---

## Executing Commands At Each Instruction

When you suspect that bugs might be occurring at several places in a program, or you have a bug that is especially difficult to track down, you can direct the debugger to execute one or more commands before *every* machine instruction is executed. For example, you might want to track the value of one or more variables through a series of detailed calculations.

The commands that you execute in this manner are called assertions.

The following example shows how to display the variables `payw8` and `paynet` before each instruction is executed:

```
>a {p payw8; p paynet}
```

The `if` command is very useful in assertion and breakpoint command lists. For example, if `paynet` should always be less than 23000, and you want to know where its value becomes greater, the assertion:

```
>a {if (paynet >=23000) {x}}
```

will stop the program when `paynet` exceeds the legal value.

---

## Using Macros

3

Macros are words that represent one or more debugger commands or expressions. You create macros by entering names for them and specifying the commands or expressions for which they stand. Macros are very useful for representing a group of commands that you execute often. You do not have to re-enter the commands; just enter the macro name for them.

The following command defines the macro `name`. Every time `name` is used, the corresponding commands are executed:

```
>def name p employee->personal_data.name.first;p employee->personal_data.name.last
```

Macro expansion can be enabled or disabled with the `tm` (toggle macros) command. Initially, macro expansion is disabled.

Note that macros have no arguments, and can only be used to represent entire commands or expressions.

---

## Altering the Execution Sequence

When the program is paused at a breakpoint or you are single-stepping through it, you can change the normal execution sequence of the program and cause it to resume at a different line. To resume execution of a program at a specific line, use the **g** (**goto**) command with the appropriate line number. The new line must be in the same procedure as the current one.

The following example directs the debugger to change the next line to execute to be line 600:

```
>g 600
```

Then you would use a **continue** or **step** command to begin execution at line 600:

```
>c
```

The above example allowed you to move to an absolute line position. If you need to move to a relative line position from your current position, you would execute a command similar to the following:

```
>g +10
```

Executing this command will move you forward 10 lines from your current position.

If you want to always skip a particular line (located at line *number*), you could execute a command similar to the following:

```
b number {Q;g +1;c}
```





## Getting Help

When you need help with the format of a debugger command or can't remember which command performs a particular function, use the **h** (**help**) command as follows:

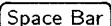

```
>h [ topic ]
```

The **h** (**help**) command with no argument shows all of the help text. The *topic* is either a specific command name (short form) or a task keyword describing groups of related items. Command name *topics* print the syntax and a brief description of that command. Task-related *topics* print the commands and other items related to that task. For a list of the available task-related *topics*, execute this command:

```
>h help
```


Help text is displayed one window at a time using *more(1)*. You can use the terminal keys , and the  *arrow* keys (or the equivalent scroll keys on your terminal) to scroll the command window. The **more** command displays enough lines to fill the command window then displays a **--More--** prompt at the bottom.

Use one of the following commands to continue from this prompt:

- |   |   |
|---|---|
|  | Displays one more window-full.  |
|  | Displays one more line.   |
| q   | Quits scrolling and ignores the rest of the help information until another debugger prompt is issued. |

The **more** feature of **help** cannot be suspended.

To create a copy (e.g., **prt.help**) of the help text suitable for printing, execute this command:

```
nroff /usr/lib/xdb.help.nro > prt.help 
```

This file (**prt.help**) includes bolding and underlining of the appropriate text. If your printer does not understand this, it can be removed by using *col(1)*.

---

## Adopting a Running Process

The `xdb` command is capable of adopting and debugging a process that was started outside of the debugger. This is accomplished by invoking `xdb` with the `-P` option. For example, this command:

```
xdb -P 12446 sort
```

adopts the process 12446 which must be running a program called `sort`.

To adopt a process, the effective user ID's of the debugger and the process to be adopted must match, or the effective user ID of the debugger must be `root`. When a process is adopted, it halts, and `xdb` displays where the program is halted, at which point the program can be debugged. If the user quits the debugger without killing the process, `xdb` removes all breakpoints from the process, and allows it to continue running.

---

**Note** The debugger cannot adopt a sleeping process (see *sleep(3)*). The easiest way to make a program adoptable by `xdb` before a certain condition arises (such as an error condition) is to make it execute a simple infinite loop ("busy wait"). Once you get control of the process, you can get out of the loop with the `g` (`goto`) command, or by changing the value of the variable that enables the loop.

The debugger *can* adopt a process that is suspended while waiting for an event or I/O.

However (for Series 600/700/800 only), if the process is waiting (blocked) in a system call, and the program was linked with the shared C library (`/lib/libc.sl`), a stack trace command will not be possible until a breakpoint has been set in your (non-shared library) code and the process has continued past the block to reach the breakpoint.

---

Because an adopted process behaves much like a process initiated under the debugger, it is easy to lose sight of the fact that it is still running in the same environment as before its adoption. In particular, signals generated from the keyboard, such as `SIGINT` and `SIGTSTP`, will not reach the adopted process

(which must have been running in the background or attached to another window or terminal before being adopted).

3

If you are accustomed to using a keyboard-generated interrupt to regain control of the child process, this will not work for an adopted process. You must use a `kill -2 pid` from another window or terminal (or after putting the debugger into the background) to send a `SIGINT` to the child. The debugger itself will not receive non-fatal signals such as `SIGINT` while it is waiting for an event in the child process. However, if the child process is running in the foreground in another window or terminal, a keyboard-generated interrupt in the child's window can probably be used to regain control.

---

**Note**

If your program is linked with shared libraries which you wish to debug, you must run the command

```
pxdb -s enable program
```

before the program is started. For more information on this, read the section "Debugging Shared Libraries in an Adopted Process" in Chapter 6 of this manual.

---

---

## Debugging a Program that Caused a Core Dump

The example program in this section has an error in it that causes a "core dump." A "core dump" is a core image of the process that the HP-UX system writes to a file called `core`, at the instant the process is terminated (see `core(4)`).

3

### Generating a Core Dump

To generate an example core dump, copy the program `gen_core.c` from the `xdb_demos` directory to your directory and name it `test_prog.c`. To do this, execute the command:

```
cp /usr/lib/xdb_demos/gen_core.c test_prog.c
```

Here is a listing of the program:

```
.  
. .  
14 set_to(x,y)  
15 int *x;  
16 int y;  
17 {  
18     *x = y;  
19 }  
20  
21 thusly(x,z)  
22 int x,z;  
23 { int i;  
24     i = x;  
25     set_to(&x,0);  
26     for (; (i++)<10; i++)  
27         if (i==z) set_to(x,i);  
28     printf("%d ",x);  
29 }  
30  
31 main()  
32 { int i;  
33     for (i=0; (i++)<10; i++) {
```

3

```
34     printf("%d ",i);
35     thusly(1,i+4);
36 }
37 printf("\n");
38 for (i=1; (i++)<10; i++) {
39     printf("%d ",i);
40     thusly(1,i+4);
41 }
42 printf("\n");
43 }
```

After you have copied and re-named the program, compile it using this command:

```
cc -g test_prog.c -o test_prog 
```

The above command prepares the program for debugging and gives the executable file the name `test_prog`. To run the program, execute

```
test_prog 
```

After you executed this command, you will see the following information on your screen:

```
1 0 3 0 5 0 7 0 9 0
Bus error(coredump)
```

This message tells you that a coredump was generated as a result of a bus error. If you list the files in your current directory, you will see that there is a new file in it called `core`. It is a good idea to re-name this file to prevent the core image you are currently examining from being overwritten by another coredump. You can re-name it `mycore`.

## Debugging the test\_prog Program

To debug the program `test_prog`, execute the following command:

```
xdb test_prog mycore
```

Your display will look like this:

```

11:  * and follow the instructions in the manual.
12:  */
13:
14:  set_to(x,y)
15:  int *z;
16:  int y;
17:  {
> 18:     *x = y;
19:  }
20:
21:  thusly(x,z)
22:  int x,z;
23:  { int i;
24:     i = x;
25:     set_to(&x,0);
File: test_prog.c  Procedure: set_to  Line: 18
Copyright Hewlett-Packard Co. 1985,1987-1992. All Rights Reserved.
<<<< XDB Version A.09.00 HP-UX >>>>
Core file from: test_prog
Child died due to: bus error
Procedures:      3
Files: 1
>

```

**Figure 3-4. Debugging the Program test\_prog**

At the command line prompt `>`, you can enter any symbolic debugger commands that you want except for those commands that require the process you are debugging to be activated. Examples of command that should not be used with core files are:

- Job control commands such as: `r`, `c`, `s`, etc. Using one of the job control commands will start execution of the object file. While this process exists, the core file is inaccessible.
- Breakpoint commands such as: `b`, `ba`, `bpc`, etc. Attempting to use a breakpoint command without an executing object file (process) will produce an error message.

Commands that allow you to view the file you are debugging can be executed at the debugger's command line prompt `>`. Examples of such commands are:

- File viewing commands such as `L`, `v`, `va`, etc.
- Data viewing commands such as `p`, `lp`, `lr`, etc.
- Stack viewing commands such as `t` and `T`

If you look at the figure “Debugging the Program `test_prog`” (Figure 3-4), you will see in the command window the message:

```
Child died due to: bus error
```

Here the debugger reports the signal that caused the coredump to occur (in case you didn't or couldn't record the message from the kernel). You should also notice that the marker (`>`) in the source window is pointing at or just after the assignment in the procedure `set_to` (line 18). This is the line where the process terminated and the coredump was generated. The stack should contain the values of the variables that were passed to the procedure `set_to`. To view the value of each variable passed to this procedure, execute this command:

```
>t Return
```

The following information is displayed.

```
>t
0 set_to (x = 00000000, y = 6)    [test_prog.c: 18]
1 thusly (x = 0, z = 6)        [test_prog.c: 27]
2 main ()                      [test_prog.c: 40]
>
```

Notice that the top of the stack (depth 0) shows that the variable `x` was passed a null pointer. This is an indication that the problem with the program must have occurred at or before the call to the procedure `set_to`. Looking down the stack to depth 1, you find that the procedure `thusly` called `set_to` from line 27 of the program. To view line 27, execute this debugger command:

```
>V 1
```

Your display would look like this:

```

20:
21: thusly(x,z)
22: int x,z;
23: { int i;
24:   i = x;
25:   set_to(&x,0);
26:   for (; (i++)<10; i++)
> 27:     if (i==z) set_to(x,i);
28:   printf("%d ",x);
29: }
30:
31: main()
32: { int i;
33:   for (i=0; (i++)<10; i++) {
34:     printf("%d ",i);
File: test_prog.c Procedure: thusly Line: 27
Procedures: 3
Files: 1
>v 1
>t
0 set_to (x = 00000000, y = 6) [test_prog.c: 18]
1 thusly (x = 0, z = 6) [test_prog.c: 27]
2 main () [test_prog.c: 40]
>

```

**Figure 3-5. Viewing the Procedure that Called set\_to**

The marker (>) in the source window is now pointing at line 27 of the program. If you look at the procedure call `set_to`, you will see that the variable `x` is passed as a value instead of being passed by reference. The cause of the bus error is a missing address operator that must be prefixed to the variable `x`. Line 27 of the program should look like this:

```
if (i==z) set_to(&x,i);
```

Making this correction will cause the program to execute correctly.



---

## Mapping of Source Directories

The debugger normally locates a source file for a given section of the program by using the file names recorded in the symbolic debug tables when the program was compiled. This path is identical to what you provided to the compiler and is not always sufficient for the debugger, especially if the program is debugged in a directory different from where it was compiled.

If you are debugging a program from a directory other than where it was compiled, you can use the debugger's `D` command or `-d` invocation option to tell the debugger where the source files are located. This command and option are sufficient if all of your source files are located in a single or small number of directories; however, if they are located in several directories, you may want to use the debugger's `apm` command.

This section provides a scenario for three uses of the `apm` command. The topics covered are:

- A Scenario for Using the `apm` Command
- Example 1: Both Old Path and New Path are Provided
- Example 2: Stripping Part of an Old Path
- Example 3: Prefixing a Path

## A Scenario for Using the `apm` Command

In this scenario, assume that you are using separate machines for development (compilation) and testing (debugging). In your development environment, you have the following directory:

```
/myprod/newdev/src
```

This is the top directory under which the sources for your related programs are kept. Since you compile several distinct parts, you keep the sources for the different parts in several subdirectories.

```
/myprod/newdev/src/driver  
/myprod/newdev/src/interface  
/myprod/newdev/src/core  
...
```

Now suppose you compile your programs in `/myprod/newdev/bin`, and you give the compiler relative path names of the form:

```
../src/driver/filename
```

to locate your source. The result is that the debug information for your driver program contains the same relative path names. Thus, unless you tell the debugger otherwise, it will look for the source to `main()` in `../src/driver/main.c`, relative to the directory where you invoke the debugger.

Next assume that testing (debugging) is done on a different machine, and your sources are NFS-mounted from the development machine. The path to your top-level source directory now looks like this:

```
/mnt/project/src
```

If you now want the debugger to be able to locate your source file, you could use the `D` command while in the debugger or the `-d` option when you invoke the debugger. If you use the `D` command to help locate your source files, you would execute commands similar to these:

```
D "/mnt/project/src/driver"  
D "/mnt/project/src/interface"  
D "/mnt/project/src/core"  
...
```

Alternatively, the `-d` invocation option accomplishes the same thing:

```
xdb -d /mnt/project/src/driver -d /mnt/project/src/interface \  
-d /mnt/project/src/core ...
```

Note that limitations to the `D` command and the `-d` option are:

- Very large software systems may have several subdirectories, each requiring a separate `D` command or `-d` option.
- If you have identical file names under any two or more subdirectories, the debugger can easily display the wrong file, depending on the order you have issued the `D` commands.

It is because of these limitations that the `apm` debugger command is recommended for use over the debugger's `D` command and `-d` command line option. The following three examples use the scenario in this section to explain different ways of using the `apm` command.

### Example 1: Both Old Path and New Path are Provided

Rather than enumerating each subdirectory, you could instead use the following debugger command:

```
apm ../src /mnt/project/src
```

This causes the debugger to translate all paths of the form:

```
../src/driver/main.c  
../src/core/eval.c
```

to

```
/mnt/project/src/driver/main.c  
/mnt/project/src/core/eval.c
```

All files are now easily located by the debugger, without any ambiguities that could have been introduced by the D command.

### Example 2: Stripping Part of an Old Path

Suppose your current directory is `/mnt/project/src` when you are debugging one of your programs. You can still use the path map illustrated in the previous example, or you could use the following path map:

```
> apm ../src
```

Since a replacement path was not specified, the specified old path is just stripped from all known file paths it applies to, rendering paths of the form:

```
../src/driver/main.c  
../src/core/eval.c
```

into

```
driver/main.c  
core/eval.c
```

3

### Example 3: Prefixing a Path

Assume that you compiled your programs while your current directory was `/myprod/newdevel/src`, and you issued names like `driver/main.c` to the compiler.

Given that your sources are now mounted under `/mnt/project/src`, you can debug anywhere on your test machine using one command to tell the debugger where to find the sources:

```
> apm "" /mnt/project/src
```

Since you didn't specify an old path, the debugger just prefixes the replacement path to the file name, rendering all file paths of the form:

```
driver/main.c  
core/eval.c
```

into

```
/mnt/project/src/driver/main.c  
/mnt/project/src/core/eval.c
```

Note that this is the only case where you need to surround a path by double quotes.

## HP Symbolic Debugger Commands

---

This chapter describes the commands recognized by the HP Symbolic Debugger. These commands are arranged by function in alphabetical order and can be entered in short form (abbreviated) or long form (spelled out). If you use the long form, space between command words is usually optional.

4

---

### Entering Commands

The HP Symbolic Debugger keeps track of the current file, procedure, line and data locations of the executing program. The current file, procedure, and line are always displayed in the *source* and *location windows*, but their values do not necessarily correspond to the point at which execution is suspended.

A program is suspended when control is transferred from the program to `xdb` by encountering a breakpoint or exception condition. Many commands use these current locations as defaults.

Note that when a program becomes suspended, the source corresponding to the suspension point is forced to be displayed in the *source window* and the items displayed in the *location window* reflect exactly where the program is suspended. When you enter a search or `v` (**view**) command, items displayed at the *location window* may change and will be reflected by a change in the contents of the *source window* (i.e., the focus is changed).

To realign items displayed at the *location window* with the source code at the point of suspension, enter the `V` (**View**) command with no arguments. Note that this only works once the child process has been started.

The debugger always knows at any point in time where to continue execution. For example, you can stop execution to view a different source file, then continue where you left off.

Most debugger commands assume that the command applies to the current location and its scope. For example, if you stop in procedure `abc` and then view procedure `def` and ask for the value of a local variable that exists in both, the debugger returns the value of that variable as it exists in `def`.

---

**Note** The procedure `def` must be a caller of `abc`, or the variable must be statically declared, for its value to be meaningful.

---

4

The general format of most debugger commands is:

`command [ location ] [ command arguments ] [ command-list ]`

Numeric modifiers after commands can be any numeric expression. They need not be just simple numbers. A blank is required before any numeric option. Multiple commands on one line must be separated by “;”.

Here are some common modifiers and other special notations:

`{A | B | C}` Any one of A or B or C is required.

`[A | B | C]` Any one of A or B or C is optional.

*class* A C++ class name.

*command-list* A series of debugger commands, separated by “;” entered on the command line or associated with a breakpoint or assertion. Commands may be grouped with `{}` for the `a` (`assert`), `b` (`breakpoint`), `if`, `i` (`if`), and `!` commands. In all the other cases, commands inside `{}` are ignored.

*count* The number of repetitions specified for a command.

*depth* A stack depth as printed by the `t` (`trace`) command. The top procedure is at a depth of zero. A negative depth acts like a depth of zero or produces an appropriate error message.

In interpreting variable references where *depth* is not explicitly specified, the debugger will try to use the special variable `$depth` as the default value for the *depth*. If the required

## 4-2 HP Symbolic Debugger Commands

procedure (either explicitly specified or taken by default from the current viewing location) is at this *depth* on the stack, the debugger looks for the variable in that stack frame. If the required procedure is not the procedure at that stack *depth*, the debugger looks for the most recent instance of the required procedure by searching down from the top of the stack. If the procedure is found, the debugger looks for the variable in that stack frame.

(Series 600/700/800 only: see also the `tst` command for a discussion on PA-RISC `stubs`)

<i>expr</i>	Any expression, but with limitations stated in “Entering Expressions.”
<i>file</i>	A file name.
<i>format</i>	A style for printing data.
<i>label</i>	A program label.
<i>line</i>	A number that refers to a particular line in a file.
<i>proc</i>	A procedure (or function, or subroutine) name. <i>proc</i> may be of the form <i>proc@shared_library</i> in cases where it is necessary to uniquely qualify <i>proc</i> . See the section “Shared Library Symbols” in Chapter 6.
<i>shared-library</i>	The basename of a shared-library, without the trailing <code>.sl</code> (for example, <code>libc</code> ). Because of limitations in the debuggers command parser, only some non-alphanumeric characters are allowed in a <i>shared-library</i> basename; specifically, any one of the following characters: <code>.</code> , <code>,</code> , <code>:</code> , <code>-</code> , <code>~</code> , <code>%</code> , <code>^</code> , <code>=</code> , <code>+</code> . Use of any of these special characters requires parentheses to delimit a shared-library reference, distinguishing the special characters from operators:

*(symbol@shared-library)*

Parentheses must also be used if one of the above special characters is used *as an operator* on the symbol. In this case, the operator appears outside the delimiters; for example:

*(symbol@shared-library) .field*



Note that shared library basenames used in a debugger command are always case sensitive; that is, they are not affected by the `tc` (`toggle case`) command.

*location* A particular line in a file (and its corresponding address in the user's program if there is executable code for that line). A *location* has the following general forms:

*line*

# *label*

*file* [ :*line* ]

[ *file* : ] *proc* [ : *proc* [ ... ] ] [ :*line* | # *label* ]

[ *class* ] :: *proc* [ :*line* | # *label* ]

If a *location* involves an overloaded C++ function, the user will be presented with a menu to allow interactive selection of the intended routine. Note that the *proc:proc ...* form is used to specify a nested procedure in HP Pascal programs.

*address* An absolute code (text) or data location in a program's active address space that has the following forms:

*expr*

*label* [ +*expr* ]

*label* [ -*expr* ]

*proc*#*line*

[ [ *class* ] :: ] *proc*#*line*

The *expr* option must evaluate to an integer, and is assumed to be unsigned. The symbol *label* is found in the program's linker symbol table and is not a source label from the symbolic-debug (-g) tables. To reference labels within a non-debuggable shared library, use the syntax:

#### 4-4 HP Symbolic Debugger Commands

*label@shared\_library\_name*

Note that some Series 600/700/800 code labels begin with \$. To reference these labels, they must be prefixed with \ to distinguish them from special variables. For example, \cerror+0xc or \\$\$dyncall+0x28.

<i>number</i>	A constant number (e.g. “9”, not “4 + 5”). Floating point (real) numbers may be used any place a constant is allowed.
<i>var</i>	A variable name. See “Entering Variable Names” later in this chapter and “Shared Library Symbols” in Chapter 6.

4

## Using Uppercase and Lowercase

HP Symbolic Debugger commands are case-sensitive. The two cases are treated differently by the debugger. For example:

<b>s</b> or <b>step</b>	Lowercase <b>s</b> tells the HP Symbolic Debugger to single step to the next executable statement and step into a procedure, if necessary.
<b>S</b> or <b>Step</b>	Uppercase <b>S</b> tells the HP Symbolic Debugger to single step to the next executable statement treating a procedure call as a single statement (it is “stepped over”).

## Abbreviating Commands

You can enter commands in their complete spelled-out form (long form) or in an abbreviated form (short form). Generally, you can abbreviate one-word commands using the first character of the word. Abbreviate two-word commands using the first character of each word in the command (do not leave a space between the two characters). If you use the long form, you can leave a space between words. For example:

4       $\left. \begin{array}{l} \text{w} \\ \text{window} \end{array} \right\} \textit{number}$

Changes the size of the source window.

$\left. \begin{array}{l} \text{db} \\ \text{delete breakpoint} \end{array} \right\} [\textit{number}]$

Deletes the breakpoint selected by *number*.

Some debugger commands are not abbreviated by following the previous rules. Refer to the individual command syntax in this chapter to find abbreviations for these commands. Note that a few commands are available only in abbreviated form.

## Entering Variable Names

Variables are referenced exactly as they are named in your source file(s).

---

**Note**              Use of variable names in debugger commands is normally case insensitive; for example, **gvar** is the same variable as **GVAR**. This may be changed with the **tc** (**toggle case**) command.

---

There are several methods used to specify a variable depending on where and what it is. The following table shows the various forms for specifying a variable.

**Table 4-1. Methods for Specifying Variables**

Method	Description
<i>var</i>	Search the stack for the current or most recent (see the description of <i>depth</i> in the section “Entering Commands” in this chapter) instance of the current procedure (the procedure in the location and source windows). If found, see if <i>var</i> is a parameter or a local variable for that procedure. If the current procedure is a C++ member function, search for <i>var</i> as a member of the class. If no such local variable is found, the current language scoping rules are used to try to locate <i>var</i> . If <i>var</i> is not found in an enclosing block, procedure or module, the debugger searches for a global variable named <i>var</i> .
<i>class::var</i>	If the current procedure is a C++ member function of <i>class</i> or of a class derived from <i>class</i> , search for var in <i>class</i> or its base classes. Otherwise, search for <i>var</i> as a static member of <i>class</i> .
<i>proc:var</i> [[ <i>class</i> ]::] <i>proc</i> : [ <i>class</i> :: ] <i>var</i>	Search the stack for the current or most recent (see the description of <i>depth</i> in the section “Entering Commands” in this chapter) instance of <i>proc</i> . If found, see if it has a parameter or a local variable named <i>var</i> as before. The second syntax form allows a C++ procedure and/or variable to be qualified by a class. Preceding <i>proc</i> only with a :: indicates a global non-member function.

**Table 4-1. Methods for Specifying Variables (continued)**

Method	Description
<p><i>proc:depth:var</i></p> <p><code>[[ class ] :: ]proc:depth: [ class :: ] var</code></p>	<p>Use the instance of <i>proc</i> that is (exactly) at stack <i>depth</i> instead of the current or most recent (see the description of <i>depth</i> in the section “Entering Commands” in this chapter) instance. This is useful for debugging multiple instances of a recursive procedure. The second syntax form allows a C++ procedure and/or variable to be qualified by a class. Preceding <i>proc</i> only with a <code>::</code> indicates a global non-member function.</p>
<p><code>: var</code></p> <p><code>:: var</code></p>	<p>Search for a global (not local) variable named <i>var</i>.</p>
<p>.</p>	<p>Dot is a shorthand for the piece of data you last viewed. It has the same size it did when you last viewed it. Dot may be treated like any other variable.</p> <p>Note: Dot (“.”) is the name of a location. It is dereferenced like any other variable name. For example, if you want the address of something that is 30 bytes farther on in memory, do not type “+.30”. That would take the contents of dot and add 30 to it. Instead, type “&amp;+.30”, which adds 30 to the address of dot.</p>

## Special Variables

Special variables have names that are prefixed by a \$. Some special variables are predefined and have special meaning. Other special variables are user-defined variables to which you can assign values. Special variable names can be up to 98 characters long, but it is recommended that you limit their names to 80 characters long for display purposes. The first time you reference special variables, they are created and set to their initial values. Special variables can be used for the duration of the debugging session or you can redefine them.

4

For example, if you enter the following command (in HP FORTRAN 77 or HP C),

```
p $xyz = 3*4
```

the special variable \$xyz is created and assigned the value of 12.

To view special variables (except hardware registers), use the `ls` (`list specials`) command. There are several special variables that are available; all but user-defined special variables are predefined by the debugger. The special variables are:

- `$var`

Represents a user-defined variable. It is of type long integer and does not take on the type of any expressions assigned to it.

- Hardware Registers

A number of special variables exist to let you access the hardware registers. To find out which names are available on your system use the `lr` (`list registers`) command. See also the appendix “Registers Displayed by the HP Symbolic Debugger in Disassembly Mode.”

- `$result`

References the return value from the last procedure called from the command line. The special variable `$result` is normally interpreted to be the same type as the last procedure call (if the call returns a structured type, `$result` defaults to integer). Note that there are two alternate ways of looking at `$result`, as a 32 bit integer (`$long`) or as a 16 bit integer (`$short`).

- `$depth`

Contains the value of the current stack level (see the description of `depth` in the section “Entering Commands” in this chapter). This is the default stack level for viewing local variables. It is set by the `V`, `up`, `down`, and `top` commands. It may be adjusted by the `tst` command (Series 600/700/800 only). It is reset to 0 (top of stack) by the following commands: `r`, `R`, `c`, `C`, `s`, `S`, `g`, and `k`. Higher numbers correspond to procedures further down the stack (greater stack depth). Setting this variable directly (`p $depth = n`) sets the local context to the specified depth but it will not update the source window.

4

- **\$lang**

Allows you to view and modify the current source language designation for expression evaluations. Valid values for **\$lang** are **C++**, **FORTTRAN**, **Pascal**, **C**, and **default**. For example, if **\$lang** is set to **C**, the debugger expects HP C expression syntax, regardless of the language you are debugging.

When **\$lang** is set to **default**, any language expression syntax used is expected to be the same as the source language of the procedure currently being viewed.

- **\$line**

Displays the current source line number (usually the next statement to be executed).

- **\$malloc**

Allows you to see the amount of heap memory (in bytes) currently allocated by the debugger for its own use. This does not reflect memory-use of the program being debugged.

- **\$print**

Allow you to alter the behavior of the **print** command when printing character data. The allowed values are **ASCII**, **native** and **raw**. The default is **ASCII**. **ASCII** causes all non-ASCII characters to be displayed as octal-escapes (**\nnn**). **native** causes unprintable characters, as determined by the locale category (environment variable) **LC\_TYPE**, to be displayed as octal-escapes. **raw** causes all bytes to be output unaltered. This switch also affects the default display format for character types. The value of **LC\_TYPE** defaults to the environment variable **LANG**, and should correspond to the character set and keyboard language of the terminal or window being used.



- **\$signal**

Allows you to see and modify the pending current child process signal number. This is the signal that will be sent to the user program when control is returned to it via the **C (Continue)** command.

- **\$cplusplus**

This special variable is interpreted as a set of flags to control the behavior of certain C++ capabilities. If bit 0 (least significant bit) is not set, printing a class object with the **K** or **T** format will only print information for any given class once, regardless of how many times it appears in the object (unless the format is **K** and the base class is not virtual). If bit 0 is set, all base class information will be printed each time it occurs in the object.

If bit 1 is not set, the **bpc** command (**breakpoint class**) will set breakpoints only on member functions of the designated class and not of its base classes. With bit 1 set, breakpoints are also set on member functions of base classes.

If bit 2 is not set, the **bi** command (**breakpoint instance**), when a specific function is not given, will set breakpoints only on member functions of the class designated by the object and not of its base classes. With bit 2 set, breakpoints are also set on member functions of base classes.

The default behavior imposed by bits 1 and 2 may be temporarily overridden by the **-c** and **-C** options to the **bpc** (**breakpoint class**) and **bi** (**breakpoint instance**) commands.

■ **\$step**

Allows you to see and change the number of machine instructions the debugger steps through, while in a non-debuggable procedure, before setting an “uplevel” breakpoint and free-running to it. (This is where a breakpoint is set immediately after the return location in the non-debuggable procedure’s caller.) This situation occurs only when the program is executing in single-step or assertion mode, and represents the debugger’s attempt to step “into” becoming a step “over.” The default value for **\$step** is 12 on Series 300/400 computers and 24 on Series 600/700/800 computers.

The actions the debugger actually performs during a single step from one source statement to the next is to execute a single machine level instruction at a time. After each instruction is executed, the debugger checks to see if the next instruction matches the beginning of a new source line. If so, execution stops, the debugger prompts for user input, and the single step is complete. However, if this low-level stepping proceeds through a procedure call, it may or may not be entering a non-debuggable procedure. Until the debugger encounters an instruction corresponding to a source line, it presumes it is in non-debuggable code. If, after **\$step** instructions, the debugger fails to find a source line, it sets an internal uplevel breakpoint at the instruction after the procedure call, free-runs to it, resets **\$step**, and then continues its search for a source line from there.

If **\$step** is set to too small a value, the debugger may erroneously fail to step into a debuggable procedure. If **\$step** is too large, it will degrade performance when stepping over non-debuggable procedures. Increasing the value of **\$step** from its default value is usually only necessary if the program being debugged regularly makes calls to debuggable procedures by indirectly calling them through (short) user-coded non-debuggable interface routines.

(Series 600/700/800 only) The HP-PA procedure calling conventions frequently require insertion of **stubs** (calling interludes) in the calling sequence. This is especially true for shared-library calls and calls that pass floating-point parameters. These **stubs** are non-debuggable, and single-steps through them (at the source level) are subject to the effects of **\$step**. It may occasionally be necessary to increase the value of **\$step** if you find that a procedure call cannot be stepped into when it should be possible to do so.

- `$fpa`

If this is set to a non-zero value, any sequence of machine instructions that constitute a single floating-point accelerator instruction is treated as a single instruction for machine-level single-stepping and display. This special variable is for Series 300 computers only.

- `$fpa_reg`

If `$fpa` is set to a non-zero value, `$fpa_reg` indicates the address register used in floating point accelerator instruction sequences. A 0 corresponds to register a0, 1 to a1, etc. The default value is 2. This special variable is for Series 300 computers only.

## Entering Expressions

An expression is a symbolic or mathematical representation. Expressions consist of variables, constants and operators, or any syntactically correct combination of these items. The HP Symbolic Debugger evaluates user expressions as if they are part of the high-level language being debugged and, therefore, uses the same operators and assignment rules as the high-level language.

See Appendices B, C, D, and E for a list of operators that you can use with each language. Note that the symbolic debugger tries as much as possible to let you write expressions with the same syntax as the current language. You can change the current language by setting the value of the special variable `$lang`. By default, this variable is set to the language of the program you are debugging.

The `$in` operator, a special unary operator, evaluates to true (1) if the operand is a debuggable procedure and if `$pc` (the current child process program counter or location) is in that procedure; otherwise, `$in` is false (0). For example, `$in load_month_table` is true if the child process is currently suspended in `load_month_table`.

The unary operator `$addr`, for retrieving the address of a variable, and `$sizeof`, another unary operator for retrieving the byte size of a variable, are available for all languages.

Constant expressions may be textual (character or string), symbolic (that is, predefined language-specific keywords), or numeric.

If you do not have an active child process or valid core file, you can only evaluate expressions containing constants.

## Character and String Expressions

The rules in each language for entering character and string constants are as follows:

- For HP FORTRAN 77 and HP Pascal, string constants are represented by one or more characters, enclosed by single quotation marks ( `'` ) or double quotation marks ( `"` ).
- For HP C and HP C++, single quotation marks enclose single characters for character constants. Double quotation marks enclose zero or more characters for string constants. String constants are treated as pointers to `char` (i.e., `char *`).

The prefix `L` can be used to denote wide-character or string constants (C type `wchar_t`). Use of this prefix will cause the value to be mapped to its wide-character equivalent before being stored (see *multibyte(3C)* in the *HP-UX Reference*). If an unmappable value is encountered, it is stored unconverted. Note that `wchar_t` is a predefined ANSI C type (see *stdlib(3C)* in the *HP-UX Reference*).

String constants are stored in a buffer in the `/usr/lib/end.o` file which must be linked with your program. This is done automatically by the compiler when the `-g` option is given.

---

**Note**            If you call the linker directly, don't forget to specify `/usr/lib/end.o` at the end of the list of object files you want to link and before any other library.

---

The debugger starts storing strings at the beginning of this buffer, and moves along as more assignments are made. If the debugger reaches the end of the buffer, it goes back and reuses it from the beginning. This does not usually cause any problems. However, if you use very long strings, or if you assign a string constant to a global pointer, problems could arise.

- Character and string constants can contain standard backslashed escapes (as understood by the HP C compiler), including those shown in the table "Escape Sequences." For hex-escapes, the longest possible value is evaluated and then truncated to the size of the destination type (either 1 or 4 bytes). A `\<newline>` is not supported in quotes or at the end of a command line.

**Table 4-2. Escape Sequences**

Character	Description
bell	\a
backspace	\b
form feed	\f
carriage return	\r
horizontal tab	\t
vertical tab	\v
backslash	\\
single quote	\'
double quote	\"
bit pattern	\ <i>nnn</i> (octal digits) or \x <i>nnn</i> (hex digits)
new line	\n

4

### **Symbolic Constants**

Expressions can also contain the symbolic constants listed in the table “Symbolic Constants.”

**Table 4-3. Symbolic Constants**

Language	Constants
HP Pascal	nil maxint minint true false
HP FORTRAN 77	.TRUE. .FALSE.
HP C	None
HP C++	None

### Numeric Constants

Integer constants can begin with 0 for octal, 0x or 0X for hexadecimal, or 0b or 0B for binary. If followed immediately by l or L, they are forced to be of type long. Likewise, u and U force the type to be unsigned. The suffix ul or UL corresponds to **unsigned long**. If no suffix is used, the smallest type in which the value will fit is used.

Floating-point constants must be of the form:

$$\text{digits} \cdot \text{digits} \left[ \begin{array}{c} \text{e} \\ \text{E} \\ \text{d} \\ \text{D} \left[ \begin{array}{c} + \\ - \end{array} \right] \text{digits} \\ \text{l} \\ \text{L} \end{array} \right] \left[ \begin{array}{c} \text{F} \\ \text{f} \\ \text{L} \\ \text{l} \end{array} \right]$$

For example, any of the following is in the correct form:

1.0

5.9L

3.14e8

26.62D-31

The suffixes **f** and **F** cause the value to be evaluated as type **float** (4-byte IEEE real). The suffixes **l** and **L** cause the value to be evaluated as type **long double** (16-byte IEEE real). Unless a direct assignment is made, **float** and **long double** types are converted to type **double** before the expression is evaluated.

4

One or more leading digits is required to avoid confusion with **.** (**dot**). A decimal point and one or more following digits is required to avoid confusion for some command formats. If the exponent does not exactly fit the pattern shown, it is not taken as part of the number, but as separate tokens. The **d** and **D** exponent forms are allowed for compatibility with HP FORTRAN 77. The **l** and **L** exponents forms are allowed for compatibility with HP Pascal.

In the absence of a suffix character, the constant is assumed to be of type **double** (8-byte IEEE real).

### Promotion of Operands

Expressions approximately follow the HP C language rules of promotion. In other words, **char**, **short**, and **int** become **long**, and **float** becomes **double**. If either operand is a **double**, floating math is used. If either operand is **unsigned**, unsigned math is used. Otherwise, normal (integer) math is used. Results are then cast to proper destination types for assignments.

If a floating point number is used with an operator that does not normally permit it, the number is cast to **long** and used that way. For example, the HP C binary number **~** (bit invert) applied to the constant 3.14159 is the same as **~3**.

### Assignment

Note that **=** means *assign* in all languages but HP Pascal; to test for equality, use **.EQ.** for HP FORTRAN 77 and **==** for HP C and HP C++.



In HP Pascal, = is a comparison operator; use := for assignments. For example, suppose you invoke the debugger, then set \$lang to Pascal:

```
p $lang = Pascal
```

To return to HP C, you must use the := operator:

```
p $lang := C
```

## Pointers, Casts, and Composite Types

4

You can dereference any constant, variable, or expression result using the HP C \* operator. If the address is invalid, an error is given.

Type casting is allowed. For simple types, the syntax is identical to HP C. For example:

```
(short) size  
(double *) mass_ptr
```

These casts are limited to `char`, `short`, `long`, `int`, `unsigned`, `float`, `double`, approximate combinations of these keywords, and single level pointer types. Also supported are `class`, `struct` and `union` pointer type dereferences. For example:

```
bat_ptr = &bat  
(struct fob) &bat  
(struct fob) bat_ptr
```

Both of these casts treat `bat` as a struct of type `fob` during printing. Class, structure, and union pointer casts can only include the keyword `class`, `struct` or `union`, an appropriate tag, and an optional “\*.” The argument of the cast is simply treated as an address.

## Arrays

Whenever an array variable is referenced without giving all its subscripts, the result is the address of the lowest element referenced. For example consider the following declared arrays:

```
HP FORTRAN 77    x(5,6,7)  
HP Pascal        x[1..5,2..6,3..7]  
HP C              x[5][6][7]
```

Referencing it simply as `x` is the same as the following:

HP FORTRAN 77      `x(1,1,1)`

HP Pascal            `x[1,2,3]`

HP C                 `x`

If a not-fully-qualified array reference appears on the left side of an assignment, the value of the right-hand expression is stored into the element at the address specified.

## Entering Procedure Calls in an Expression

You can include calls to procedures in expressions. You can call any executable procedure from the command line whether or not it was compiled with debugger information. You can use the `lp` (**list procedures**) command to list the debuggable procedures in the program.

Member functions of C++ classes may be called using C++ syntax:

`[ [ class ] :: ] proc(parameter list)`

`expr. [ class :: ] proc(parameter list)`

`expr-> [ class :: ] proc(parameter list)`

C++ overloaded operators can only be specified by function names. For example, if `+` is overloaded, `a = b.operator+(c)` is allowed but not the corresponding `a = b + c`.

The following command evaluates an expression that calls the procedure `ref` and uses its return value:

```
p $xyz = $abc*(3 + ref (ghi - 1, jkl, "Hi Folks"))
```

An argument list must follow each procedure call, even if it is empty (for example, `proc()`). When a procedure is called, the following considerations apply:

- The HP Symbolic Debugger has one active command line at a time. During a command line procedure call, any breakpoints reached during execution are treated as usual (by suspending execution as specified). If execution is stopped in a called procedure, the remainder of the original command line is discarded and you are informed of this.
- 4 ■ If you try to call a procedure when the child process is not active, then a child process is started by the debugger. This will invalidate the contents of the core file if one was specified on debugger invocation. This process is similar to using the single-step command to initially activate a child process.

## Window Mode Commands

Window mode commands let you control what is displayed on the screen. The window mode commands are:

- `td` (toggle disassembly)
- `fr` (floating point registers)
- `tf` (toggle float)
- `gr` (general registers)
- `+r`
- `-r`
- `sr` (special registers)
- `ts` (toggle screen)
- `u` (update)
- `U` (Update)
- `w` (window)

4

The source window displays source lines in a program. In disassembly mode, the top five lines of the screen show the floating point, general or special registers (the register window) followed by assembly language instructions (the assembly window). In split-screen mode, the top part of the screen displays source code followed by the corresponding assembly language instructions.

### **td (toggle disassembly)**

$$\left\{ \begin{array}{l} \text{td} \\ \text{toggle disassembly} \end{array} \right\}$$

Toggles the source window between disassembly mode and source mode. When in disassembly mode, this command displays the assembly language instructions, corresponding to the source code, below one of the sets of registers (floating point, general, or special (for Series 600/700/800)).

When in disassembly mode, the single step command steps one machine instruction at a time (rather than one source statement at a time).

The assembly language display of each instruction consists of: the source line number, the address in hexadecimal, the address in the form of the nearest label plus the offset from the label, and the actual assembly instruction mnemonic and operands.

## Window Mode Commands

### fr (floating point registers)

{ fr  
floating point registers }

Display the floating point registers in the register window when the debugger is in assembly (non-split-screen) mode.

4 On a Series 300 computer having multiple floating point co-processors, you will be asked which set of registers you want to display. The floating-point register sets supported are those for the MC68881/MC68882, HP 98635, and HP 98248 floating-point co-processors. When the value of a register is changed by a command (usually a **step**), that register is highlighted until after the next command.

On Series 600/700/800 computers, these registers may be examined or modified by using the debugger special variables **\$f0** through **\$f15** (**\$f0** through **\$f31** on PA-RISC 1.1 computers). Floating-point registers **\$f0** through **\$f3** are read-only registers.

### tf (toggle float, Series 600/700/800 computers only)

{ tf  
toggle float }

Toggle the display of the numeric floating point registers in the register window from single-precision to double-precision or vice-versa. The current mode is displayed in the line dividing the register window from the disassembly window as **SGL** or **DBL**. Numeric floating point registers are registers **f4** through **f15** on PA-RISC 1.0 implementations, and **f4** through **f31** on PA-RISC 1.1 implementations.

In double-precision mode, each numeric floating point register is interpreted as an 8-byte floating point quantity, and is simultaneously displayed in both hexadecimal and double-precision decimal formats.

In single-precision mode, the left (high-order) half of each floating point register is interpreted as a 4-byte floating point quantity, and is simultaneously displayed as both hexadecimal and single-precision decimal formats. For PA-RISC 1.1 implementations, the right half of each floating point register is also displayed in single-precision format.

**gr (general registers)**

$$\left\{ \begin{array}{l} \text{gr} \\ \text{general registers} \end{array} \right\}$$

Display the general registers in the register window when the debugger is in assembly (non-split-screen) mode. When the value of a register changes, that register is highlighted until after the next command. General registers may be modified by using debugger special variables (see the appendix “Special Variables Used by the Symbolic Debugger”). When displaying the general registers or the floating point registers, the line dividing the registers from the assembly code also displays certain special processor registers. Some registers are displayed as a string of letters, each letter representing a bit in the register. A lowercase letter indicates that the corresponding bit is off, uppercase means on.

4

**+r and -r**

$$\left\{ \begin{array}{l} +r \\ -r \end{array} \right\}$$

The **+r** command scrolls the floating-point register display forward four registers. The **-r** command scrolls the floating-point register display backward four registers. Note that you can press **(Return)** to repeat this command.

**sr (special registers, Series 600/700/800 computers only)**

$$\left\{ \begin{array}{l} \text{sr} \\ \text{special registers} \end{array} \right\}$$

Display the special registers (Series 600/700/800 space and control registers) in the register window when the debugger is in assembly (non-split-screen) mode. When the value of a register changes, it is highlighted until after the next command. The control registers cannot be modified.

## Window Mode Commands

### ts (toggle screen)

$$\left\{ \begin{array}{l} \text{ts} \\ \text{toggle screen} \end{array} \right\}$$

Toggles the source window between all source or assembly and split-screen mode. In split-screen mode, the source window displays both source code and corresponding assembly instructions. Single stepping occurs at either the source statement or the assembly instruction level, depending on the part of the split-screen in which you are single stepping. The stepping mode is displayed in the line separating the source and assembly windows, and is toggled with the **td** command.

4

### u (update)

$$\left\{ \begin{array}{l} \text{u} \\ \text{update} \end{array} \right\}$$

Updates the source and location windows to show the location where the user program is suspended. This command is useful in an assertion. For example, this command:

$$\text{a } \{ \text{u} \}$$

will continuously update the screen to follow the execution of the program as it proceeds.

### U (Update)

$$\left\{ \begin{array}{l} \text{U} \\ \text{Update} \end{array} \right\}$$

Clears the screen of all data and redraws the screen. Use this command if the screen gets corrupted by a system-wide announcement that overwrites your session.

### w (window)

$$\left\{ \begin{array}{l} w \\ \text{window} \end{array} \right\} \textit{size}$$

If your terminal supports windowing (window mode), this command changes the height of the source window to the number of lines that you specify. By default the source window will occupy about two thirds of the total window or terminal size. Changing the size of the source window also changes the size of the command window.

4

If your terminal does not support windowing or you have forced line mode by including the `-L` option when invoking the debugger, this command prints the specified number of lines surrounding the current line. If no number is specified, the last number used with the `w (window)` command is used again. You can press `Return` to repeat this command. The next specified number of lines will be displayed.



---

## File Viewing Commands

The file viewing commands let you view program source code. The file viewing commands are:

- +
- -
- v (view)
- /
- ?
- n (next)
- N (Next)
- D (Directory)
- ld (list directories)
- lf (list files)
- L (Location)
- va (view address)

4

+

+ [ *number* ]

Moves the viewing location forward in the current file the specified *number* of lines (or the specified number of instructions in disassembly mode). If you do not enter a number, the next line (or instruction) becomes the current line (or instruction).

You can press a **Return** to repeat this command. If your terminal supports windowing, a new group of lines are displayed. If it does not support windowing, or you have forced line mode by including the `-L` option when invoking the debugger, only the new current line and its line number are displayed.

4

-

- [ *number* ]

Moves the specified *number* of lines (or the specified number of instructions in disassembly mode) backward in the current file and updates the windows. The default is one line (or instruction) before the current line (or instruction).

You can press **Return** to repeat this command. If your terminal supports windowing, a new group of lines (or instructions) are displayed. If it does not support windowing, or you have forced line mode by including the `-L` option when invoking the debugger, only the new current line and its line number are displayed.

## File Viewing Commands

### v (view)

$$\left\{ \begin{array}{l} \mathbf{v} \\ \mathbf{view} \end{array} \right\} [location]$$

Displays one source window forward from the current source window if no *location* is given. One line from the previous window is preserved for context. If your terminal does not support windowing, or if you have forced line mode by including the -L option when invoking the debugger, only the new source line is displayed.

A *location* can be a particular line, procedure, or any text file, whether used in the program or not. Using the location modifier causes the specified location to become the current location, and the source at the specified location is then displayed in the source window. The source location window is adjusted accordingly.

If a procedure (*proc*) name is specified for the location, the procedure's first executable line becomes the current line.

You can press **Return** to repeat this command. If a location was given, subsequent **Return**'s move forward from that point.

---

### Note

In order for the debugger to associate a source file with the corresponding code when the *location* includes a *filename*, the file name must be a *basename* only. By default, the debugger uses the same path names for finding source files as were used during compilation. These path names may be either relative or absolute. If the source files are not accessible via the compile-time path names, use the commands described in the section "Source File Mapping Commands" found in this chapter to provide correct paths. Alternately, the -d option (when invoking the debugger) or the D (**Directory**) command (see D under "File Viewing Commands") can be used.

Any text file can be examined with the **view** command, and the *filename* given can be an absolute or relative path name, but the debugger will not recognize these files as source for the program.

---

/

*/[string]*

Searches forward in the file for the specified *string*. Searches wrap around the end of the file. If you do not enter a string, the last search string you entered is used again. The string must be literal; wild cards are not supported.

You can select case sensitivity for string searches with the **tc** (**toggle case**) command. Initially, searches are case insensitive.

4

?

*?[string]*

Searches backward in the current file for the specified *string*. Searches wrap around the beginning of the file. If you do not enter a string, the last search string you entered is used again. The string must be literal; wild cards are not supported.

You can select case sensitivity for string searches with the **tc** (**toggle case**) command. Initially, searches are case insensitive.

**n (next)**

$$\left\{ \begin{array}{l} \text{n} \\ \text{next} \end{array} \right\}$$

Repeats the previous search (*/* or *?*) command.

## File Viewing Commands

### N (Next)

$$\left\{ \begin{array}{l} \text{N} \\ \text{Next} \end{array} \right\}$$

Repeats the previous search (/ or ?) command, searching in the opposite direction.

### D (Directory)

$$\left\{ \begin{array}{l} \text{D} \\ \text{Directory} \end{array} \right\} "dir"$$

Adds the directory that you specify to the end of the list of directory search paths for source files. You can add more than one directory, but only one can be added at a time. Directories are searched in the order that they are added. When searching for the source *file* in an alternate directory *altdir* which has been specified by the D (Directory) command (where *file* itself is made up of a directory and base name: *dirname/basename*), the debugger first attempts to open *altdir/dirname/basename*. If this fails, the debugger then attempts to open *altdir/basename*.

The D command is equivalent to the command-line option `-d`.

For more information on source file mapping, read the section “Source File Mapping Commands” found in this chapter.

**ld (list directories)**

```
{ ld
  list directories }
```

Lists all the alternate directories that are searched when the debugger tries to locate the source files. The list order is the same as the search order.

**lf (list files)**

```
{ lf
  list files } [ string ] [ @shared-library ]
```

4

Lists all source files containing executable statements that were compiled (with the `-g` option) to build the program (`a.out`). Code address ranges are shown for each file. Only files containing debuggable executable code are shown. If a *string* is specified, only those files beginning with this string are listed.

This command also lists any include files containing executable code with their code addresses. A file may appear several times if it contains include files. An example of the output is:

```
0: /usr/project/src/tree1.c      0x00001834 to 0x00002524
1: /usr/global/src/treeglobs.c  0x00002530 to 0x00003210
2: /usr/project/src/tree2.c      0x00003344 to 0x00004002
```

Files which belong to shared libraries which are currently not active (not mapped into the process) are shown as:

```
3: /usr/project/lib/libxyz/mod.c      (not mapped)
```

## File Viewing Commands

### L (Location)

$$\left\{ \begin{array}{l} \text{L} \\ \text{Location} \end{array} \right\}$$

Displays in the command window the current file, procedure, line number and the source text for the current viewing location. When used in a breakpoint or assertion command list, the current point of execution is displayed. This command allows you to determine where you are in the program and is useful when included in an assertion or breakpoint command list. For example:

4

```
>L
doproc.c: eval_q: 8: if (qp != NULL) {
```

You cannot press **Return** to repeat this command.

### va (view address)

$$\left\{ \begin{array}{l} \text{va} \\ \text{view address} \end{array} \right\} \textit{address}$$

Displays in the disassembly window the assembly code at the specified *address*, which can be an absolute address or symbolic code label with an optional offset (for example, `_start + 0x20`). Symbolic addresses within shared libraries (see the section “Shared Libraries Symbols” found in Chapter 6) can be referenced using the syntax *label@shared\_library\_name* (for example, `_printf@libc`). This command is used in disassembly mode only.

Note that if the specified *address* is not a valid code address, the assembly code at the address closest to the given address will be displayed instead.

(Series 600/700/800 only) For programs that are linked with shared libraries, and for shared libraries themselves, a code label may appear twice in the linker symbol table. In such cases, the `ll` (**list labels**) command will display both a **Code/Univ** (actual entry point) and a **Entry/Univ** (stub) symbol with the same name, but differing addresses. A symbol used in the address provided to the `va` command will always be associated with the actual entry point by that name, rather than the stub. To view a named stub, the `list labels` command must first be used to locate the actual address of the stub, and that numeric address provided to the `va` command.

## Source Directory Mapping Commands

The complete path names of source files listed on a compiler command line are stored without change in the symbolic-debug information for the resulting program. The debugger will attempt to locate the sources using that entire path. The path-map facility provides an alternate method for locating these source files. The source file mapping commands are:

- `apm` (add path map)
- `lpm` (list path map)
- `dpm` (delete path map)

4

If all maps are exhausted and the file has not yet been located, the alternate directories (as specified with `D` or `-d`) are then used (as in previous releases of the debugger).

### `apm` (add path map)

$$\left\{ \begin{array}{l} \text{apm} \\ \text{add path map} \end{array} \right\} \left\{ \begin{array}{l} \text{old\_path} \\ "" \end{array} \right\} [ \text{new\_path} ]$$

Allows you to modify the path the debugger will use to locate a set of source files (see below).

### `lpm` (list path map)

$$\left\{ \begin{array}{l} \text{lpm} \\ \text{list path map} \end{array} \right\}$$

Lists the path maps in the order in which they will be searched.

### `dpm` (remove path map)

$$\left\{ \begin{array}{l} \text{dpm} \\ \text{delete path map} \end{array} \right\} \left[ \begin{array}{l} n \\ * \end{array} \right]$$

Removes the latest path map entered if used with no arguments. If a positive integer  $n$  is given, the  $n$ th path map will be removed. If a  $*$  is given, all the path maps will be removed.



---

## Data Viewing and Modification Commands

Data viewing and modification commands allow you to view program data in a variety of formats and change the values of variables. The data viewing and modification commands are:

- `l` (list)
- `lc` (list common)
- `lcl` (list classes)
- `lct` (list class templates)
- `lft` (list function templates)
- `lg` (list globals)
- `ll` (list labels)
- `lm` (list macros)
- `lo` (list overload)
- `lp` (list procedures)
- `lr` (list registers)
- `ls` (list specials)
- `lsl` (list shared libraries)
- `ltf` (list template functions)
- `lx` (list exceptions)
- `mm` (memory map)
- `p` (print)
- `pq` (print quiet)

### **l** (list)

$$\left\{ \begin{array}{l} 1 \\ \text{list} \end{array} \right\} \left\{ \left\{ [proc[:depth]] \right\} \left\{ [class]::[proc[:depth]] \right\} \right\}$$

Lists all parameters and local variables for the current procedure. You can optionally specify any active procedure and its *depth* on the stack. In interpreting variable references where *depth* is not explicitly specified, the debugger will try to use the special variable `$depth` as the default value for the *depth*. If the required procedure (either explicitly specified or taken by default from the current viewing location) is at this *depth* on the stack, the debugger looks for the variable in that stack frame. If the required procedure is not the procedure at that stack *depth*, the debugger looks for the most recent instance of the required procedure by searching down from the top of the stack. If the

## Data Viewing and Modification Commands

procedure is found, the debugger looks for the variable in that stack frame. If an invocation of the procedure other than the default is desired, then a depth must be specified. The following illustrates the use of this command.

If the current stack trace (generated with the command `t 5`) is:

```
0 groucho( )      [marx.c:23]
1 harpo( )       [marx.c:70]
2 chico( )       [marx.c:55]
3 harpo( )       [marx.c:73]
4 main( )        [marx.c:16]
```

4

and `groucho` is the procedure currently viewed (and where execution is currently suspended), then:

- 1 Lists the local variables and parameters of `groucho`.
- 1 `harpo` Lists the local variables and parameters of `harpo` at level 1 on the stack.
- 1 `harpo:3` Lists the local variables and parameters of `harpo` at level 3 on the stack. Alternately executing:
  - >V 3
  - >1will also list the local variables and parameter of `harpo` at level 3 on the stack.
- V 2 Makes `chico` the current procedure and 2 the current stack depth.
- 1 `harpo` Lists parameters and variables for `harpo` at level 1 on the stack.

The `\n` (normal) format is used to display the procedures, parameters, and local data except for arrays and pointers, which are displayed as addresses.

The second form of the argument to this command allows a class to be specified in qualifying a C++ function.

## Data Viewing and Modification Commands

### lc (list common)

$$\left\{ \begin{array}{l} \text{lc} \\ \text{list common} \end{array} \right\} [ \textit{string} ]$$

Used when debugging an HP FORTRAN 77 program, this command displays HP FORTRAN 77 common blocks and their associated variables (this command is only supported on Series 600/700/800 computers). If a *string* is specified, only common blocks whose names begin with that string are printed; otherwise, all common blocks visible from within the current subroutine or function are printed.

4

Sample output is:

```
>lc
COMMON  /COM1/
BR4      = 0
INT1     = 0
BR8      = 0
BI4      = -2097152000
BI2      = -32000
BCX8     = 0
BC1      = '\000'
BL4      = .FALSE.
```

**lcl (list classes)**

$$\left\{ \begin{array}{l} \text{lcl} \\ \text{list classes} \end{array} \right\} [ \textit{string} ] [ @\textit{shared-library} ]$$

Lists all classes (regular classes and templates) known to the debugger. The optional *string* causes only classes whose names start with that *string* to be listed.

If debuggable shared-libraries are present, the **lcl** (**list classes**) command will show (matching) classes within each library. If only *@shared-library* is provided, all classes within the named library are shown.

4

**lct (list class templates)**

$$\left\{ \begin{array}{l} \text{lct} \\ \text{list class templates} \end{array} \right\} [ \textit{string} ] [ @\textit{shared-library} ]$$

Lists all class templates known to the debugger. The optional *string* causes only templates whose names start with that *string* to be listed.

If debuggable shared-libraries are present, the **lct** (**list class templates**) command will show (matching) class templates within each library. If only *@shared-library* is provided, all class templates within the named library are shown.

**lft (list function templates)**

$$\left\{ \begin{array}{l} \text{lft} \\ \text{list function templates} \end{array} \right\} [ \textit{string} ] [ @\textit{shared-library} ]$$

Lists all function templates known to the debugger. The optional *string* causes only templates whose names start with that *string* to be listed.

If debuggable shared libraries are present, the **lft** (**list function templates**) command will show (matching) function templates within each library. If only *@shared-library* is provided, all function templates within the named library are shown.

## Data Viewing and Modification Commands

### lg (list globals)

$$\left\{ \begin{array}{l} \text{lg} \\ \text{list globals} \end{array} \right\} [ \textit{string} ] [ @\textit{shared-library} ]$$

Lists all global variables and their values. If a *string* is specified, only global variables whose names begin with this string are listed.

If debuggable shared libraries are present, the `lg (list globals)` command will show (matching) globals within each library. If only `@shared-library` is provided, all globals within the named library are shown.

4

### ll (list labels)

$$\left\{ \begin{array}{l} \text{ll} \\ \text{list labels} \end{array} \right\} [ \textit{string} ] [ @\textit{shared-library} ]$$

Lists all external labels and program entry points known to the linker, as well as their type (i.e., code and data). If shared libraries are present, the name of the library containing the symbol is also shown. If a *string* is specified, only symbolic addresses with this prefix are used. If *string* ends in `@shared_library`, only those labels within *shared\_library* are shown. For example, executing:

```
ll @libm
```

shows only and all of those symbols in `libm.sl`.

(Series 300/400 only) In a program linked with shared libraries, many code symbols will appear more than once with the same symbol types. For example, in a program that calls `printf(3C)`, the symbol `printf` will appear at least twice:

- once in the program and each library that calls `printf`, where the symbol denotes the location of the PLT (Procedure Linkage Table) entry for `printf`

<code>_printf</code>	<code>0x000010a8</code>	Code (2ary)	<code>a.out</code>
----------------------	-------------------------	-------------	--------------------

- once in the library that defines the symbol.

<code>_printf</code>	<code>0x80057678</code>	Code (2ary)	<code>libc</code>
----------------------	-------------------------	-------------	-------------------

(Series 600/700/800 only) In programs linked with shared libraries, many code symbols will appear more than once, with the same or different symbol types.

## 4-40 HP Symbolic Debugger Commands

## Data Viewing and Modification Commands

For example, in a program that calls *printf(3C)*, the symbol `printf` may appear as 3 different symbol-types:

<code>printf</code>	<code>0x00001c38</code>	<code>Stub/Extern</code>	<code>a.out</code>
<code>printf</code>	<code>0x800958cc</code>	<code>Entry-stub/Univ</code>	<code>libc</code>
<code>printf</code>	<code>0x80095904</code>	<code>Code/Univ</code>	<code>libc</code>

where the symbol types denote the following:

<code>Stub/Extern</code>	shared-library import stub
<code>Entry-stub/Univ</code>	shared-library export stub
<code>Code/Univ</code>	actual entry point

4

An import stub will appear once in each shared library that calls `printf`, as well as in the program itself. The export stub and entry point will each appear once in the library that defines the procedure (for example, `libc`).

Referencing a code label in a *ba* (breakpoint address) or *va* (view address) will always default to the actual entry point (`Code/Univ`) by that name, if it exists.

### **lm (list macros)**

$$\left\{ \begin{array}{l} \text{lm} \\ \text{list macros} \end{array} \right\} [ \textit{string} ]$$

Displays all user-defined macros and their definitions. If a *string* is specified, only macros whose names begin with this string are listed.

Sample output is:

```
>lm
pheadtuti ==> p flavor:list->head.tuttifrutti
unS ==> bu\t {}; c
Overall macros state: ACTIVE
```

## Data Viewing and Modification Commands

### lo (list overload)

$$\left\{ \begin{array}{l} \text{lo} \\ \text{list overload} \end{array} \right\} \left[ [ \text{class} ] : : [ \text{string} ] [ @\text{shared-library} ] \right]$$

List overloaded C++ functions. If *string* is specified, only functions with the same initial characters are listed. The search may also be qualified by a class name.

4

If debuggable shared libraries are present, the `lo (list overload)` command will show (matching) overloaded functions within each library. If only `@shared-library` is provided, all overloaded function within the named library are shown.

### lp (list procedures)

$$\left\{ \begin{array}{l} \text{lp} \\ \text{list procedures} \end{array} \right\} [ \text{string} ] [ @\text{shared-library} ]$$

Lists all procedure names and their aliases as well as their starting and ending addresses, and their starting and ending source line numbers. If a string is specified, only procedures whose names begin with this string are listed. For C++, the list may be restricted to particular class member functions with:

```
lp [[class] : : [string]]
```

Sample output is:

```
0:  main          0x00000868 to 0x0000089c  [ c.c: 5 - 7]
0:  _MAIN_
1:  proc1         0x000008a4 to 0x000008b4  [ c.c: 11 - 12]
2:  proc2         0x000008bc to 0x000008cc  [ c.c: 16 - 17]
3:  _end_         0x000019b8 to 0x000019cc  [ end.c: 95 - 96]
```

---

**Note** The procedure name `_MAIN_` is used as the alias name for the main program in all supported languages. Do not use it for any debuggable procedures.

---

If debuggable shared libraries are present, the `lp` (`list procedures`) command will show (matching) procedures within each library. If only `@shared-library` is provided, all procedures within the named library are shown. Procedures that are in shared libraries that are currently not active (not mapped into the process) are shown as:

```
4:  shlib_proc      UNMAPPED      [ libxyz.c: 104 - 142 ]
```

4

### **lr (list registers)**

$$\left\{ \begin{array}{l} \text{lr} \\ \text{list registers} \end{array} \right\} [ \textit{string} ]$$

Lists all hardware registers and their contents. This command displays all general and floating point registers, as well as the program counter, stack pointer registers, and other registers. If a *string* is specified, only registers beginning with this string are listed (the `$` is significant). All registers except Series 600/700/800 floating-point registers are printed in hexadecimal. A list of registers available on the supported architectures can be found in the appendix “Registers Displayed by the HP Symbolic Debugger in Disassembly Mode.”



## Data Viewing and Modification Commands

### ls (list specials)

$$\left\{ \begin{array}{l} \text{ls} \\ \text{list specials} \end{array} \right\} [string]$$

Lists all special variables and their values. Registers are not listed. If a *string* is specified, only those special variables whose names begin with this string are listed (the \$ is significant).

Sample output is:

4

```
$lang      = FORTRAN
$depth     = 0
$line      = 49
$signal    = 0
$malloc    = 43008
$print     = ascii
$cplusplus = 0
$step      = 100
$long      = 0
$short     = 0
$result    = 0
```

You can also list special variables defined by usage. For example:

```
p $var = 10
```

defines the variable `$var` to be equal to 10. The `ls (list specials)` command will also display `$var` and its current value.

**lsl (list shared libraries)**

```
{ lsl
  list shared libraries }
```

Lists all shared libraries known to the debugger, including those found as dependents to the program or other shared libraries, as well as those explicitly given to the debugger by way of the `-l` invocation option.

For each library, the command results indicate whether the library is active (currently mapped into the process), and whether symbolic debug information is available and/or loaded into the debugger.

4

Sample output is:

Name	Mapped	SymDebug	Path
myprog	Yes	Yes	myprog
libdld	Yes	No	/usr/lib/libdld.sl
lib1	Yes	Not loaded	./lib1.sl
libc	Yes	No	/lib/libc.sl
lib2	Yes	Not loaded	lib2.sl
lib3	No	Not loaded	lib3.sl

**ltf (list template functions)**

```
{ ltf
  list template functions } [string] [@shared-library]
```

Lists all template functions known to the debugger. The optional *string* causes only template functions whose names start with that *string* to be listed.

If debuggable shared libraries are present, the **ltf** (**list template functions**) command will show (matching) function templates within each library. If only *@shared-library* is provided, all function templates within the named library are shown.

## Data Viewing and Modification Commands

### lx (list exceptions)

$$\left\{ \begin{array}{l} \text{lx} \\ \text{list exceptions} \end{array} \right\}$$

Lists the current state of the **throw** and **catch** toggles and command-list associated with them.

### mm (memory map)

$$\left\{ \begin{array}{l} \text{mm} \\ \text{memory map} \end{array} \right\} [ \textit{string} ]$$

Shows a memory-map of all currently loaded (mapped) shared-libraries and the main program. If *string* is present, only the memory-map for the named library is listed. The memory-map provides the following information for each loaded region:

- The basename of the library (as used in symbolic names; for example, **libc**).
- The upper and lower bounds of both text and data addresses.
- The handle (see *shl\_load(3X)*).
- The complete path name.
- Information on whether the region is writable (debuggable) or read-only (shared).

## Data Viewing and Modification Commands

- Information on whether symbolic debug (“symdebug”) information is present in the library, and whether it is currently available to the debugger:

```
symdebug: Available
symdebug: Available (but not loaded)
symdebug: Not available
```

where:

**Available** Means that (at least part of) the library was compiled with `-g`, and that references to symbols in the library can be made without qualification.

**Available (but not loaded)** Means that an explicit reference (for example, `symbol@library`) must be made once to force loading of the symbolic-debug information into the debugger.

**Not available** Indicates that no part of the library was compiled with `-g`.

Note that libraries explicitly loaded with `shl_load(3)` are visible to the debugger only until they are unloaded.

### **p (print)**

$$\left. \begin{array}{l} \text{p} \\ \text{print} \end{array} \right\} \left\{ \begin{array}{l} \text{expr} \left[ \left\{ \begin{array}{l} \backslash \\ ? \end{array} \right\} \text{format} \right] \\ \text{class}:: \\ \left[ \begin{array}{l} + \\ - \end{array} \right] \left[ \left[ \backslash \right] \text{format} \right] \end{array} \right\}$$

Displays and optionally modifies program data. You can choose to display data in one of the formats shown in tables “Data Viewing Formats” and “Shorthand Notation for Size.” The `p (print)` command is also used to evaluate arbitrary expressions involving constants and/or program data.

## Data Viewing and Modification Commands

### Displaying Data

The following form of the print command:

```
p class::
```

is used to print all the static data members of a class.

A **format** has the syntax:

```
[ count ] formchar [ size ]
```

- 4 The format specifier *formchar*, which is required, specifies the actual format in which you choose to display the data (see the table “Data Viewing Formats” (Table 4-4) for a list of valid *formchar*’s). The *count* option is the number of times to apply the format. The *size* option is the number of bytes that are formatted for each data item, and overrides the default size for the given format. The count must be a decimal, octal, or hexadecimal number. The size must be a decimal number or one of the letters **b**, **s**, **l**, **D**, or **L** (see the table “Shorthand Notation for Size” in this chapter). For example:

```
>p abc\4x2
```

prints four two-byte numbers in hexadecimal starting at the address designated by the variable `abc`. If `abc` is an array, you need to specify a subscript if you want to see the contents of consecutive array elements. For example:

```
>p abc[5]\4n
```

will display four elements of array `abc`, starting with element 5, in normal (type-dependent) format.

Use the `\format` option to display the value of the expression in a specific format. For example:

```
>print abc\x
```

prints the contents of `abc` in hexadecimal. If a format is not given, the expression is displayed in a format consistent with the type of the expression. For example:

```
>print (abc*3/25)+2
```

prints the results of evaluating the given expression (using the current value of `abc`) in decimal format.

## Data Viewing and Modification Commands

Use the *?format* option to print the address of the evaluated expression in the selected format. For example:

```
>print abc?o
```

prints the address of `abc` in octal. If the expression is not a named data item, `?` is equivalent to `\`.

You can display the contents of absolute addresses with the `p` (`print`) command when you are debugging a program with no debugger information. For example:

```
>p *0xC0000348
```

or

```
>p *($sp-36)\x
```

or

```
>p *_errno@libc
```

Note that using a symbolic address to print a value (`_errno` in this example) requires a dereference operator (`*`).

`p+` prints the next element. Based on the size of the last item displayed, `p+` increments the current data address by the size of the previous format and then displays the contents of memory starting at the new address, using the format if it is supplied, or the previous format, if not supplied. This command is useful for displaying successive elements of an array. The initial `p` (`print`) command can determine the array's format by its type.

`p-` prints the previous element. Based on the size of the last item displayed, `p-` decrements the current data address by the size of the previous format and then displays the contents of memory starting at the new address, using the format if it is supplied, or the previous format, if not supplied.

---

### Note

`p- something` (or `p+ something`) is ambiguous. It could mean print the negative of *something* or it could mean print the next location with format *something*. The debugger will assume that you meant the latter, so if you want the former, use parentheses accordingly: `p (- something)`.

---

## Data Viewing and Modification Commands

### Modifying Data

The `p` (`print`) command is also used to modify the value of a variable. Modification of variables is done by using the assignment operator in the expression (`=` in HP C, HP FORTRAN 77, and HP C++, or `:=` in HP Pascal). For example:

```
>p fob=7
```

In the case of an assignment, the debugger will also show the name of the variable being modified (or the address used if the expression is not a simple name), followed by the assigned value.

4

Here are some special considerations that apply to the `p` (`print`) command.

1. When you try to display a variable which is an HP FORTRAN 77 format label, an HP Pascal file-of-text, or an HP Pascal set, with no display format or with normal format (`\n`), the value is shown as `{format-label}`, `{file-of-text}`, or `{set}`, respectively. You can use other formats, such as `\x`, to display the contents of such variables.
2. When a compiler does not know array dimensions, such as for some HP FORTRAN 77 and HP C array parameters, it uses `0:MAXINT` or `1:MAXINT` as appropriate. The `\t` format shows such cases with `[ ]` (no bounds specified), and subscripts from 0 (or 1) to `MAXINT` are allowed in expressions.

## Data Viewing and Modification Commands

3. Some variables are indirect, so a child process must be active in order for the debugger to know their addresses. When there is no child process, the **address** of any such variable is shown as **0xffffffffe**.

The table “Data Viewing Formats” lists the possible data formats and corresponding *formchars*. Note that there is usually a difference between a lowercase and uppercase character.

For example, the **d** and **D** formats print in short and long decimal:

- d** Displays 16 bits
- D** Displays 32 bits

4

Short and long form apply only to the following formats:

<u>Short</u>	<u>Long</u>
b	B
d	D
e	E
f	F
g	G
o	O
u	U
x	X
z	Z

Many of the the data formats have a default size if the size is not given. For example, **X** has a default size of four bytes. There are also some shorthand notations for *size*. These shorthand notations are shown in the table “Shorthand Notation for Size.” Shorthand notations can be appended to *formchar* instead of a numeric size. For example, the format:

**\xb**

prints one byte in hexadecimal.



## Data Viewing and Modification Commands

There is also a default for the format, if the format is not specified. For example: **D** is the default for a long integer variable or field, **X** is the default for a pointer or array variable or field, and **S** is the default for a structure variable. The **n** format specifies the default. In general, if the expression describes a named data object, the debugger will display its value in a manner consistent with the object's declared type, even if it is a structured type. If the debugger cannot determine the type of an expression or data object, **X** is used.

The following example prints a dynamically allocated C structure that is local to procedure `flavor`.

4

```
>p *flavor:list->head
0x68023004 struct {
    chocolate = 1597845365;
    tuttifrutti = 2.21414e-10;
}
```

## Data Viewing and Modification Commands

**Table 4-4. Data Viewing Formats**

Formchar	Description
a	Prints a string using the expression as the address of the first byte.
(b B)	Prints a byte in decimal.
c	Prints a character.
C	Prints a wide-character. Attempts conversion to the external character-set (as determined by the locale category LC_CTYPE) before printing (see <i>multibyte(3C)</i> ).
(d D)	Prints in decimal as an integer or long integer, respectively.
(e E)	Prints in e floating point notation as a float or double, respectively. (4 bytes, 8 bytes)
(f F)	Prints in f floating point notation as a float or double, respectively.
(g G)	Prints in g floating point notation as a float or double, respectively.
i	Prints a disassembled machine instruction.
k	This is identical to the S format.
K	This is identical to the S format except for C++ <b>class</b> and <b>struct</b> objects where base class and struct data will also be displayed.
n	Prints in normal (default) format, based on the type. (if known)
(o O)	Prints in octal as an integer or long integer, respectively.
p	Prints the name of the procedure containing the given address.
r	Prints the template of an object (C++).
R	Prints the template of an object with base classes displayed (C++).
s	Prints a string using the expression as the address of a pointer to the first byte. In HP C, this is the same as specifying <i>*expr\</i> a.

4

## Data Viewing and Modification Commands

**Table 4-4. Data Viewing Formats (continued)**

Formchar	Description
S	Prints a formatted dump of structures, fields and their values. The expression must be the address of a structure, not the address of a pointer to a structure.
t	Shows the type of the expression ( <i>expr</i> ), usually a variable or procedure name.
T	This is identical to the <b>t</b> format except for C++ <b>class</b> and <b>struct</b> objects where base class and struct type information will also be displayed.
(u U)	Prints the expression ( <i>expr</i> ) in unsigned decimal as an integer or long integer. If the quantity is known to be a full word, <i>u</i> gives the same result as <i>U</i> .
w	Prints a wide character string.
W	Prints address of wide character string.
(x X)	Prints in short and long hexadecimal, respectively. If the quantity is known to be a full word, <i>x</i> gives the same result as <i>X</i> .
(z Z)	Prints in short and long binary, respectively.

**Table 4-5. Shorthand Notation for Size**

Mnemonic	Actual Size
b	1 byte (8 bits)
s	2 bytes (16 bits)
l	4 bytes (32 bits)
D	8 bytes (64 bits) can only be used with floating-point formats
L	16 bytes (128 bits) can only be used with floating-point formats

**pq (print quiet)**

$$\left. \begin{array}{l} \text{pq} \\ \text{print quiet} \end{array} \right\} \left\{ \begin{array}{l} \text{expr} \left[ \left\{ \begin{array}{l} \backslash \\ ? \end{array} \right\} \text{format} \right] \\ \text{class}:: \\ \left[ \begin{array}{l} + \\ - \end{array} \right] \left[ \left[ \backslash \right] \text{format} \right] \end{array} \right\}$$

Does not print anything unless an error occurs. Otherwise, the action is the same as for `p`. The `pq` command can be used to do assignments without causing output. This is useful in breakpoint and assertion command lists. For example:

```
a if $x != x {pq $y = $y + 1; pq $x = x}
```

counts in `$y` how many time the program variable `x` changes value. `$x` and `$y` are special variables. `$y` should be set to zero (`p $y = 0`) before running with this assertion. The use of `pq` instead of `p` keeps the assertion from printing the values of `$x` and `$y` after each change.

---

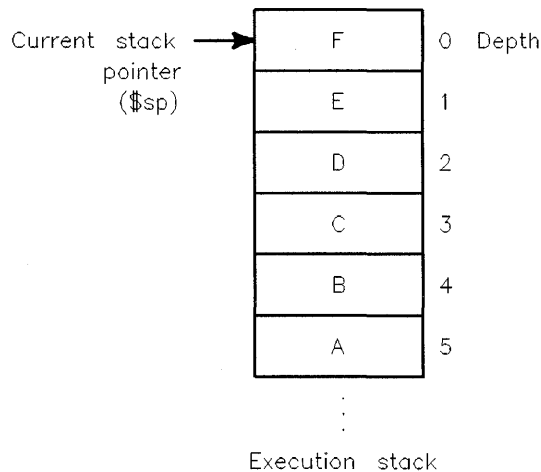
## Stack Viewing Commands

Stack viewing commands trace the stack of a program. The stack viewing commands are:

- t (trace)
- T (Trace)
- V (View)
- up
- down
- top
- tst (toggle stubs)

4

The “Stack Depth” figure illustrates the stack depth of a program and shows that A called B, B called C, C called D, D called E, E called F, and program execution is currently suspended in F. The procedure at which the program is currently stopped is always at depth zero.



**Figure 4-1. Stack Depth**

To make it easier to examine local variables of procedures not at the top (*depth* 0) of the stack, it is possible to set a stack depth to be used by default for all name references. In interpreting variable references where *depth* is not explicitly specified, the debugger will try to use the special variable `$depth` as the default value for the *depth*. If the required procedure (either explicitly specified or taken by default from the current viewing location) is at this *depth* on the stack, the debugger looks for the variable in that stack frame. If the required procedure is not the procedure at that stack *depth*, the debugger looks for the most recent instance of the required procedure by searching down from the top of the stack. If the procedure is found, the debugger looks for the variable in that stack frame.

4

Note that the stack depth is reset to 0 (the top of the stack) by the following commands: `r`, `R`, `c`, `C`, `s`, `S`, `g` and `k`.

### **t (trace)**

$$\left\{ \begin{array}{l} \text{t} \\ \text{trace} \end{array} \right\} [ \text{depth} ]$$

Prints a stack trace. You can optionally specify a *depth*. The default depth is 20 levels. If an optional depth is supplied, only the procedures up to this depth in the stack are displayed. For each procedure in the stack trace, the following is displayed:

- Stack depth
- Name of procedure at that depth
- Name of procedure parameters and their values (printed in normal (`\n`) format). For procedures that are not compiled with the `-g` option, `xdb` displays the name of the procedure, and in parentheses, a best guess at the procedure parameters. For Series 300/400 computers, it is five integers. For Series 600/700/800 computers, it is the first four words of the parameter “spill” area. Note that the procedure might not have “spilled” the four argument registers (the values might still be in the argument registers, or might have been moved to some other registers), and therefore the values printed by `xdb` are not guaranteed to be the correct values of the first four words of the procedure’s argument list.

## Stack Viewing Commands

- Source file and line number where it is suspended (depth 0) or where a call to the next procedure (at the next lowest depth) occurred.

The following example is an example of a trace.

```
>t
 0 icecream (i = 7)      [ice.c: 8]
 1 flavor (year = 1988)  [flavors.c: 19]
 2 main ()              [main.c: 59]
```

4

All arrays, structures, and pointers are shown as addresses. Only the first word of a structure is shown.

(Series 600/700/800) The appearance of stack traces will differ depending on the current state of the **stubs** toggle (see the **tst** command). If the toggle is “on”, any stub that has a return path will be shown as a separate stack frame:

```
 0 func1@liblib1 (param = 0x7af55070)    [sl_lib1.c: 41]
 1 func1@liblib1 + 0x00000008 (hp-ux export stub)
 2 main (argc = 1, argv = 0x7b033590)    [sl_main.c: 153]
```

If the toggle is “off”, stubs are not shown:

```
 0 func1@liblib1 (param = 0x7af55070)    [sl_lib1.c: 41]
 1 main (argc = 1, argv = 0x7b033590)    [sl_main.c: 153]
```

If the current location (level 0) is in a stub, it will be shown regardless of the **tst** toggle.

**T (Trace)**

$$\left\{ \begin{array}{l} \text{T} \\ \text{Trace} \end{array} \right\} [depth]$$

Prints a stack trace with local variables. You can optionally specify a *depth*. The default depth is 20 levels. If an optional depth is supplied, only the procedures up to this depth in the stack are displayed. For each procedure in the stack trace, the following is displayed:

- Stack depth
- Name of procedure at that depth
- Name of procedure parameters and their values (printed in normal (\n) format)
- All local variables and their values (printed in normal (\n) format)
- Source file and line number where execution is suspended (depth 0) or where a call to the next procedure (at the next lowest depth) occurred.

All arrays, structures, and pointers are shown as addresses. Only the first word of a structure is shown.

The following example is an example of a Trace.

```
>T
0 icecream (i = 7)      [ice.c: 8]
   c                = 00000000
1 flavor (year = 1988) [flavors.c: 19]
   harpo            = 1995
   list             = 0x680235bc
2 main ()             [main.c: 59]
   i                = 3
   j                = 2987
   k                = 1988
   icecream         = 0x00000000
   buff            = 0x6802377e
```



## Stack Viewing Commands

### V (View)

$$\left\{ \begin{array}{l} \text{V} \\ \text{View} \end{array} \right\} [ \textit{depth} ]$$

Displays the text for the procedure at the *depth* on the program stack that you specify. If you do not enter a *depth*, the current active procedure is used. This command is normally used to reset the current viewing location to the current point of suspension after it has been moved elsewhere in the program.

- 4 If your terminal supports windowing, the new lines are displayed in the window. Pressing **Return** lets you view successive windows. If your terminal does not support windowing, or if you have forced line mode by including the -L option when invoking the debugger, the current line (including its line number and description) is displayed. Pressing **Return** lets you view the next line in sequence.

Note that you can query the stack *depth* by printing the special variable `$depth`. If you set the `$depth` variable directly (`p $depth = n`), this will set the local context to the specified *depth* but it will not update the source window.

### up

$$\text{up } [ n ]$$

Moves up *n* (default one) levels toward the top of the stack. The default value of *n* is 1. This subtracts *n* from `$depth` (see the section “Special Variables” found in this chapter). The display is updated to view the procedure at the new level. Successive carriage-returns repeat with an offset of 1.

### down

$$\text{down } [ n ]$$

Moves down *n* (default one) levels toward the bottom of the stack. The default value of *n* is 1. This adds *n* to `$depth` (see the section “Special Variables” found in this chapter). The display is updated to view the procedure at the new level. Successive carriage-returns repeat with an offset of 1.

**top**

```
top
```

Moves to the top of the stack. It is shorthand for the debugger command `V 0`, which moves you to the top of the stack. `$depth` (see the section “Special Variables” found in this chapter) is set to 0 and the display is updated to view the procedure where the program is currently stopped.

**tst (toggle stubs) (Series 600/700/800 computers only)**

4

```
{ tst
  toggle stubs }
```

Toggles the visibility of inter-procedural **stubs** as independent contexts (stack frames) in a calling sequence. The `tst` command also affects the appearance of stack traces generated with the `t` (**trace**) and `T` (**Trace**) commands. It also affects the *depth* modifier to various commands (for example, `break uplevel`), and the current value of `$depth`. The default is “off”.

If the toggle is off, stubs are not shown in stack traces. This is especially helpful in programs linked with shared libraries, as stubs appear between each procedure call to/from a shared library.

Here is an example with the `tst` toggle on, indicating that stubs should be visible. An *export* stub is visible between calls into a shared library. If you execute the following command with stubs toggled on:

```
>t
```

the results displayed are similar to this:

```
0 printf@libc + 0x00000004 (0x14, 0x7b033720, 0x7b0335bc, 0x1)
1 printf@libc + 0x00000008 (hp-ux export stub)
2 func1@liblib1 (param = 0x7af55070) [sl_lib1.c: 44]
3 func1@liblib1 + 0x00000008 (hp-ux export stub)
4 func2@liblib2 (param = 0x400012bc) [sl_lib2.c: 33]
5 func2@liblib2 + 0x00000008 (hp-ux export stub)
6 main (argc = 1, argv = 0x7b0335b4) [sl_main.c: 153]
```

## Stack Viewing Commands

With the toggle off, stubs are not shown, and the calling relationships on the stack are much more understandable. If you execute the following command with stubs toggled off:

```
>t
```

the results displayed are similar to this:

```
0 printf@libc + 0x00000004 (0x14, 0x7b033720, 0x7b0335bc, 0x1)
1 func1@liblib1 (param = 0x7af55070)      [sl_lib1.c: 44]
2 func2@liblib2 (param = 0x400012bc)      [sl_lib2.c: 33]
3 main (argc = 1, argv = 0x7b0335b4)      [sl_main.c: 153]
```

4

Since stubs are inserted by the PA-RISC linker to facilitate shared-library calls and to otherwise preserve calling interfaces between modules, they can usually be ignored when you are debugging at the source level. For this reason, the default is “off.”

Note that if the current location (level 0) is in a stub, it will be shown regardless of the state of the `tst` toggle.

The current value of `$depth`, if non-zero, will be reset to its equivalent (with/without stubs) by the `tst` command. You cannot toggle stubs off if `$depth` is currently set (such as with the `V` command) to a stub.

## Status Viewing Command

The status viewing commands display the state of the debugger and the program being debugged. This includes various `list` commands. Refer to the section on Data Viewing and Modification for further information about `list` commands. The other major status viewing command is:

- I (Inquire)

### I (Inquire)

4

```
{ I
  Inquire }
```

Prints the current status of the debugger. The output contains information such as the version number of the debugger, program name, number of source files and procedures, process-ID of the child process, number of breakpoints, record and playback information and so on. A sample output is displayed:

```
Version ..... HP9245X-02A A.09.00 HP SYMBOLIC DEBUGGER (XDB)
Program ..... "tree"
Core File ..... None
Procedures ..... 10
Mapped Images .... 3
Child process .... None
Breakpoints ..... 4 (Active)
Assertions ..... 3 (Suspended)
Macros ..... 9 (Active)
Stubs Visible .... No
Recording ..... Suspended
Record file ..... None
Record-all ..... Active
Record-all file .. mysession
Playback file .... None
Searches ..... NOT case sensitive
Address format ... "%#lx"
Bytes malloc'd ... 7168
Run arguments .... ""
```

---

## Job Control Commands

The job control commands let you control execution of the program. The parent (HP Symbolic Debugger) and child (*object file*) processes take turns running. The debugger is only active and able to execute commands while the child process is stopped due to encountering a signal or a breakpoint, or by terminating.

The job control commands are:

4

- **r** (**run**)
- **R** (**Run**)
- **c** (**continue**)
- **C** (**Continue**)
- **g** (**goto**)
- **k** (**kill**)
- **s** (**step**)
- **S** (**Step**)

Executing any of the above commands resets `$depth` to zero.

### **r (run)**

$$\left\{ \begin{array}{l} \mathbf{r} \\ \mathbf{run} \end{array} \right\} [arguments]$$

Runs a new child process with the *argument* list (if any). The existing child process, if any, is terminated first (after confirmation is given). If no arguments are given, the ones used with the last **r** command are used again (none if **R** was used last).

The *arguments* can contain `<` and `>` for redirecting standard input and standard output. (`<` does an *open(2)* of file descriptor 0 for read-only; `>` does a *creat(2)* of file descriptor 1 with mode 0666). Redirection can also be done with `>>` and `>&`. Arguments can contain shell variables and meta characters, quote marks, or other special syntax (that will be expanded by a Bourne Shell (*sh(1)*)). The remainder of the input line following the **r** command is used as the argument-list, so it cannot be enclosed in a command list (`{ }`). Thus, the **r** command cannot be used within a breakpoint, assertion, or **if** command. The environment for the child process is the same as for the debugger.

**R (Run)**

$$\left\{ \begin{array}{l} \text{R} \\ \text{Run} \end{array} \right\}$$

Lets you run a program as a new child process with no argument list. If a child process already exists, the debugger asks if you want to terminate the child process first. Use this command to explicitly indicate no arguments after previously using the **r (run)** command. The environment for the child process is the same as for the debugger.

4

**c (continue)**

$$\left\{ \begin{array}{l} \text{c} \\ \text{continue} \end{array} \right\} [ \textit{location} ]$$

Resumes execution after a breakpoint has been encountered, ignoring the pending signal, if any. If a *location* is specified, a temporary breakpoint is set at that location. See “Breakpoint Commands” in this chapter for more information.

**C (Continue)**

$$\left\{ \begin{array}{l} \text{C} \\ \text{Continue} \end{array} \right\} [ \textit{location} ]$$

Resumes execution after a breakpoint has been encountered, allowing the pending signal, if any, to be received by the child process. If a *location* is specified, a temporary breakpoint is set at that location. See “Breakpoint Commands” in this chapter for more information.

Continuing with a signal that prevents further execution, such as an untrapped bus error, may cause the signal to be re-asserted or terminate the child process. The pending signal may be examined or modified with the debugger special variable `$signal`.

## Job Control Commands

### g (goto)

$$\left\{ \begin{array}{l} \text{g} \\ \text{goto} \end{array} \right\} \left[ \begin{array}{l} \textit{line} \\ \# \textit{label} \\ + \textit{num} \\ - \textit{num} \end{array} \right]$$

Go to a location in the procedure on the stack at depth zero (not necessarily the same as the current procedure). This changes the program counter so that the first executable statement at or after *line* or *#label* is the next to be executed. The + and - signs:

- In source mode, determine the equivalent *line* by adding (or subtracting) *num* from the line with the current program counter position and then proceed as stated in the previous sentence.

For negative offsets, it is necessary to specify an offset which reaches a line that corresponds to instructions to cause a change in the program counter. (Use **td** (**t**ogg**l**e **d**is**a**ssemb**l**y) to see which source lines have corresponding instructions.)

- In disassembly mode, move the program counter the specified *num* of instructions from the instruction at the current program counter position.

A **g** without arguments is equivalent to **V 0**, which restores the viewing location to the point where execution is suspended.

### k (kill)

$$\left\{ \begin{array}{l} \text{k} \\ \text{kill} \end{array} \right\}$$

Terminates the current child process, if any. You are asked to confirm this command; this guards against accidental termination of the child process.

**s (step)**

$$\left\{ \begin{array}{l} \text{s} \\ \text{step} \end{array} \right\} [number]$$

Single steps through a program, executing one source statement (or machine instruction) at a time before pausing and prompting for another command. In source mode, one source statement is executed (or one step of a multiple step statement in HP Pascal or HP C); in disassembly mode, one machine instruction is executed (several machine instructions might be equivalent to one source statement). If a procedure call is encountered, the procedure is single stepped in the same manner (stepped “into”). Note that *number* must be greater than zero (0).

4

The child process continues with the current signal if any. To prevent the child process from receiving the current signal, set `$signal` to zero (0).

When single-stepping (at the source level) into a non-debuggable procedure, successive instructions will be executed until debuggable code is again reached, or the limit defined by `$step` is reached. At this point the debugger will set an uplevel breakpoint and continue to it, and then again check to see if debuggable code has been reached. As a result, an `s` command at a call to a non-debuggable procedure will frequently behave like the `S` (step-over) command. (See `$step` under “Special Variables” at the beginning of this chapter.)

---

**Note**            One `s (step)` is required to go from the calling statement to the first statement of the called procedure.

---

To execute more than one statement or instruction, enter that number as the *number* parameter. The debugger executes this number of statements or instructions before stopping, unless it encounters a breakpoint first.

You can press `(Return)` to repeat this command. The *number* is discarded.



## Job Control Commands

---

### Note

Single stepping, in disassembly mode, through a procedure for which there is no debugger information (for example, `printf`) can be slow. You might prefer to use the `c` (**continue**) or `S` (**Step**) command instead.

If you accidentally step down into a procedure you don't care about, use the `bu` command to set a temporary "uplevel" breakpoint, and then continue using a `c` (**continue**) command:

```
bu \t Return  
c Return
```

---

4

Issuing an `s` command when stopped at a `throw` statement will cause the debugger to step into the first statement of the first member-function (compiled with the `-g` command-line option) implicitly called as a result of the `throw` statement. If a simple type is thrown (that is, no member functions are implicitly called), the debugger will step directly to the `catch` clause if it was compiled with the `-g` command-line option.

If a statement count is given with the `s` command, the debugger will proceed until either that many statements have been executed, a breakpoint is reached, or the `catch` clause is reached.

### Note

If you single step or run with assertions through a call to `longjmp` (see `setjmp(LIBC)`), the child process will probably take off free-running as the debugger sets but never hits an uplevel breakpoint.

---

## S (Step)

$$\left\{ \begin{array}{l} S \\ \text{Step} \end{array} \right\} [ \textit{number} ]$$

Single steps through a program. In source mode, one source statement (or one step of a multiple step statement in HP Pascal or HP C) is executed; in disassembly mode, one machine instruction is executed (several machine instructions might be equivalent to one source statement). If a procedure call is encountered, it is *not* "stepped into". Instead, execution steps to the statement following the call. The procedure call is treated as a single statement. If a

breakpoint is encountered in the procedure or any that is called, its *commands* are executed. Note that *number* must be greater than zero (0).

The child process continues with the current signal if any. To prevent the child process from receiving the current signal, set `$signal` to zero (0).

---

### Note

Using a `c` (`continue`) command in a breakpoint command list within a procedure will cause the program to keep executing through the procedure! If the breakpoint does not explicitly `continue`, the current act of stepping “over” the procedure ceases. The command:

```
bu \t {}; c Return
```

continues back to the calling statement, effectively completing the `S` (`Step`) command.

---

To execute more than one statement or instruction, enter that number as the *number* parameter. The debugger executes this number of statements or instructions, unless it encounters a breakpoint first.

You can press Return to repeat this command as a single step. The *number* is discarded.

Issuing an `S` command when stopped at a `throw` statement will cause the debugger to step directly to the appropriate `catch` clause. The debugger will execute through any member-functions implicitly called as a result of the `throw` statement unless a breakpoint is encountered in one of those members.

If a statement count is given with the `S` command, the debugger will proceed until either that many statements have been executed, a breakpoint is reached, or the `catch` clause is reached.

---

## Breakpoint Commands

A **breakpoint**, when encountered, suspends the execution of the program at a particular location. HP Symbolic Debugger provides a number of commands for setting, deleting, and managing breakpoints. The breakpoint commands are:

### ■ Overall

- lb (list breakpoints)
- tb (toggle breakpoints)

4

### ■ Creation

- b (breakpoint)
- ba (breakpoint address)
- bb (breakpoint beginning)
- bi (breakpoint instance)
- bpc (breakpoint class)
- bpo (breakpoint overload)
- bt (breakpoint trace)
- bu (breakpoint uplevel)
- bx (breakpoint exit)

### ■ Status

- ab (activate breakpoint)
- bc (breakpoint count)
- db (delete breakpoint)
- sb (suspend breakpoint)

### ■ All-Procedures

- bp (breakpoint procedure)
- bpt
- bpx
- dp (delete procedure)
- Dpt
- Dpx

- Global
  - abc
  - dbc
- Auxiliary
  - *"any string"*
  - i (if)
  - Q (Quiet)

Once a breakpoint has been encountered during program execution, you can interactively examine the program state, unless the breakpoint command list includes a command that causes the child process to continue or terminate. Examples of these commands are the **c** (**continue**), **r** (**run**), **k** (**kill**) and **q** (**quit**) commands.

Breakpoints can be activated or deactivated (suspended) individually. Individual breakpoints are identified by a unique number, which is assigned by the debugger. When a breakpoint is suspended, information for that breakpoint is retained, but it will not affect program execution.

There is also an overall breakpoint mode for breakpoint activation and suspension, which is independent of the state of any individual breakpoint. Any given breakpoint will affect program execution only if it is individually activated and the overall mode is active.

Any active breakpoint whose location is visible in the source window will be marked with an asterisk (\*) in the leftmost screen column. Note that only breakpoints that are associated with a line number are so marked in source mode. In disassembly mode, all breakpoints are displayed, whether associated with a line or machine instruction. A breakpoint set at a location which does not begin a source statement does not show an asterisk marker in the source window unless the debugger is in disassembly mode.

## Breakpoint Commands

Three parameters are associated with breakpoint commands, *location*, *count* and *command list*. These parameters are described below:

*location* You can set a breakpoint at the current **location** (where the prompt (>) appears in the source window) or at any other executable statement or instruction. You can specify the location of the breakpoint in a variety of ways (see the section “Entering Commands” in this chapter for the specific syntax for *location*):

- line number
- procedure name
- label
- symbolic address (with or without offset)
- absolute (numeric) address

Each of these ways of specifying a location is simply an alternate way to specify the breakpoint’s address. The breakpoint is encountered whenever the *location* is about to be executed, regardless of the path taken to get there.

---

### Note

The *location* can be within a procedure linked from a shared library only if the debugger was invoked with the `-s` or `-l` option.

---

*count* The number of times the breakpoint is encountered prior to recognition. A count is of the form `\expr`, `\expr p` (`p` for permanent, the default), or `\expr t` (`t` for temporary). The *count* decrements with each encounter. Each time *count* goes to zero (0), the breakpoint is recognized; otherwise, it is ignored and the *count* is decremented. If the breakpoint is permanent, *count* is reset to the original *count*. If the breakpoint is temporary, once *count* goes to zero (0), the breakpoint is recognized, then deleted.

*command list* A **command list** is one or more commands that are executed when its associated breakpoint occurs. Separate commands in a command list by semicolons. Use braces `{}` to separate the breakpoint command list from other debugger commands on the same line.

---

**Note**

Only one active *command line* can exist at one time. A *command line* is either the sequence of commands you enter at the debugger prompt or the *command list* associated with a breakpoint or assertion. If a breakpoint's *command list* is encountered before all commands in the previous command line are executed, those remaining commands are discarded. For example, suppose you set a breakpoint in a function called `func1` which has the following command list:

```
{Q;p "hello\n";c}
```

Then, from the command line you execute:

```
>p func1();p "goodbye\n"
```

This will print `hello`, but not `goodbye`.

---

## Breakpoint Commands

### Types of Breakpoints

Breakpoints can be separated into two general classes:

- Individual (single) breakpoints

These are explicitly set by the user at a given location or logical group of locations.

- All-Procedure breakpoints

These are breakpoints attached to all debuggable procedures by a single command. They do not have a count or lifespan.

4

The following six breakpoint types are classified as single breakpoints. There can only be one of these at any given location in the code.

Generic	Set with the <b>b (breakpoint)</b> command at a given source-line.
Address	Set with the <b>ba (breakpoint address)</b> command at a given address (which might not correspond directly to a source line).
Procedure beginning (entry)	Set with the <b>bb (breakpoint beginning)</b> command at the first executable statement of a procedure.
Procedure exit	Set with the <b>bx (breakpoint exit)</b> command at the common exit point of a procedure, for example, the procedure epilogue where all returns go through (usually does not correspond to a source line).
Procedure trace (entry/exit)	Set with the <b>bt (breakpoint trace)</b> command at the procedure entry and exit.
Uplevel	Set with the <b>bu (breakpoint uplevel)</b> command at the return address of a given procedure call, at the first instruction executed after the return (which might not correspond directly to a source line).

## Breakpoint Commands

For C++ functions, the following single breakpoints are also available. Multiple breakpoints of these types may co-exist with any other breakpoints at the same location.

Overloaded functions	Set with the <b>bpo</b> ( <b>breakpoint overload</b> ) command at the first executable statement of all functions with the same name.
Instance	Set with the <b>bi</b> ( <b>breakpoint instance</b> ) command at the first executable statement of all or the specified member functions of a class instance.
Class functions	Set with the <b>bpc</b> ( <b>breakpoint class</b> ) command at the first executable statement of all the member functions of a given class.

4

There are three basic types of all-procedure breakpoints. These may co-exist with other all-procedure breakpoints and/or a single breakpoint at a given location.

Procedure (beginning)	Set with the <b>bp</b> ( <b>breakpoint procedure</b> ) command at the first executable statement of all procedures.
Procedure exit	Set with the <b>bpx</b> command at the common exit point of all procedures.
Procedure trace	Set with the <b>bpt</b> command at the entry and exit of all procedures.



## Breakpoint Commands

Notice that at any given procedure entry, it is possible to have multiple command lists associated with the location:

4

Type of Command List	How It Is Set
Global breakpoint command list	Set with the <b>abc</b> command
Individual procedure beginning breakpoint command list	Set with the <b>bb (breakpoint beginning)</b> command
All-procedure beginning breakpoint command list	Set with the <b>bp (breakpoint procedure)</b> command
All-procedure trace breakpoint command list	Set with the <b>bpt (breakpoint trace)</b> command
Overloaded functions breakpoints	Set with the <b>bpo (breakpoint overload)</b> command
Instance breakpoints	Set with the <b>bi (breakpoint instance)</b> command
Class breakpoints	Set with the <b>bpc (breakpoint class)</b> command

Also, at any given procedure exit, up to four command lists can be associated with the location:

Type of Command List	How It Is Set
Global breakpoint command list	Set with the <b>abc</b> command
Individual procedure exit breakpoint command list	Set with the <b>bx (breakpoint exit)</b> command
All-procedure exit breakpoint command list	Set with the <b>bpx</b> command
All-procedure trace breakpoint command list	Set with the <b>bpt</b> command

# Overall Breakpoint Commands

## lb (list breakpoints)

```
{ lb  
  list breakpoints } [ @library-shared ]
```

Displays all breakpoints in the program, both active and suspended, and the overall breakpoint state. For generic breakpoints, the display shows the number, count, status and commands for each breakpoint. The *@shared-library* syntax is used to list only those breakpoints set in the named shared library. The figure “Listing a Breakpoint” gives an example of the information that is displayed for a typical breakpoint. This information is also displayed whenever a breakpoint is added or deleted.

4

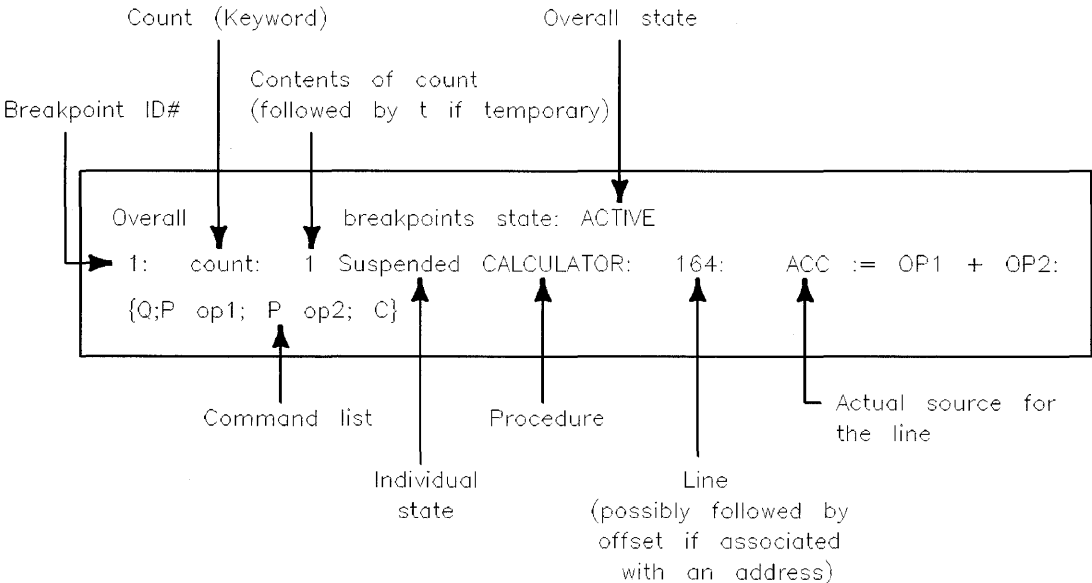


Figure 4-2. Listing a Breakpoint

## Overall Breakpoint Commands

### **tb (toggle breakpoints)**

`{ tb  
toggle breakpoints }`

Toggles the overall breakpoint state from active to suspended or vice versa. The state of the individual breakpoints remains unchanged.

## Breakpoint Creation Commands

### **b (breakpoint)**

$$\left\{ \begin{array}{l} \text{b} \\ \text{breakpoint} \end{array} \right\} [ \textit{location} ] [ \backslash \textit{count} ] [ \textit{command-list} ]$$

Sets a breakpoint at the *location* that you specify. If you do not enter a *location*, the current line in the source or disassembly window is used. The breakpoint is executed on each occurrence (*count*) that you specify. You can enter a list of commands to be executed at the breakpoint by giving a *command-list*. The command list will be executed when the breakpoint is reached and its count is zero. See the definition for *location*, *count*, and *command-list* at the beginning of this section, “Breakpoint Commands.”

In the following example, a breakpoint is set at the current location in the source window and is executed every fourth execution of that statement. Since there is no command list, no commands are executed when the breakpoint is reached. Instead, the debugger will just re-enter command mode at that point.

```
> b \4
```

To set a breakpoint in a different file or procedure, use the **v (view)** command to display the file or procedure in the current viewing location window and search for the line on which to set the breakpoint. If you know where to set the breakpoint in another file or procedure enter this command with the procedure and line. For example, the following command sets a breakpoint at line 355 in procedure `cmp80`.

```
>b cmp80:355
```

To set a breakpoint using a label instead of a line number, enter the label name instead of the line number. For example,

```
>b cmp80#totals
```

## Breakpoint Creation Commands

### ba (breakpoint address)

$\left. \begin{array}{l} \text{ba} \\ \text{breakpoint address} \end{array} \right\} \text{address} [\backslash \text{count}] [\text{command-list}]$

4 Sets a breakpoint at the specified *address*. Note that the *address* can be specified by giving the name of a procedure or an expression containing such a name. The breakpoint is executed on each occurrence (*count*) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the *command-list*. See the definition for *address (location)*, *count*, and *command-list* at the beginning of this section, “Breakpoint Commands”.

The following is an example:

```
>ba printf+0x0018
Overall breakpoints state:  ACTIVE
Added:
  2: count:   1  Active   printf +0x00000018: (line unknown)
```

---

**Caution** Be sure the address given in the **ba (breakpoint address)** command is a valid code address in the child process or errors might ensue. Also, note that this address can be within a procedure linked from a shared library only if the debugger was invoked with the **-s** or **-l** option.

---

(Series 600/700/800 only) A code (linker) symbol used in the address provided to the **ba** command will always be associated with the actual entry point by that name, rather than any **stub**. To view a stub, the **ll (list labels)** command must first be used to locate the actual address of the stub, and that numeric address provided to the **ba** command.

**bb (breakpoint beginning)**

$$\left\{ \begin{array}{l} \text{bb} \\ \text{breakpoint beginning} \end{array} \right\} [depth] [\backslash count] [command-list]$$

Sets a breakpoint at the first executable statement of the procedure at the specified *depth* on the program stack. If you do not enter a *depth*, the procedure shown in the source window is used (this might not be the same as the procedure at depth zero in the stack).

The breakpoint is executed on the occurrence (*count*) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the *command-list*. See the definitions for *depth*, *count*, and *command-list* at the beginning of this section, “Breakpoint Commands”.

4

**bi (breakpoint instance)**

$$\left\{ \begin{array}{l} \text{bi} \\ \text{breakpoint instance} \end{array} \right\} expr.proc [\backslash count] [command-list]$$

After evaluating *expr* to what must be a class instance, set an *instance* breakpoint at the first executable line of *proc* for the instance’s class. This breakpoint is only recognized when *proc* is called for this instance. See the definitions for *count* and *command-list* at the beginning of this section, “Breakpoint Commands”.

If there are commands, they will be executed when the breakpoint is hit. If there are none, the debugger pauses for command input. In cases when the debugger can determine when the given instance will cease to exist because program execution exits the scope in which it is defined, the breakpoint will be automatically deleted upon leaving that scope.

**bi (breakpoint instance)**

$$\left\{ \begin{array}{l} \text{bi} \\ \text{breakpoint instance} \end{array} \right\} \left[ \begin{array}{l} -c \\ -C \end{array} \right] expr [command-list]$$

After evaluating *expr* to what must be a class instance, set *instance* breakpoints at the first executable line of all member functions of the instance’s class. These breakpoints are only recognized when the member functions are

## Breakpoint Creation Commands

called for this instance. See the definition for *command-list* at the beginning of this section, “Breakpoint Commands”.

If *-c* is given, breakpoints will be set only on member functions of the class designated by the object and not of its base classes. If *-C* is given, breakpoints are also set on member functions of base classes. If neither *-c* or *-C* is given, behavior defaults to what is defined by bit 2 of the *\$cplusplus* special variable. See “Entering Commands” at the beginning of this chapter.

4

If there are commands, they will be executed when one of these breakpoints is hit. If there are none, the debugger pauses for command input. In cases when the debugger can determine when the given instance will cease to exist because program execution exits the scope in which it is defined, the breakpoint will be automatically deleted upon leaving that scope.

### bt (breakpoint trace)

$$\left. \begin{array}{l} \text{bt} \\ \text{breakpoint trace} \end{array} \right\} \left[ \begin{array}{l} \textit{proc} \\ \textit{depth} \end{array} \right] [\backslash \textit{count}] [\textit{command-list}]$$

Sets a *trace* breakpoint at the current or named procedure or at the procedure that is at the specified *depth* on the program stack. A breakpoint is set at the entry and exit point of the procedure. The breakpoint is executed on the occurrence (*count*) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the *command-list*. See the definitions for *count* and *command-list* at the beginning of this section, “Breakpoint Commands”.

If you include a *command-list*, it is executed at the beginning of the procedure. The following *command-list* is for Series 600/700/800 computers and will be executed at the end of the procedure.

```
{ Q;p $ret0\d;c }
```

## Breakpoint Creation Commands

The following *command-list* is for Series 300/400 computers and will be executed at the end of the procedure.

```
{ Q;L;c }
```

If you omit a *command-list*, the following *command-list* is executed at the beginning of the procedure.

```
{ Q; t 2; c }
```

The entry command list above displays the two procedures at the top of the stack (the current procedure and the procedure which called it) and their parameters, then continues. For Series 600/700/800 computers, the exit command list prints the return value of the procedure, then continues. For Series 300/400 computers, it prints the current location and continues.

To enter a different command list for the exit point of the procedure or subprogram, use the **bx** (**breakpoint exit**) command.

---

**Note** The default entry and fixed exit *command-lists* contain a **c** (**continue**) command. Single-stepping into these breakpoints will cause the debugger to continue. If this “run-away” behavior is a problem, use the **bb** and **bx** commands with appropriate *command-lists* instead.

---

### bpc (breakpoint class)

```
{ bpc  
  breakpoint class } [ -c ] class [ command-list ]
```

Set *class* breakpoints at the first executable line of all member functions of *class*.

If **-c** is given, breakpoints will be set only on member functions of the designated *class* and not of its base classes. If **-C** is given, breakpoints are also set on member functions of base classes. If neither **-c** or **-C** is given, behavior defaults to what is defined by bit 1 of the **\$cplusplus** special variable. See “Entering Commands” at the beginning of this chapter.



## Breakpoint Creation Commands

When one of these breakpoints is hit, commands are executed. If there are none, the debugger pauses for command input.

### bpo (breakpoint overload)

$$\left\{ \begin{array}{l} \text{bpo} \\ \text{breakpoint overload} \end{array} \right\} \left[ [ \textit{class} ] :: \right] \textit{proc} [ \textit{command-list} ]$$

4 Set *overload* breakpoints at the first executable line of all overloaded functions with name *proc* (which may be qualified by a *class*.) When one of these breakpoints is hit, commands are executed. If there are none, the debugger pauses for command input.

### bu (breakpoint uplevel)

$$\left\{ \begin{array}{l} \text{bu} \\ \text{breakpoint uplevel} \end{array} \right\} [ \textit{depth} ] [ \backslash \textit{count} ] [ \textit{command-list} ]$$

Sets an *uplevel* breakpoint to occur immediately on return to the procedure at the specified *depth* on the program stack. This command is useful for examining values returned from procedures. For example, when execution pauses in procedure B (called from procedure A), you can set an uplevel breakpoint so that a breakpoint occurs when execution returns to procedure A.

If you omit *depth*, 1 is used (0 is the current location). If `$pc` corresponds to the beginning of a source line, a *depth* of 0 is equivalent to:

```
>b
```

The breakpoint is executed on the occurrence (*count*) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the *command-list*. See the definitions for *count* and *command-list* at the beginning of this section, “Breakpoint Commands”.

(Series 600/700/800) The behavior of the `bu` command will differ depending on the current state of the `stubs` toggle (see the `tst` command). If the toggle is “on”, any stub that is in the current call chain will be visible in a stack trace (`t` or `T` commands), and is a potential candidate for an uplevel breakpoint. If the toggle is “off”, stubs are not visible, and breakpoints cannot be set in them with the `bu` command.

## Breakpoint Creation Commands

(Series 600/700/800) Because of the way pointers-to-functions are handled on PA-RISC architectures, a `bu` command with a depth of 2 is required to set the breakpoint in the actual caller of a function called through a function pointer. A depth of 1 will place the breakpoint in the special milli-code routine `$$dynCALL`.

### **bx (breakpoint exit)**

$$\left\{ \begin{array}{l} \text{bx} \\ \text{breakpoint exit} \end{array} \right\} [ \textit{depth} ] [ \backslash \textit{count} ] [ \textit{command-list} ]$$

4

Sets an *exit* breakpoint at the epilogue code of the procedure at the specified *depth* on the program stack. The breakpoint is set at a point such that all returns go through it. If you do not enter a *depth*, the procedure shown in the source window is used (this might not be the same as the procedure at depth zero in the stack).

The breakpoint is executed on the occurrence (*count*) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the *command-list*. See the definitions for *count* and *command-list* at the beginning of this section, "Breakpoint Commands".

---

## Breakpoint Status Commands

### ab (activate breakpoint)

$$\left\{ \begin{array}{l} \text{ab} \\ \text{activate breakpoint} \end{array} \right\} \left[ \begin{array}{l} \textit{number} \\ * \\ \textit{@shared-library} \end{array} \right]$$

4

Activates the breakpoint having the *number* that you specify. If you do not enter a *number*, the breakpoint at the current line is activated if one exists (use the **lb (list breakpoints)** command to determine the number to enter). To activate an instance, class, or overload breakpoint, *number* must be given.

Use the asterisk (\*) to activate all breakpoints, including all-procedure breakpoints. Use the *@shared-library* syntax to activate all breakpoints in the named shared library.

### bc (breakpoint count)

$$\left\{ \begin{array}{l} \text{bc} \\ \text{breakpoint count} \end{array} \right\} \textit{number expr}$$

Sets the count of the specified breakpoint *number* to the integer value of the evaluated expression *expr* that you enter. Use the **lb (list breakpoints)** command to determine the *number* to enter.

---

**Note**            The count may not be changed for all-procedures, class or overloaded breakpoints nor for instance breakpoints which involve all member functions of a class.

---

### db (delete breakpoint)

$$\left\{ \begin{array}{l} \text{db} \\ \text{delete breakpoint} \end{array} \right\} \left[ \begin{array}{l} \textit{number} \\ * \\ \textit{@shared-library} \end{array} \right]$$

Deletes the breakpoint having the *number* that you specify. If you do not enter a *number*, the breakpoint at the current line is deleted. If the breakpoint

## Breakpoint Status Commands

that you specify does not exist, the debugger displays all the breakpoints so that you can select one to delete. To delete an instance, class, or overload breakpoint, *number* must be given.

Use the asterisk (\*) to delete all breakpoints, including all-procedure breakpoints. Use the *@shared-library* syntax to delete all breakpoints in the named shared library.

### sb (suspend breakpoint)

$$\left. \begin{array}{l} \text{sb} \\ \text{suspend breakpoint} \end{array} \right\} \left[ \begin{array}{l} \textit{number} \\ * \\ \textit{@shared-library} \end{array} \right]$$

4

Suspends (deactivates) the breakpoint having the *number* that you specify. If you do not enter a *number*, the breakpoint at the current line is suspended if one exists (use the **lb** (**list breakpoints**) command to determine the *number* to enter). To suspend on instance, class, or overload breakpoint, *number* must be given. To reactivate the breakpoint use the **ab** (**activate breakpoint**) command.

Use the asterisk (\*) to suspend all breakpoints, including all-procedure breakpoints. Use the *@shared-library* to suspend all breakpoints in the named shared library.

---

## All-Procedures Breakpoint Commands

### bp (breakpoint procedure)

$$\left. \begin{array}{l} \text{bp} \\ \text{breakpoint procedure} \end{array} \right\} [\textit{@shared-library}] [\textit{command-list}]$$

4

Sets permanent *procedure* breakpoints at the first executable statement of every procedure for which debugger information is available (this is equivalent to executing a **bb** (**breakpoint beginning**) for every procedure). The breakpoint is encountered each time the procedure is entered. When any procedure breakpoint is encountered, the *command-list* is executed. See the definition for *command-list* at the beginning of this section, “Breakpoint Commands”.

The following example sets breakpoints at the beginning of each procedure. The command list causes the name of the procedure and the values of its arguments to be displayed before continuing.

```
bp {Q; t 1; c}
```

You can set other breakpoints, either permanent or temporary, at the same locations as the procedure breakpoints without superseding them. However, if an all-procedure and a nonprocedure breakpoint are set at the same location, the nonprocedure breakpoint is executed first.

You cannot alter the count of a procedure breakpoint. You also cannot set or delete procedure breakpoints individually. To delete procedure breakpoints, use the **dp** (**delete procedure**) command.

For programs that are linked with debuggable shared libraries (and the **-l** invocation option is used appropriately), an unqualified **bp** command will set a procedure breakpoint at each debuggable procedure in the main program, as well as those in each library that is active (mapped into the process) at the time the **bp** command is issued.

The *@shared-library* syntax can be used to set procedure breakpoints only at the debuggable procedures in the named library. These breakpoints are set *in addition* to any procedure breakpoints already set. Note that each shared library may have its own *command-list* which can be replaced by reissuing the command with a different *command-list*.

**bpt**

```
bpt [@shared-library] [command-list]
```

Sets permanent *procedure trace* breakpoints at the first and last executable statement of every procedure for which debugger information is available. The breakpoints are encountered each time the procedure is entered and exited. The *command-list*, if any, is associated with the entry breakpoint. See the definition for *command-list* at the beginning of this section, “Breakpoint Commands.” See also the discussion on shared libraries for the `bp` command above.

4

If no command list is specified, the entry command list defaults to:

```
{Q;t 2;c}
```

The exit *command-list*, for Series 600/700/800 computers, is:

```
{Q;p $ret0\d;c}
```

The exit *command-list*, for Series 300/400 computers, is:

```
{Q;L;c}
```

You can set other breakpoints, either permanent or temporary, at the same locations as the procedure breakpoints without superseding them. However, if an all-procedure and a nonprocedure breakpoint are set at the same location, the nonprocedure breakpoint is executed first.

You cannot alter the count of a procedure trace breakpoint. You also cannot set or delete procedure breakpoints individually. To delete procedure trace breakpoints, use the `Dpt` command.

**Note**


---

The default entry and fixed exit *command-lists* contain a `c` (**continue**) command. Single-stepping into these breakpoints will cause the debugger to continue. If this “run-away” behavior is a problem, use the `bp` and `bpX` commands with appropriate *command-lists* instead.

---

## All-Procedures Breakpoint Commands

### **bpx**

```
bpx [@shared-library] [command-list]
```

Sets permanent *procedure exit* breakpoints after the last executable statement of every procedure for which debugger information is available. The breakpoint is encountered each time the procedure is exited. When any procedure exit breakpoint is encountered, the *command-list* is executed. See the definition for *command-list* at the beginning of this section, “Breakpoint Commands”.

- 4 You can set other breakpoints, either permanent or temporary, at the same locations as the procedure breakpoints without superseding them. However, if an all-procedure and a nonprocedure breakpoint are set at the same location, the nonprocedure breakpoint is executed first.

You cannot alter the count of a procedure exit breakpoint. You also cannot set or delete procedure exit breakpoints individually. To delete procedure exit breakpoints, use the **Dpx** command.

Also see the discussion on shared libraries for the **bp** command.

### **dp (delete procedure)**

```
{ dp  
  delete procedure } [@shared-library]
```

Deletes all all-procedure breakpoints set with the **bp** (**breakpoint procedure**) command. All breakpoints set by commands other than the **bp** command will remain set.

You cannot delete procedure breakpoints individually.

The *@shared-library* syntax can be used to only delete those procedure breakpoints in the named library. All other procedure breakpoints currently defined remain in place.

### Dpt

Dpt [*@shared-library*]

Deletes all *procedure trace* breakpoints at the first and last executable statement of every procedure. All breakpoints set by commands other than the **bpt** command will remain in effect.

You cannot delete procedure trace breakpoints individually.

The *@shared-library* syntax can be used to only delete those procedure trace breakpoints in the named library. All other procedure trace breakpoints currently defined remain in place.

4

### Dpx

Dpx [*@shared-library*]

Deletes all *procedure exit* breakpoints at the last executable statement of every procedure. All breakpoints set by commands other than the **bpx** command will remain in effect.

You cannot delete procedure exit breakpoints individually.

The *@shared-library* syntax can be used to only delete those procedure exit breakpoints in the named library. All other procedure exit breakpoints currently defined remain in place.



---

## Global Breakpoint Commands

### **abc**

*abc command-list*

4 Defines a global breakpoint *command-list* which will be executed whenever any user-defined breakpoint is encountered. This includes generic, procedure, address, procedure trace, procedure exit, instance, class, and overload breakpoints. These commands will be executed before any commands associated with the breakpoint. See the definition for *command-list* at the beginning of this section, “Breakpoint Commands”.

This example suppresses the “breakpoint at address” message normally printed for all breakpoints.

```
>abc Q
```

### **dbc**

*dbc*

Deletes the global breakpoint command list.

## Auxiliary Breakpoint Commands

Although the *any string*, *if*, and *Quiet* commands are not actually breakpoint commands, they are used almost exclusively in breakpoint and assertion command lists. Consequently, they are documented here.

### “*any string*”

`"any string"`

Causes any string that is enclosed in quotation marks to be echoed to the screen. This string command is useful for labeling breakpoint output, particularly for recording a debugger session. You can include character escape sequences in the string (for example, `\t`). See the table “Escape Sequences” for more information.

In the following example, the “*any string*” command is used to label the display of a data-item which otherwise doesn’t have a name (the debugger just prints an address in such cases). Note the use of the character escape `\n` (newline).

```
>"flavor_list head =>\n"; p *flavor:list->head
flavor_list head =>
0x68023004 struct {
    chocolate = 1597845365;
    tuttifrutti = 2.21414e-10;
}
```

### **i (if)**

$$\left\{ \begin{array}{l} i \\ if \end{array} \right\} expr \{command-list\} [\{command-list\}]$$

Lets you conditionally execute commands in a command list. If the expression evaluates to a non-zero value, the first group of commands is executed. If the expression evaluates to zero, the second command list (if it exists) is executed. The command lists must be enclosed in braces (`{ }`). The **i (if)** command can be nested in other command lists.

## Auxiliary Breakpoint Commands

The following **b** (**breakpoint**) command (set at entry to procedure **proc**) uses the **i** (**if**) command to conditionally print a value only if a certain condition is true. Execution always continues after executing this command list:

```
>b proc {Q; if (list->head.fld > 0) {p list->head.name}; c }
```

## Q (Quiet)

```
{ Q  
  Quiet }
```

4

Suppresses the “breakpoint at address” debugger messages that are normally displayed when a breakpoint is encountered. This enables you to display variable values without cluttering the command window. The **Q** (**Quiet**) command must be the first command in a command list; otherwise, it is ignored.

## Exception Handling Commands

HP's symbolic debugger provides the following exception handling support:

- It provides the ability to stop at (prior to execution) any **throw** statement and optionally execute a command-list. This ability to stop at any **throw** statement can be toggled and is *on* by default.
- It notifies you that a throw is about to occur and lets you know approximately where the exception will be caught.
- It provides the ability to stop at the first statement of any **catch** clause and optionally execute a command list. This ability to stop at the first statement of a **catch** clause can be toggled and is *on* by default.
- It notifies you that a **catch** has occurred and lets you know where the exception was thrown from.
- It lists the current toggle status of the debugger exception handling commands.
- It provides the ability to step directly from a **throw** statement to its corresponding **catch** statement.
- It provides the ability to explicitly prevent destruction of auto-objects during the stack-unwinding that follows an exception throw.

The exception handling commands are:

- `txt` (toggle exception throw)
- `xtc` (exception throw command)
- `txc` (toggle exception catch)
- `xcc` (exception catch command)

### **txt** (toggle exception throw)

```
{ txt
  toggle exception throw }
```

Turns off and on the stopping of the debugger immediately prior to an exception throw. By default, the debugger stops immediately prior to an exception throw.

## Exception Handling Commands

### **xtc (exception throw command)**

$$\left\{ \begin{array}{l} \text{xtc} \\ \text{exception throw command} \end{array} \right\} [ \textit{command-list} ]$$

Defines a debugger *command-list* to be executed when a stop on throw occurs.

### **txc (toggle exception catch)**

4 
$$\left\{ \begin{array}{l} \text{txc} \\ \text{toggle exception catch} \end{array} \right\}$$

Turns off and on the stopping of the debugger at the first statement of any catch clause. By default, the debugger stops at the first statement of any catch clause.

### **xcc (exception catch command)**

$$\left\{ \begin{array}{l} \text{xcc} \\ \text{exception catch command} \end{array} \right\} [ \textit{command-list} ]$$

Defines a debugger *command-list* to be executed when a stop on catch occurs.

---

## Assertion Control Commands

An **assertion** is a list of one or more debugger commands that are executed before each machine instruction. Assertions are useful for tracing serious software defects, such as corrupt global variables, or mysterious side effects. The assertion control commands are:

- `a` (`assert`)
- `aa` (`activate assertion`)
- `da` (`delete assertion`)
- `la` (`list assertions`)
- `sa` (`suspend assertion`)
- `ta` (`toggle assertions`)
- `x` (`exit`)

4

Assertions can be activated or inactivated (suspended) individually. When an assertion is suspended, information for that assertion is retained, but it will not be evaluated during program execution. There is also an overall assertion mode for assertion activation and suspension which is independent of the state of any individual assertion. Any given assertion will be evaluated during program execution only if it is individually activated and the overall mode is active.

Assertions are not evaluated during single-step execution. They are only evaluated during execution following a `run` or `continue` command.

The `if`, `Quiet` and “*any string*” commands are useful in assertion command lists. For more information about these commands, see the subsection called “Auxiliary Breakpoint Commands” in the “Breakpoint Commands” section.

---

### Note

Assertions slow down program execution because the commands for all active assertions are executed before each machine instruction in the program. If you use the assertion control commands in a breakpoint command list, you will be able to limit the regions of slowed execution to your actual areas of interest in the program. See the section “Hints for Using Assertions” in the appendix “Limitations and Hints” for an example of how to use assertions in a useful way.

---

## Assertion Control Commands

### a (assert)

$\left. \begin{array}{l} \text{a} \\ \text{assert} \end{array} \right\} \text{command-list}$

Creates an assertion consisting of the *command-list* that you enter. You can enclose an assertion *command-list* in braces to separate it from other commands on the same line. Errors in assertion command lists are not identified until the assertion is executed. If there is an error, an error message is displayed, but execution continues. Assertions, like breakpoints, are identified by a unique number assigned by the debugger. They also have an overall state, whereby all assertions can be activated or suspended as a group. Use the `la` (**list assertions**) command to see a list of assertions, their identifying numbers, and the overall state.

---

#### Note

In an assertion *command-list*, you can use the following job control commands only after an `x` (**exit**) command, which suspends execution of the program.

- `r` (run)
- `R` (Run)
- `c` (continue)
- `C` (Continue)
- `s` (step)
- `S` (Step)
- `k` (kill)

Also, job control commands cannot be used in an assertion command list unless all assertions are suspended first. The following is an example of a typical assertion command sequence.

```
{if(i != 0) {ta;x 1;c}}
```

---

The following examples show how to use the `a` (**assert**) command.

```
a {L}
```

## Assertion Control Commands

This “assert list” command traces program execution one line at a time until the program stops. (The program stops on normal termination, when a breakpoint is encountered or when your break character is pressed).

```
a [L; if (xyz > (def-9) *10) {ta;x 1; c} {p abc -= 10}}
```

This assertion displays the line that will be executed next, then checks the `if` statement condition. If it is true, assertion mode and all assertions are suspended, and the program continues executing. If the condition is false, the value of `abc` is decremented by 10, the next source line is executed, and the command list is executed again. The number after the `exit` command (`x 1`) enables the debugger to recognize the continue command which follows it. If just `x` or (`x 0`) was used, the remainder of the command would not be executed, and the debugger would again prompt for commands as if a breakpoint were reached. Note that the `ta` (**toggle assertions**) command is used to toggle assertions to suspend them because the `c` (**continue**) command cannot be used while assertions are active.

```
a {if (abc .NE. $abc) {p $abc = abc; if (abc .GT. 9) {x} } p abc}
```

This command list displays the value of the global variable, `abc`, and suspends program execution if the variable exceeds a certain value. `$abc` is a special variable that keeps track of when the value of `abc` changes.

---

### Note

If you single step or run with assertions through a call to `longjmp` (see `setjmp(LIBC)`), the child process will probably take off free-running as the debugger sets but never hits an uplevel breakpoint.

---

### aa (activate assertion)

$$\left\{ \begin{array}{l} \text{aa} \\ \text{activate assertion} \end{array} \right\} \left[ \begin{array}{l} \textit{number} \\ * \end{array} \right]$$

Activates the assertion having the *number* that you specify. Use the `la` (**list assertions**) command to determine the *number* associated with an assertion. Using the `*` option causes all assertions to be activated.

Overall assertion mode is activated if any individual assertion is activated.



## Assertion Control Commands

### da (delete assertion)

$$\left\{ \begin{array}{l} \text{da} \\ \text{delete assertion} \end{array} \right\} \left[ \begin{array}{l} \textit{number} \\ * \end{array} \right]$$

Deletes the assertion having the *number* that you specify. Use the **la (list assertions)** command to determine the *number* associated with an assertion. Using the **\*** option causes all assertions to be deleted.

### 4 la (list assertions)

$$\left\{ \begin{array}{l} \text{la} \\ \text{list assertions} \end{array} \right\}$$

Lists the number, the state (active or suspended) and the command list for each assertion, as well as the overall assertion state (active or suspended).

Use this command to find the number of a particular assertion before using the **aa (activate assertion)**, **da (delete assertion)** and **sa (suspend assertion)** commands.

The following example lists the status of two assertions:

```
Overall assertion state: ACTIVE

1: Active      if(abc.NE.$abc){p $abc = abc; if(abc.GT.9){x}}

2: Suspended  L;if(xyz.GT.(def-9)*10) {ta;x 1;c} {p abc-=10}
```

### sa (suspend assertion)

$$\left\{ \begin{array}{l} \text{sa} \\ \text{suspend assertion} \end{array} \right\} \left[ \begin{array}{l} \textit{number} \\ * \end{array} \right]$$

Suspends the assertion having the *number* that you specify. Use the **la (list assertions)** command to determine the *number* associated with an assertion. Using the **\*** option causes all assertions to be suspended.

Suspended assertions continue to exist but are not evaluated until activated again. Overall assertion mode is suspended if the last active assertion is suspended.

**ta (toggle assertions)**

$$\left\{ \begin{array}{l} \text{ta} \\ \text{toggle assertions} \end{array} \right\}$$

Toggles the overall assertion state between active and suspended. The overall assertion state does not affect the state of individual assertions.

**x (exit)**

$$\left\{ \begin{array}{l} \text{x} \\ \text{exit} \end{array} \right\} [ \text{expr} ]$$

4

Causes program execution to stop as if a breakpoint has been reached. A message like the following will be printed:

```
Hit on assertion number: command-list
Last line executed was:
    file: source text
Next line to execute is:
    file: source text
```

If the expression (*expr*) is not given or it evaluates to zero, the debugger returns to command mode, ignoring any remaining commands in the assertion command list. If *expr* evaluates to non-zero, any remaining commands in the command list are executed. This command can only be used in an assertion command list.

---

## Record and Playback Commands

The record and playback commands allow reproduction of an HP Symbolic Debugger session by saving debugger commands in a file, which can later be used to execute the commands. The record and playback commands are useful for finding bugs that require many debugger actions to isolate or reproduce. The record-all command is useful for saving a log of the entire session.

The record and playback commands *do not*:

- 4 ■ Save debugger responses to commands in the record file. An exception to this is the record-all command that logs all debugger output as well as user input to the debugger. Note that a record-all file cannot be used as a playback file.
- Record commands in command lists for breakpoints and assertions as they are executed. The only commands recorded are those read from the keyboard or a playback file.
- Copy command lines that begin with > , < , or ! to the current record file. However, this limitation can be overridden by beginning those lines with one or more spaces before the command character.
- Record output from the user program (child process). This may be done using output redirection (>) in the **r** (run) command line, or the **-e** and **-o** invocation options.

The table “Record and Playback Commands” lists the record, record-all, and playback commands. The record-all commands are used to log all of the output generated in the command window by the debugger. You should remember that output generated by the child process is *not* recorded.

---

<b>Caution</b>	Do not try to play back from a file currently opened for recording or record from a file currently opened for playback. This could cause problems with your debugger session.
----------------	---

---

## Record and Playback Commands

**Table 4-6. Record and Playback Commands**

Command	Description
> <i>file</i>	Sets or changes the record file to <i>file</i> , turns recording on, rewrites the file from the beginning, and only records commands. If <i>file</i> exists, you are asked if you want to overwrite.
>> <i>file</i>	Sets or changes the record file to <i>file</i> , turns recording on, and only records commands. All recording is appended to the existing <i>file</i> ; otherwise, a new file is created.
>	Displays the recording state and the current recording file. Can also use >>.
< <i>file</i>	Starts playback from the <i>file</i> .
<< <i>file</i>	Starts playback from the <i>file</i> using a “line-at-a-time” feature. Each command line from the playback file is shown before it is executed, and the debugger provides a list of the following options for you to take some action:  <i>command</i> (<cr>,S, <num>, C, Q, or ?):  You can use any of the above options as described:  <cr>    execute one command line S        skip one command line <num>   execute number of command lines C        continue through all playback Q        quit playback mode ?        gives this explanation of the above commands
tr	<b>toggle record</b> Toggles recording; toggles the state of the record mechanism between active and suspended.

## Record and Playback Commands

**Table 4-6. Record and Playback Commands (continued)**

Command	Description
>t	Turns recording on. (active) <sup>1</sup>
>f	Turns recording off. (suspended) <sup>1</sup>
>c	Closes the record file. <sup>1</sup>
>@file	Sets or changes the <i>record-all</i> file to <i>file</i> , rewrites from the beginning, and turns recording on. If <i>file</i> exists, you are asked if you want to overwrite. Captures all input to and output from the debugger command window, except user program output.
>>@file	Sets or changes the <i>record-all</i> file to <i>file</i> , and turns recording on. Appends <i>record-all</i> output to the existing <i>file</i> . Captures all input to and output from the debugger command window, except user program output.
>@	Displays the current <i>record-all</i> state and file. <sup>1</sup> Can also use >>@.
tr @	<b>toggle record @</b> Toggles the state of the <i>record-all</i> mechanism between active and suspended.
>@t	Turns <i>record-all</i> on.
>@f	Turns <i>record-all</i> off.
>@c	Closes the <i>record-all</i> file.

<sup>1</sup> In order to record to a file named **t**, **f**, **c**, or **@** use **./t**, **./f**, **./c**, or **./@**.

---

## Macro Facility Commands

The macro facility allows you to substitute your own names for debugger commands, sequences of debugger commands, or expressions. To do so, you simply define the text to be used as a straight replacement for the macro name. Thereafter, you can use your newly defined macro name to represent the debugger commands or expressions while inside a debugger session.

The macro commands are:

- `def`
- `tm (toggle macros)`
- `undef`

4

When defining a macro, replacement text is not immediately scanned for additional macro invocations. Rather, macro substitutions are performed as late as possible by HP Symbolic Debugger. This means that when a macro is referenced and has been evaluated, its replacement text is rescanned to determine if the replacement text contains any additional macros. Macros are not recognized inside character constants, strings, or comment (`#`) commands during command line processing.

You can use the macro facility to give your favorite names to the debugger commands. For example, you might define `bplist` to be `list breakpoints` (equivalently, `lb`).

---

**Note**                      Macros do not allow argument substitution, and they cannot be used to modify debugger command syntax.

---

The invocation of recursive macros is trapped and terminates with an error message. Recursive macros are macros whose replacement text contains another reference to the same macro, or to a macro whose expansion eventually references the same macro. For example,

```
define a a
```

is flagged as an error.

Macros are not recognized unless the state of the macro mechanism is activated with the `tm (toggle macros)` command. If you want to see a list of your

## Macro Facility Commands

macros and their current state (active or suspended), use the `lm` (`list macros`) command.

### **def**

```
def name replacement-text
```

Defines a macro substitution for HP Symbolic Debugger commands or expressions. The argument *name* can be any string of letters or digits, beginning with a letter. The argument *replacement-text* can be any string of letters, blanks, tabs or other printing characters that represent one or more debugger commands or expressions. The string begins with the first non-white-space character following *name* and ends with the first `(Return)`. For example:

```
>def ptuti p flavor:list->head.tuttifrutti
ptuti ==> p flavor:list->head.tuttifrutti
```

---

### **Note**

If a macro is defined with the same name as a previous macro, the new definition will replace the old one, until it is undefined with the `undef` command, at which point the old definition is again active.

---

### **tm (toggle macros)**

```
{ tm
  toggle macros }
```

Toggles the state of the macro mechanism between active and suspended. When macros are suspended, the currently defined macros continue to exist, but are not replaced in the command line by their definitions. Additional macros can be defined while the macro state is suspended.

### **undef**

`undef`  $\left\{ \begin{array}{l} \textit{name} \\ * \end{array} \right\}$

Removes the macro defined by *name*. Using the \* option causes all macros to be removed.



## Signal Control Commands

### lz (list signals)

```
{ lz  
  list signals }
```

Lists the current handling of all signals sent to the child process. When this command is entered, a five column table is displayed as shown below.

4

```
hpterm  
31: main()  
32: { int i;  
> 33:   for (i=0; (i++)<10; i++) {  
34:     printf("%d ",i);  
File: test_prog.c Procedure: main Line: 33  
Sig Stop Ignore Report Name  
1 Yes No Yes hangup  
2 Yes Yes Yes interrupt  
3 Yes No Yes quit  
4 Yes No Yes illegal instruction  
5 Yes Yes Yes breakpoint  
6 Yes No Yes IOT instruction  
7 Yes No Yes EMT instruction  
8 Yes No Yes floating point exception  
9 No No No kill  
10 Yes No Yes bus error  
11 Yes No Yes segmentation violation  
12 Yes No Yes bad arg to sys call  
13 Yes No Yes write on pipe with no reader  
14 No No No alarm clock  
15 Yes No Yes software termination  
16 Yes No Yes user signal 1  
17 Yes No Yes user signal 2  
--More--
```

Figure 4-3. Signal Numbers for the z Command

## Signal Control Commands

The columns shown in the previous display are defined as follows:

**Sig** is the signal number.

**Stop** is set to either **Yes** to stop on the signal or **No** to continue.

**Ignore** is set to either **Yes** to ignore the signal or **No** to assign it to `$signal`. It will be passed to the child process if **Stop** is **No** or if the **C**, **s** or **S** command is used after the child process is stopped by the signal.

**Report** is set to either **Yes** to report the signal to the user or **No** to not report it.

**Name** is what the signal does. For the actual signal name, see the *signal(5)* command in the *HP-UX Reference*.

4

## Signal Control Commands

### z (signal)

$$\left\{ \begin{array}{l} z \\ \text{signal} \end{array} \right\} \textit{number} \left[ \begin{array}{l} i \\ r \\ s \\ Q \end{array} \right]$$

4 Modifies the `signal` handling table. The *number* must be a valid signal number (see *signal(5)* in the *HP-UX Reference*). The options (which must appear as one token) toggle the state of the appropriate flag: `i`)gnore, `r`)report, or `s`)top. If `Q` is present, the new state of the signal is not printed. The default is to print the new state of the signal after toggling the flags. If no options are given, the current state of the signal is printed.

For example, assume that the current state of the *alarm clock* signal is: do not ignore, do not report, and do not stop (that is, silently pass the signal directly to the child process). To modify the signal to: stop, do not ignore, and report, you would execute this command:

```
z 14 sr Return
```

The results from executing the previous command are:

Sig	Stop	Ignore	Report	Name
14	Yes	No	Yes	alarm clock

To return back to the previous signal state for *alarm clock*, execute this command:

```
z 14 sr Return
```

When the child process stops or terminates on a signal, it is always reported (except for the `breakpoint` signal with a breakpoint command list starting with `Q`).

When the debugger ignores a signal, `$signal` does not get set, the `C` command is not made aware of it, and the signal cannot be passed to the child process. Signals indicating the child process cannot continue execution (e.g. `SIGILL`, `SIGSEGV`) can be ignored, but strange results may occur.

Note that the debugger catches all signals bound for the child process before the child process sees them (this is a function of the *ptrace(2)* mechanism used for tracing processes). For many signals, this is a reasonable thing to do. Most

## Signal Control Commands

programs are not set up to handle segmentation errors, etc. However, some programs do quite a bit with signals and the constant need to continue from a caught signal can be tedious. The **z** command can be used to simplify this task.

---

**Note** Since *ptrace(2)* cancels all pending signals before servicing a `PT_CONTIN` or a `PT_SINGLE` request, any signals received by the child process while it is suspended (for example between **S** commands) will be lost. In other words, any signal that arrives while the debugger is waiting for a user command will be lost.

---

A signal can be manually sent to the child process by assigning the signal value to the special variable `$signal` and either using the **C**, **s**, or **S** command. For example, to send a *bus error* signal to a program being debugged, execute the following commands:

```
p $signal = 10 (Return)
S (Return)
```

Note that if your program has a signal handler that handles bus errors, the symbolic debugger will step “into” the signal handler and not “over” it as a result of the **S** command given above.

---

## Miscellaneous Commands

The miscellaneous commands perform a variety of individual tasks. The miscellaneous commands are:

- !
- #
- Return
- ~
- am (activate more)
- sm (suspend more)
- f (format)
- h (help)
- q (quit)
- ss (save state)
- tc (toggle case)
- M (Map)
- Mc
- Mt
- tM (toggle maps)

4

!

! [*command*]

Shell-escapes out of the debugger into the operating system. If a command is specified, it is automatically executed. Otherwise, a shell is invoked and must be explicitly exited before the debugger can resume. When you execute the ! command interactively, return to the debugger by hitting the Return key after being prompted to do so. When you use this command in an assertion or breakpoint command list, control returns to the debugger automatically.

## Miscellaneous Commands

A command can be enclosed in braces (`{}`) to delimit it from debugger commands on the same line. For example:

```
b 14 {!ls -l}; continue}; trace; list assertions
```

If you use the escape without giving a list of commands, you are given a shell prompt. You can now execute any HP-UX command. You can return to the debugger by typing `exit` at the shell prompt.

**#**

```
# [text]
```

Causes the *text* to be interpreted as a comment. This command can be used to document the contents of record and playback files. The number symbol (`#`) must be the first non-blank character on the line. The rest of the line is treated as a comment and is written to the record file if the recording is on. Otherwise, it is ignored.

**Return**

**Return**

Repeats the previous command. You can only use this command with the following commands:

```
+  
-  
p (print)  
v (view)  
+r  
-r  
s (step)  
S (Step)
```

This command is synonymous with the `~` command. Any count associated with the repeated command is discarded.

## Miscellaneous Commands

~

~

Repeats the previous command. You must use the **Return** key after typing the ~. You can use this command with the following commands:

```
+  
-  
p (print)  
v (view)  
+r  
-r  
s (step)  
S (Step)
```

4

This command is synonymous with the **Return** command, but of the two, it is the form required in a playback file, and **Return** is recorded in record files as ~. Any count associated with the repeated command is discarded.

### am (activate more)

```
{ am  
  activate more }
```

Activates (enables) the **more** feature. (Active is the initial state). When activated, all command window output following a debugger command is presented to you a window-full at a time, and you are prompted before displaying successive windows.

Use one of the following commands to continue from the **--More--** prompt.

- |                  |   |
|------------------|---|
| <b>Space Bar</b> | Displays one more window-full.  |
| <b>Return</b>    | Displays one more line.   |
| <b>q</b>         | Quits scrolling and ignores the rest of the output until another debugger prompt is issued. |

## Miscellaneous Commands

To view command window output in a continuous stream, use the **sm** (**suspend more**) command to suspend the **more** feature. **(CTRL)S** may be used to temporarily suspend scrolling when the **more** feature is suspended. Use **(CTRL)Q** to continue scrolling.

---

**Note** Output from the child process (program being debugged) also appears in the command window, but it is *not* controlled by the **more** feature.

---

### **sm (suspend more)**

4

{ **sm**  
  **suspend more** }

Suspends the **more** feature and lets you view the output in a continuous stream. **(CTRL)S** and **(CTRL)Q** can be used to temporarily suspend scrolling when the **more** feature is suspended.

Use this command when you do not want the debugger to pause at the end of each window of output waiting for a continuation command. This command is particularly useful when you are using **record-all** to collect large amounts of output in a file for later review. To view the command window output one window-full at a time, use the **am (activate more)** command to activate the **more** feature.



## Miscellaneous Commands

### f (format)

$$\left\{ \begin{array}{l} \mathbf{f} \\ \mathbf{format} \end{array} \right\} [ \text{"printf-style-format"} ]$$

Sets the printing format used by the debugger to print an address (see *printf(3S)* in the *HP-UX Reference*) for a discussion of valid formats).

Using the **f (format)** command without an argument will reset the format to the default format: 8 hexadecimal digits, preceded by "0x".

4

---

**Note** This command is generally not needed for typical debugger use.

If you set the address printing format to something *printf* does not like, you might get an error (usually memory fault) each time you try to print an address, until you fix the format with another **f (format)** command.

---

### h (help)

$$\left\{ \begin{array}{l} \mathbf{h} \\ \mathbf{help} \end{array} \right\} [ \text{topic} ]$$

Prints a command summary which describes the syntax and use of each command. This facility references the short form of the command only, not the long form.

If no *topic* is given, the entire **help** text is displayed. If a *topic* is given, only the text related to that *topic* is displayed. Available topics include the abbreviated form of each command, which shows the syntax and a brief description of the command. To get a list of other topics, use the command:

```
h help
```

The **more** facility is used to display the file. The **help** text is displayed a window full at a time, and you are prompted before displaying successive windows.

## Miscellaneous Commands

Use one of the following commands to continue from the `--More--` prompt.

`Space Bar`      Displays one more window-full.

`Return`          Displays one more line.

`q`                Quits scrolling and ignores the rest of the help information.

Note that the `sm` command (`suspend more`) does not apply to `help` output.

The file `xdb.help.nro` contains `nroff(1)` coded source for the `xdb.help` file. Executing the following command:

```
nroff xdb.help.nro > file-name
```

produces a formatted copy suitable for printing or direct viewing with `more(1)`.

### q (quit)

$$\left\{ \begin{array}{l} \text{q} \\ \text{quit} \end{array} \right\}$$

Quits the debugger after asking for confirmation: enter `y` (yes) or `n` (no). This command returns control to the shell and terminates the debugging session. All files are closed and the terminal is restored to a normal mode.

### ss (save state)

$$\left\{ \begin{array}{l} \text{ss} \\ \text{save state} \end{array} \right\} \text{file}$$

Save the current set of breakpoints, macros, and assertions in *file*. This file can then be used with the `-R` option to restore this information to another invocation of the debugger on the *same* object file.

This file may also be used as a playback file. It should be noted that this bypasses the verification the debugger attempts with the `-R` option. The recorded locations of breakpoints may not be valid in a new object file. You should be sure to read the limitations section on `ss (save state)` files before trying to use one as a playback file. See “Save State Limitations” in the appendix “Limitations and Hints.”

## Miscellaneous Commands

### tc (toggle case)

{ tc  
toggle case }

Toggles case sensitivity; determines whether or not searches or names are case sensitive (initially, they are case insensitive). This command affects file and procedure names, variables, and search strings used with the / or ? commands.

4

---

### Note

Case insensitive searches equate some non-letters with other non-letters. For example, [ and { are equal, as are @ and '.

---

### M (Map)

{ M  
Map }

Prints the current text (*objectfile*) and core (*corefile*) address maps. This includes both the initial and modifiable maps for the *corefile* with an indication of which is currently active. An *address map* is a set of triples (b,e,f) that determine how *memory addresses* are translated into *file locations*. A triple consist of a *beginning address* in memory (b), an *ending address* in memory (e), and an *offset* value (f). When a *memory address* meets the following condition:

*beginning address* <= *memory address* < *ending address*

the *file location* is calculated using the formula:

*file location* = (*memory address* - *beginning address*) + *offset*

If the *memory address* doesn't satisfy the condition for any triple for the file, it is invalid.

To view the address map for *objectfiles* and *corefiles*, execute the following debugger command:

>M

Your display will look similar to this:

```

16: int y;
17: {
> 18:     *x = y;
19: }
File: test_prog.c Procedure: set_to Line: 18
>M
Object file (test_prog):
0x00001000 0x000023ac 0x00002000
0x40001000 0x400013f4 0x00004000
Core file (mycore):
Kernel: 0x7ffe6c60 0x7ffe6c9c 0x00000010
Exec: 0x7ffe6c1c 0x7ffe6c60 0x0000005c
Core: 0x7ffe6c0c 0x7ffe6c10 0x000000b0
Data: 0x40001000 0x40002000 0x000000c4
MMF: 0x7b00a000 0x7b00c000 0x000010d4
MMF: 0x7b00c000 0x7b018000 0x000030e4
MMF: 0x7b018000 0x7b031000 0x0000f0f4
MMF: 0x7b031000 0x7b033000 0x00028104
Stack: 0x7b033000 0x7b036000 0x0002a114
Registers: 0x7ffe6d60 0x7ffe6f68 0x0002d124
Core file (mycore): (inactive map)
00000000 0x01000000 00000000
00000000 0x01000000 00000000

```

4

Figure 4-4. A View of Object and Core Address Maps

## Miscellaneous Commands

You can see from looking at the address maps that they are divided up into three categories:

- Object file address map that consists of two triples created from information in the object file.
- Core file initial address map that consists of several triples created from information in the core file.
- Core file address map (labeled inactive in the figure) consists of two modifiable triples for use with the `Mc` command.

4

Each category has rows of addresses in it that are separated into three columns. The first column is the *beginning address*, the second column is the *ending address*, and the third column is the *offset* value. These addresses are used in the *file location* formula when the debugger accesses the object or core file instead of an executing process. The range of addresses in the object file category provide access to text and data information. The range of addresses in the default core file category provide access to version (`Kernel` and `Core`), `exec` area (`Exec`), data, stack, memory-mapped file (`MMF`) and register information. The `exec` area information is used to verify that the core file was generated by executing the object file. The `MMFs` are those memory-mapped files that are mapped private and are necessary in order for the debugger to handle core files where shared libraries are invoked. The modifiable core file triples (initially inactive) allow you to define an alternate address map to use with the core file. You can also modify the triples in the object file map.

---

**Read This** You should read this note before using the `Mc`, `Mt`, and `tM` commands.

While a file mapping different from the original one set by the debugger is active, debugger commands that translate symbolic names into addresses or use indirect addressing (such as examining variables by name, stack traces, etc.) will produce unexpected results.

---

## Mc

4

```
Mc [ expr [ ; expr [ ... ] ] ]
```

Sets the modifiable core (*corefile*) address map. The first zero to six map values are set to the *exprs* given. For example, executing this command:

```
Mc 0x00004000 0x00004223 0x00000040 0x00005000 0x00005400 0x00000040
```

changes the modifiable core-file address map from a map that may look like this:

```
Core file (mycore): (inactive map)
00000000 0x01000000 00000000
00000000 0x01000000 00000000
```

to one that looks like this:

```
Core file (mycore):
0x00004000 0x00004223 0x00000040
0x00005000 0x01005400 0x00000040
```

If less than six expressions are given, the remaining map parameters are left unchanged.

The `Mc` command also switches the active *corefile* mapping to the modifiable core-file map. The `tM` command can be used to toggle you back to the initial core-file map.

## Miscellaneous Commands

### Mt

```
Mt [ expr [ ; expr [ ... ] ] ]
```

Sets the text (*objectfile*) address map. The first zero to six map values are set to the *exprs* given. For example, executing this command:

```
Mc 0x00000010 0x00000400 0x00000020 0x00001020 0x00001100 0x00000020
```

changes the object-file address map from a map that may look like this:

```
4 Object file (test_prog):  
    00000000 0x00000674 0x00000040  
    0x00001000 0x000011b8 0x000006b4
```

to one that looks like this:

```
Object file (test_prog):  
    0x00000010 0x00000400 0x00000020  
    0x00001020 0x00001100 0x00000020
```

If less than six expressions are given, the remaining map parameters are left unchanged.

Note that it is a good idea to write down the original values before changing the object file map because the only way to restore them is by re-entering them with the **Mt** command.

### tM (toggle maps)

```
{ tM  
  toggle maps }
```

Toggles the address mapping of the *corefile* between the initial core-file map and the modifiable core-file mapping pair which the user can set with the **Mc** command.

The most likely alternate core-file map (using the *memory address* as a *file location*) is set up by the debugger as the default modifiable core-file mapping.

## C++ and the Symbolic Debugger

---

This chapter covers information that is specific to the use of the symbolic debugger for debugging C++ programs.

On Series 300/400 computers, there is C++ support in both the *xdb* and the *cdb* debugger programs. On Series 600/700/800 computers, you must use the *xdb* program. Note that this chapter only uses the *xdb* syntax. For a list of differences between *xdb* and *cdb*, read the appendix “Comparison between the *xdb* and *cdb* Symbolic Debuggers” found in this manual.

5

Topics covered in this chapter are as follows:

- Summary of Debugger Support for C++
- How the Debugger Deals with C++ Scopes
- C++ Expressions
- Displaying Static Data Members
- Listing Local Variables
- Listing Functions
- Viewing Functions with the Debugger
- Breakpoint Commands
- Handling Exceptions
- Debugging Parameterized Types
- Using Nested Classes
- Customizing Default Debugger Behavior
- Sample C++ Debugging Sessions



---

## Summary of Debugger Support for C++

The following table summarizes the abundant features the debugger provides to support the object oriented nature of C++.

**Table 5-1. Debugger Support for C++**

Feature	Description
Transparent Name Demangling	The debugger lets you debug using your actual C++ variable and function names as they were declared. There is no need to translate C++ names into the C names generated from them. This prevents confusion and possible error.
Overloaded Functions and Operators	When a debugger command involves an overloaded function, a menu of possible choices is displayed, allowing you to resolve the ambiguity by identifying the intended function. Breakpoints can be set at all overloaded functions with a given name using a single command. The same debugger commands that apply to an overloaded function also apply to an overloaded operator.
C++ Scope Rules	The debugger conforms to C++ scope by allowing access to identifiers either directly from within its scope or by means of the C++ scoping operator (::) from outside its scope.
C++ Data Types	The debugger provides support for C++ constant types, enumeration types, pointers to class members, reference types, and anonymous unions.
Member Functions	In addition to specifying single step, view, and breakpoint operations for member functions, you can also call a member function from the command line.

**Table 5-1. Debugger Support for C++ (continued)**

<b>Feature</b>	<b>Description</b>
Classes and Objects	Simple or extended versions of class information can be viewed. That is, you have the choice of whether to display inheritance members with the extended version. The function and data members of a class can be accessed via dot (.), arrow (->) and scope (::) operators. The data members of a class object can be examined and modified. Static data members of a class can also be accessed.
Class Commands	The debugger provides powerful commands which allow access to all members of a class. Breakpoints can be set at all member functions of a class by a single command.
Object Identification	In C++, a pointer to an object may point to its declared class or any derived class. Since it is not possible to determine the correct type of the object until run-time, the debugger supports automatic dynamic object identification. This can be a tremendous help in debugging object-oriented code.
Instance Breakpoint	A breakpoint can be set at a member function for a particular instance of a class. This reduces the number of breakpoints that are reached for member functions and can therefore lead to significantly improved productivity.

**Table 5-1. Debugger Support for C++ (continued)**

Feature	Description
Exception Handling	The C++ language provides <i>exception handling</i> for dealing with special conditions produced by the programmer as well as for dealing with execution of invalid programming operations (for example, dividing by zero). The debugger supports the handling of <b>throw</b> and <b>catch</b> exceptions.
Parameterized Types	In general, <i>class template names</i> can be used anywhere a class name is valid. Likewise, <i>function template names</i> can be used anywhere a function name is valid. The debugger provides support for referencing and setting breakpoints on these parameterized types.
Nested Classes	References to static members of an enclosing class and references to class names of enclosed classes are supported by the debugger.

5

---

## How the Debugger Deals with C++ Scopes

This section explains how the debugger handles C++ scopes. These scopes include class scopes and declaration-statement scopes.

### What Does Scope Mean

The term *scope* is the region of a program in which an *identifier* has meaning. An identifier is a sequence of characters that represent an entity such as a function or data object.

As an example of scope in the C language, the scope of a local variable having an identifier `xyz` is within the function where `xyz` is defined. On the other hand, the scope of a global variable `abc` includes all functions in the program which have not redefined `abc` themselves.

The current viewing location of the debugger may imply a certain scope in which a given identifier has a particular meaning. If the location is changed, this identifier's meaning may also change, or it may have no meaning at all. The debugger's scoping rules match the language of the source code which is being debugged with a few intuitive extensions to provide added flexibility during debugging. The syntax also provides the ability to designate the scope where it may be outside the current viewing location or where it may be ambiguous.

There are two scopes that require special consideration when dealing with the symbolic debugger and the C++ language:

- Class
- Declaration Statement

These scopes are covered in subsequent sections.

### Class Scope

Because of its class data type, C++ has a unique kind of scope called the "class scope." The class scope consists of all the member functions, variables, constants, and enumerators included within the class definition. For example,

```

class employee {
    static int employee_count;
    char name[50];
    const int employee_number;
public:
    employee(char *, int);
    enum employment_status { parttime, fulltime };
    void print_name();
};

class professor: public employee {
public:
    int department_number;
    professor(char *, int);
    int salary();
};

```

In this example, `employee_count`, `name`, `employee_number`, `employee` (the constructor), `parttime`, `fulltime`, and `print_name` are names within the scope of class `employee`. The class `professor` not only has `department_number`, `professor` (the constructor) and `salary` within its scope, but also all members of its base class `employee`.

Depending upon the circumstances, members within a class's scope may be accessed in a variety of ways. If the current viewing location is within a member function of a class, members in that scope may be identified by a simple name. For example, if the current location is within the member function `salary`, `department_number` may be referred to as simply `department_number` without qualification since the class object is implied. It may also be called:

```

this->department_number
professor::department_number
this->professor::department_number

```

In general, the debugger has the same restrictions as the C++ language itself. If the current location is not within a member function of `professor`, `department_number` may not be referred to as

```

professor::department_number

```

## 5-6 C++ and the Symbolic Debugger

However, you may use `prof1.department_number` if `prof1` is an object of type `professor`. One exception to these restrictions is that the debugger ignores access declarations. This means that it is possible to use

```
employee::employee_count
```

outside a member function of `employee` even though it is a private member.

Here is a summary of the principle ways class members may be referenced:

- If the current viewing location is not within a member function of the class, these are valid:
  - *object.member*
  - *class::member* if the member is static
- If the current viewing location is within a static member function, these are valid:
  - *member* if the member is static
  - *class::member* if the member is static
- If the current viewing location is within a non-static member function, these are valid:
  - *member*
  - *class::member*
  - *this->member*
  - *this->class::member*

5

## Declaration Statement Scope

Since variables can be declared anywhere in a program, certain scoping restrictions are implied by the location of any particular declaration. For example:

```
class A {
public:
    int a;
    A() {a = 0;}
};
```

```
main()
{
    int i;
    i = 4;
    A x;
    x.a = 3;
    return 0;
}
```

5

shows `i` and `x` at the same scope level; however, `i` is not at the same scope level as `x` to the debugger. The debugger interprets the scope levels as if `main` were instead written as follows:

```
main()
{
    int i;
    i = 4;
    {
        A x;
        x.a = 3;
        return 0;
    }
}
```

Thus, if an attempt is made to display `x` at the line:

```
i = 4;
```

an appropriate error message is given.

## Setting Breakpoints at the End of a Scope

When you set a breakpoint at the end of a scope, there may be some variables in that scope that are not accessible at the breakpoint. This is because implicit scopes are sometimes created for statements like the `for` statement. Variables inside the implicit scope will not be accessible outside the implicit scope, for example at the end of the enclosing scope.

When you set a breakpoint, the debugger warns you if any variables in implicit scopes are not accessible at that breakpoint. For example:

```
void main(){
int i;
    // An implicit scope starts here because of the for statement.
    for (int x=0;x<5;x++) { ... }
    int z; // z is inside the implicit scope.
    ...
    // The implicit scope ends here, before the closing }
} // end of function scope; setting a breakpoint here,
// you can't see z in xdb. The debugger warns you of this
// when you set the breakpoint.
```

5



---

## C++ Expressions

The debugger can evaluate a rich subset of valid C++ expressions. This section discusses which types of expressions are supported and which are not. It also lists syntax extensions which enhance flexibility.

### Variables

This syntax is used for C++ variables in an expression:

`[ [ [ class_name ] :: ] function_name : [ depth : ] ] [ class_name :: ] variable`

The *variable* name may be qualified by a *function\_name*, *depth* on the stack, and/or a *class\_name* to uniquely identify which variable and instance of that variable is desired. For example:

5

- |                               |  |
|-------------------------------|--|
| <code>p rate</code>           | Print the value of <code>rate</code> in the current function.  |
| <code>p employee::name</code> | This may be used in several cases: <ul style="list-style-type: none"><li>■ Class <code>employee</code> has a static member called <code>name</code>. This expression is then valid anywhere class <code>employee</code> is visible.</li><li>■ Class <code>employee</code> has a member function called <code>name</code> and this expression will evaluate to the address of that function. An error message will be issued if the function is pure virtual, inlined, or declared but not defined.</li><li>■ The current viewing location is within a member function of class <code>employee</code> or a member function of a class which uses <code>employee</code> as a base class. It is valid for <code>name</code> to be any member type, static or not.</li></ul> |
| <code>p link:index</code>     | Find the most recent occurrence on the stack of a function called <code>link</code> and print its local variable called <code>index</code> .   |

`p sort:4:pointer`

Find an invocation of a function called `sort` on the stack at depth 4 and print its local variable called `pointer`. (An explicit depth is useful when there are recursive calls.)

`p ::merge:top`

Find the most recent occurrence on the stack of a global function called `merge` and print its local variable called `top`.

`p matrix::invert:2:vector::length`

Find an invocation of class `matrix`'s member function called `invert` at depth 2 on the stack and print a member of class `vector` called `length`. (Presumably, `vector` is a base class of `matrix`.)

5

## Global Variables

To use global variables in expressions, the `::` operator may be used just as in the C++ language itself. Note that the child process must be active for references to *variable* to be active. The syntax is:

`::variable`

Given this program:

```
int i = 3;

main()
{
    int i = 4;
    return 0;
}
```

if the debugger's current viewing location is at the return statement, the global variable `i` may be used in an expression by referring to it as `::i`. For example:

```
p ::i - 1
```

If there had been no ambiguity between a local and global variable of the same name (e.g. the local variable was called `j` instead of `i`), this expression would have achieved the same result:

```
p i - 1
```

## Reference Types

If the debugger supports a particular variable type in an expression, then a reference to the same type is also supported.

```
main()
{
    int i;
    int &j = i;
    i = 4;
    return 0;
}
```

In the program shown above, it is permissible to use `j` in an expression. For example:

```
p j + 3
```

## Function Calls

Calls to functions may be included in expressions, with certain restrictions on parameter types and return types.

```

class account {
    long number;
    static short count;
public:
    account(long acct) {number = acct; count++;};
    long get_number();
    static short get_count();
    long operator==(long);
};

short account::count = 0;

long account::get_number() { return number; }

short account::get_count() { return count; }

long account::operator==(long index)
{
    return number == index ? 0 : (number > index ? 1 : -1);
}

unsigned long total(char *ptr)
{
    unsigned long sum = 0;
    char c;
    while (c = *ptr++)
        sum += (unsigned long) (c - '0');
    return sum;
}

main()
{
    account old_account = 10014;
    account *pointer = &old_account;
    return 0;
}

```

5

For the program shown above, all of the following are valid command line function calls:

- `p old_account.get_number()`
- `p pointer->get_number()`
- `p old_account.get_count()`
- `p old_account.account::get_count()`
- `p old_account::get_count()`
- `p old_account.operator==(10000)`
- `p total("123")`

The following types of functions may not be called from the command line (an appropriate error message will be issued if you attempt to do so):

- A function returning a class object.
- A function returning a pointer to a member function.
- A function parameter which is a class object.
- A function parameter which is a pointer to a member function.
- A call where a difference in type between formal and actual argument requires implicitly calling a constructor or conversion operator.
- Implicit calls to overloaded operators.
- Implicit calls to constructors.
- Implicit calls to conversion operators.
- Calls to functions which are pure virtual.
- Calls to functions which have been declared but not defined.
- Calls to functions which have been inlined and no static copy of the function exists. If a function is declared to be inlined, a static copy will be created only if the `+d` compile option is used, or if the address of the function is required somewhere in the code (e.g. initializing a pointer to point to it), or if the code in the function is deemed to be too complex to be inlined.
- Calls involving expressions which include dereferencing a pointer to a member function (e.g. `(object.*pointer)(1, 2, 3)`).

If a function has default parameters, those parameters must be explicitly stated. If the required number of parameters is not given to the command-line call, you will be given a warning message and an opportunity to cancel the call, except in cases where optional parameters are declared with *ellipsis* (...).

For cases where an overloaded function is to be called during the course of expression evaluation, the user will be presented with a menu to enable disambiguation. For example:

```
char abc(short s) { return (char) s; }
char abc(long l) { return (char) l; }

main()
{
    return 0;
}
```

5

If this expression is executed:

```
p abc(1)
```

the debugger will display:

```
abc
~
Overloaded function; please choose one:
1 char ::abc(short);
2 char ::abc(long);
function number?
```

The user can then respond with 1 or 2 to indicate which one is desired.

## Operators

Operators include such things as +, -, >>, [], ->, =, \*&, etc. In general, an expression may use any operators if its evaluation does not require the implicit invocation of such things as constructors or conversion operators. For example, in the following program:

```
class A {
public:
    int a;
    A() {a = 0;};
    operator int();
};

A::operator int()
{
    return a;
}

class B {
public:
    int b;
    B(int i) {b = i;};
    B(A &);
};

B::B(A &x)
{
    b = x.a;
}

main()
{
    int m = 7;
    A a1;
    B b1 = 2;
    return 0;
}
```

5

If the debugger is stopped on `main`'s return statement, the following expressions are allowed:

```
p m + 30
p a1
p b1.b = 4
```

However, although the following expressions would be allowed in the C++ program, they are not supported in the debugger because of the implicit calls to a conversion operator or constructor:

```
p m = a1;
p b1 = a1;
```

Similarly, overloaded operators may not be implicitly called:

```
class A {
    long a[2];
public:
    A(long i) {a[0] = -i; a[1] = i;};
    long operator[](long);
};

long A::operator[](long i)
{
    return a[i != 0];
}

main()
{
    A x = 10;
    long i = x[4];
    return 0;
}
```

From `main`, it is not possible to execute the following debugger expression:

```
p x[100]
```

However, the desired effect can be achieved in this case with:

```
p x.operator[](100)
```



## Class Objects

The operations that are allowed on class objects are:

- print the type of the object
- print the members of the object in a structured format
- take the address of the object with the `&` operator
- take the size of the object with the `sizeof` or `$sizeof` operators

No other operations may be performed on the object as a whole although more extensive operations are allowed on individual members of the class object.

The following program will be used in the next two subsections.

```
5  class A {
      char a;
  public:
      A() {a = 'a';}
  };

  class B : public virtual A {
      char b;
  public:
      B() {b = 'b';}
  };

  class C : public virtual A {
      char c;
  public:
      C() {c = 'c';}
  };

  class D : public B, public C {
      char d;
  public:
      D() {d = 'd';}
  };
```

```

main()
{
    D object;
    return 0;
}

```

### Displaying Type Information for an Object

The `t` and `T` format specifiers are used for printing the type of an object. The syntax is:

```
p expression\t
```

```
p expression\T
```

where *expression* reduces to an object. The difference between the two formats is that the `t` displays information only for the immediate class whereas `T` displays information for the class and all its base classes. If the following command is executed for the example program given above:

```
p object\t
```

the result will be:

```

class D: public B, public C {
private:
    char d;
public:
    inline D(A *);
    inline D(A *, const D &);
} object

```

Notice that only the type information for class `D` is displayed and not for classes `A`, `B`, and `C`. (The two special constructors are automatically created by the compiler and can be ignored for now.)

To get type information which includes base classes, execute:

```
p object\T
```

This will print:

```
class D: public B, public C {

    class B: public virtual A {

        class A {
        private:
            char a;
        public:
            inline A();
        }

    private:
        char b;
    public:
        inline B(A *);
        inline B(A *, const B &);
    }

    class C: public virtual A {
    private:
        char c;
    public:
        inline C(A *);
        inline C(A *, const C &);
    }

private:
    char d;
public:
    inline D(A *);
    inline D(A *, const D &);
} object
```

5

Notice that the type information for class A is only printed once although it is inherited twice (by classes B and C). It is possible to control the default behavior of whether or not this base class information will be duplicated where appropriate. One bit of a special variable called `$cplusplus` controls this. If bit 0 of this variable is set, the base class information will be printed at each point where it has been inherited. In other words, to enable this feature execute:

```
p $cplusplus |= 1
```

If bit 0 of this variable is not set, type information will only be printed once as in the display shown above. To request this behavior execute:

```
p $cplusplus &= ~1
```

The `$cplusplus` variable also contains bits to control the default behavior of other C++ features, so it is important to affect only bit 0 when using it to modify the behavior in printing base class information. By default, bit 0 of `$cplusplus` is cleared.

5

With bit 0 of `$cplusplus` set, executing this same command:

```
p object\T
```

will result in:

```
class D: public B, public C {

    class B: public virtual A {

        class A {
        private:
            char a;
        public:
            inline A();
        }

    private:
        char b;
    public:
        inline B(A *);
        inline B(A *, const B &);
```

```

}
class C: public virtual A {
    class A {
    private:
        char a;
    public:
        inline A();
    }

    private:
        char c;
    public:
        inline C(A *);
        inline C(A *, const C &);
}

```

5

```

private:
    char d;
public:
    inline D(A *);
    inline D(A *, const D &);
} object

```

The `t` and `T` formats show all type information about an object including access declarations (public, protected, private), access modifications, inheritance information, friends, data members, and member functions with parameter and return types.

## Displaying the Contents of an Object

The `k` and `K` format specifiers are used in printing the contents of an object. The syntax is:

- `p expression`
- `p expression\k`
- `p expression\K`

where *expression* reduces to an object. If there is no format specifier, the meaning is the same as if the `k` specifier had been given. The difference between `k` and `K` formats is that the `k` displays information only for the immediate class whereas `K` displays information for the class and all its base classes. If the following command is executed for the example program given above:

```
p object\k
```

the result will be:

```
object = class D: public B, public C {  
private:  
    d = 'd';  
}
```

Notice that only the information for class D is displayed and not for classes A, B, and C.

To get information which includes base classes, execute:

```
p object\K
```

This will print:

```
object = class D: public B, public C {  
  
    class B: public virtual A {  
  
        class A {  
            private:  
                a = 'a';  
        }  
  
        private:  
            b = 'b';  
    }  
  
    class C: public virtual A {  
        private:  
            c = 'c';  
    }  
  
    private:  
        d = 'd';  
}
```

5

Notice that data for class A is only printed once although it is inherited twice (by classes B and C). As with the `t` and `T` formats, it is possible to control whether or not this base class information will be duplicated where appropriate. One bit of a special variable called `$cplusplus` controls this. If bit 0 of this variable is set, the base class information will be printed at each point where it has been inherited. In other words, to enable this feature execute:

```
p $cplusplus |= 1
```

If bit 0 of this variable is not set, information will only be printed once as in the display shown above. To request this behavior execute:

```
p $cplusplus &= ~1
```

The \$cplusplus variable also contains bits to control the default behavior of other C++ features, so it is important to affect only bit 0 when using it to modify the behavior in printing base class information. By default, bit 0 of \$cplusplus is cleared.

With bit 0 of \$cplusplus set, executing this same command:

```
p object\K
```

will result in:

```
object = class D: public B, public C {  
  
    class B: public virtual A {  
  
        class A {  
        private:  
            a = 'a';  
        }  
  
        private:  
            b = 'b';  
    }  
  
    class C: public virtual A {  
  
        class A {  
        private:  
            a = 'a';  
        }  
  
        private:  
            c = 'c';  
    }  
  
    private:  
        d = 'd';  
}
```

5



Duplication of base class information is avoided with bit 0 of `$cplusplus` cleared only if this really does constitute duplication of information. In the example shown above, there is really only one data member called `a` in `object`. However, if we change the program slightly so that the two public virtual inheritances are private instead, we will now have two distinct data members called `a`. Both of them will always be printed with the `K` or `T` format specifier independent of the value of bit 0 in `$cplusplus`.

The `k` and `K` formats shows data about an object including access declarations (public, protected, private), inheritance information, and data members. Information about friends, access modifications, and member functions is not included.

### Object Identification

5

It is important to consider the case where the expression which evaluates to an object is a dereference of a pointer to an object. C++ allows a pointer to a base class type to point to an object of a derived type. For example:

```
p *object_pointer\t
p *object_pointer\K
```

If the pointer does indeed point to a derived object, the information for that object will be printed instead of for the class type which the pointer type implies. This feature is called object identification. This capability is a tremendous help when debugging object-oriented C++ code. Since a C++ pointer to an object may point to its declared class or any derived class, it is not possible to determine its correct type until run-time.

```
class A {
    char a;
public:
    A() {a = 'a';}
    virtual int f() { return 1; }
};
```

```

class B : public A {
    char b;
public:
    B() {b = 'b';}
    int f() { return 2; }
};

main()
{
    A a1;
    B b1;
    A *ptr;
    ptr = &a1;
    ptr = &b1;
    return 0;
}

```

5

In the program shown above, if we execute the following command just after `ptr` has been assigned to point to `a1`:

```
p *ptr\k
```

the result will be something like:

```

0x68ff33d8  class A {
private:
    a = 'a';
    __vptr = 0x40000018;
}

```

If we step past the next line which assigns `ptr` to point to `b1` and execute the same command, the result will be something like:

```

0x68ff33cc  class B: public A {
private:
    b = 'b';
}

```

## Class Members

### Data Members

Data members of a class object may be viewed and, when appropriate, modified.

```
class A {  
public:  
    const char c;  
    static short s;  
    long l;  
    enum { e1, e2, e3 } e;  
    A(char x, long y) : c(x) {l = y; s++; e = e1;}  
};
```

```
short A::s = 0;
```

```
main()  
{  
    A object('a', 10);  
    return 0;  
}
```

For this program, the following data viewing commands may be executed:

```
p object.c  
p object.s  
p object.l  
p object.e  
p A::s  
p A::e2
```

As for modifying object members, the following commands are valid:

```
p object.c = 'b'  
p object.s = 2  
p object.l = -12  
p object.e = A::e3  
p A::s = 100
```

Of course, an object's data members may also be used in more complex expressions. For example:

```
p A::s = object.l * object.c + 14
```

A member may also be qualified by a class:

```
p object.A::l
```

This will be useful in cases where, because of class inheritance, there is more than one member with the same name.

When the current viewing location is within a member function, the object is implied (unless the function is static) and data members may be referenced just as they are in the C++ language itself.

```
class A {
    long i;
    static long j;
public:
    long get_i();
    static long get_j();
    A(long a, long b) {i = a; j = b;}
};

long A::get_i()
{
    return i;    // viewing location 1
}
long A::get_j()
{
    return j;    // viewing location 2
}

main()
{
    A object(1, 2);
    long m1 = object.get_i();
    long m2 = object.get_j();
    return 0;
}
```

5

In this program, if the debugger is stopped at the line marked `// viewing location 1`, all data member expressions that are valid are:

```
i
A::i
this->i
this->A::i
j
A::j
this->j
this->A::j
```

- 5 If the debugger is stopped at viewing location 2, only `j` and `A::j` are valid (because `get_j` is a static function).

### Member Functions

The principle operations that may be done on a member function of an object are:

- call it from the command-line
- use its address as a pointer to a member function within a class (usually for assignment to a pointer variable)
- determine its address
- print its type

If the name of a member function is used in an expression and a parameter list is included, the function is called. Calling member functions is described in the section called “Function Calls.” Pointers to member functions are covered in the section called “Member Pointers.”

If the name of a member function is used in an expression *without* giving an argument list, the address of the function is used. For example:

```
class A {
public:
    int f();
    short g();
};

int A::f() { return 1; }
short A::g() { return 2; }

main()
{
    A object;
    A *ptr = &object;
    return 0;
}
```

Executing the command:

```
p A::f
```

will print the address of A::f. These commands are also valid:

```
p object.g
p object.A::f
p ptr->f
p ptr->A::g
```

A function's type may be printed by using either the `t` or `T` formats. This is true for both global and member functions. Given this program:

```
class A {
    class B {
    public:
        long B::bf() { return 1; }
    } b;
    class C {
    public:
        long (B::*bp)();
    } c;
public:
    A() { c.bp = &B::bf; }
    long f(long (B::*C::*a)(), long);
};

long A::f(long (B::*C::*p)(), long l)
{
    long (B::*bp)() = c.*p;
    return (b.*bp)() + l;
}

main()
{
    A x;
    long i = x.f(&C::bp, 4);
    return 0;
}
```

5

If this command is executed:

```
p x.f\t
```

the result will be:

```
long A::f(long (B::*C::*)(), long)
```

## Object Pointers

If a pointer to a class object is dereferenced, the debugger will attempt to identify the type of object that it is pointing to. This feature is called *object identification* and is described more fully in under “Object Identification” in the section called “Class Objects”.

If a base class pointer is assigned to point to a derived object, the debugger automatically takes care of all necessary pointer adjustments. For example:

```
class A {
    char a;
public:
    A() {a = 'a';}
    virtual int f() { return 1; }
};

class B : public A {
    char b;
public:
    B() {b = 'b';}
    int f() { return 2; }
};

main()
{
    A a1;
    B b1;
    A *a_ptr = &a1;
    B *b_ptr = &b1;
    return 0;
}
```

5



The debugger will make appropriate adjustments for the following command:

```
p a_ptr = &b1
```

It should be noted that when requesting the type of a class pointer, its declared type will be printed instead of the type of the object that the pointer is pointing to. Even after executing the previous command, the following command:

```
p a_ptr\t
```

will print:

```
A *a_ptr
```

## Member Pointers

Pointers to class data members and to member functions are supported for both viewing and modification.

```
class A {
public:
    long l1, l2;
    long f1();
    long f2();
};

long A::f1() { return 1; }
long A::f2() { return 2; }

main()
{
    A object, *object_ptr = &object;
    long A::*p = &A::l1;
    long (A::*pf)() = &A::f1;
    return 0;
}
```

5

Given the program above, the following viewing commands are accepted by the debugger:

```
p p
p pf
p object.*p
p object.*pf
p object_ptr->*p
p object_ptr->*pf
```

When the value of a member pointer is printed, the actual class name and member are displayed. For the command:

```
p p
```

the debugger will respond with:

```
p = &A::l1
```

Similarly, for:

```
p pf
```

we will see:

```
pf = &A::f1
```

If a pointer has a garbage value for some reason, the debugger will say:

```
p = <uninitialized>
```

As for modifying pointers to members, the following are two examples of commands that are supported by the debugger:

5

- p p = &A::l2
- p pf = &A::f2

## Casts

Classes can be used in casts. The primary use of this feature is when an address of a class object is known and the user wants the contents of memory beginning at that address printed out in the object's class type format.

Consider the following example program:

```
class A {
public:
    long a, b, c;
};

main()
{
    A object;
    return 0;
}
```

Let's say that during the course of debugging, we know that an object of class type A can be found at memory location 0x5000. To print the object out in a structured format, we can use a cast:

```
p *((class A *) 0x5000)
```

This may give us something like:

```
0x00005000 class A {  
public:  
    a = 10;  
    b = 20;  
    c = 30;  
}
```

Note that the keyword `class` is required. As a shorthand way of doing the same thing, the command may be given as:

```
p (class A) 0x5000
```

In other words, the argument to the cast is an address that is treated as if it were the location of an object of class type A. In a C++ program, such an expression would mean “convert the number 0x5000 to a class A object”, but since such conversions are not supported in the debugger, this notation is given an alternate meaning.

Another use of casts is to display a class definition even when an object of that type does not exist. For example, this command:

```
p (class A) 0\t
```

will print:

```
class A {  
public:  
    long a;  
    long b;  
    long c;  
} <unnamed>
```

The T format specifier may be used to display base class information if inheritance is involved. However, an even simpler way of doing this is with either of these two commands:

5

```
p class\t
```

```
p class\T
```

Here are two examples of commands that deal with members of class objects that are only known by address:

```
p ((class A) 0x5000).a  
p ((class A *) 0x5000)->b = 75
```

Casts must include a `class` keyword, the name of the class, and optionally a `*` to indicate a pointer type. More than one pointer level is not allowed (e.g. `(class A **)`). The keyword `struct` or `union` may be substituted in place of `class` regardless of how the type was actually declared. Because such type definitions are usually only referred to by their name without the keyword in C++ programs, this allows some flexibility if the exact declaration type cannot be immediately remembered.

The argument to a cast may be any expression which evaluates to a number (address). However, care must be taken in certain cases or the results will be other than what was expected. For example:

```
class A {
public:
    int a;
};

class B {
public:
    int b;
};

class C : public A, public B {
public:
    int c;
};

main()
{
    C object;
    B *object_ptr = &object;
    object.a = 1;
    object.b = 2;
    object.c = 3;
    return 0;
}
```

If the debugger is stopped at the return statement and you wish to print the value of `*object_ptr`, you would first be inclined to execute this command:

```
p (class C) object_ptr
```

This command shows a B type pointer pointing to a C type object.

However, instead of getting the expected;

```
0x68ff33c4 class C: public A, public B {
public:
    c = 3;
}
```

it gives:

```
0x68ff33c8 class C: public A, public B {
public:
    c = 0;
}
```

This is because casts do not do any adjustments when dealing with pointers to class objects. Such a debugger capability is not supported.

5

## Anonymous Unions

The debugger fully supports anonymous unions. A member of such a union may be referenced in the same way it is in a C++ program.

```
class A {
public:
    union {
        long l;
        short s;
    };
};

main()
{
    A object;
    object.l = 12345678;
    union {
        char c;
        double d;
    };
    return 0;
}
```

With the debugger stopped on the return statement in the program above, these commands may be executed:

```
p object.l  
p c = 'a'
```

If the class object is printed with the command:

```
p object
```

we will get:

```
object = class A {  
public:  
    union {  
        l = 12345678;  
        s = 188;  
    };  
}
```

5



---

## Displaying Static Data Members

The debugger provides the capability of printing the values of all static data members of a particular class. The syntax for this command is:

```
p class_name::
```

Consider this example program:

```
class A {  
public:  
    static long l;  
};
```

```
long A::l = 1;
```

```
class B: public A {  
public:  
    static long m;  
};
```

```
long B::m = 2;
```

```
main()  
{  
    B object;  
    return 0;  
}
```

5

If the location where the debugger is stopped is the return statement, executing this debugger command:

```
p A::
```

displays:

```
A::l          = 1
```

Likewise, this command:

```
p B::
```

will result in:

```
B::m          = 2
```

```
A::l          = 1
```

---

## Listing Local Variables

Extensions to the `l` command are provided for listing local variables of C++ functions. The syntax is:

```
l [[ [ class_name ] :: ] function_name [ : depth ] ]
```

All parameters and local variables of the specified *function\_name*, along with their current values, are printed. For example:

5

```
l                List local variables of current function
l read_next      List local variables of the most recent invocation of a
                  function called read_next which is on the stack
l sort:3         List local variables of a function called sort which is
                  at a depth of 3 on the stack
l ::rotate       List local variables of the most recent invocation of a
                  global function called rotate which is on the stack
l list::delete:7 list local variables of a function called delete that is a
                  member of a class called list and is found at a depth
                  of 7 on the stack
```

---

## Listing Functions

The debugger provides extensions to the `lp` command for listing functions and an `lo` command for listing overloaded functions.

### Listing Functions

The `lp` debugger command is used to list all functions whose names start with the string that is given as its argument. This command's syntax is as follows:

```
lp [ [ class_name ] :: ] [ string ]
```

A string can be qualified with a *class\_name* which means that only member functions of this class are of interest. For example:

<code>lp</code>	List all functions
<code>lp A::</code>	List all member functions of class A
<code>lp ::</code>	List all global functions
<code>lp ::m</code>	List all global functions which begin with the letter "m"
<code>lp B::set_</code>	List all member functions of class B which begin with the string "set_"

5

## Listing Overloaded Functions

The `lo` debugger command is used to list all overloaded functions whose names start with the string that is given as its argument. It works exactly like the `lp` command except that a function must be overloaded to be listed. This command's syntax is as follows:

```
lo [ [ class_name ] :: ] [ string ]
```

A string can be qualified with a *class\_name* which means that only member functions of this class are of interest. For example:

<code>lo</code>	List all overloaded functions
<code>lo A::</code>	List all overloaded member functions of class <b>A</b>
<code>lo ::</code>	List all overloaded global functions
<code>lo ::m</code>	List all overloaded global functions which begin with the letter "m"
<code>lo B::set_</code>	List all overloaded member functions of class <b>B</b> which begin with the string "set_"

5

---

## Viewing Functions with the Debugger

The debugger extends the `v` command (view) to qualify a function name with a class name. Its syntax is as follows:

```
v [ [ class_name ] :: ] function_name [ : line_number ]  
                                [ # label_name ]
```

### Example

```
1: class decimal {  
2:     char number[50];  
3: public:  
4:     decimal(char *);  
5:     void increment();  
6: };  
7:  
8: decimal::decimal(char *s)  
9: {  
10:     char *p1 = number, *p2 = s;  
11:     while (*p1++ = *p2++);  
12: }  
13:  
14: void decimal::increment()  
15: {  
16:     char *p1 = number;  
17:     while (*p1++);  
18:     char *p2 = --p1;  
19:     while (--p1 >= number) {  
20:         if (++*p1 <= '9')  
21:             return;  
22:         *p1 = '0';  
23:     }  
24:     shift: while (--p2 > number)  
25:         *(p2 + 1) = *p2;  
26:     *p2 = '1';  
27:     return;  
28: }  
29:  
30: void increment(long &i)
```

5

```
31: {
32:     addone: i++;
33:     return;
34: }
35:
36: long decrement(short &s)
37: {
38:     s--;
39:     return (long) s;
40: }
41:
42: long decrement(long &i)
43: {
44:     i--;
45:     return i;
46: }
47:
48: main()
49: {
50:     decimal x = "999";
51:     long i = 3;
52:     x.increment();
53:     increment(i);
54:     return 0;
55: }
```

Executing the following `v` commands on the program shown above will result in the current location becoming the indicated line number:

Command	Line Number
<code>v increment</code>	32
<code>v decimal::increment</code>	16
<code>v ::increment</code>	32
<code>v decimal::increment:26</code>	26
<code>v ::increment:33</code>	33

Command	Line Number
<code>v increment:32</code>	32
<code>v decimal::increment#shift</code>	24
<code>v ::increment#addone</code>	32
<code>v increment#addone</code>	32

In some cases, the interpretation of the `v` command argument will depend upon the current viewing location. For example:

```
v increment
```

will refer to the member function of the class `decimal` if the current location is in a member function of `decimal`; otherwise, it will refer to the global function `increment`.

If an overloaded function is the argument of a `v` command, the debugger will present a menu to allow you to disambiguate the reference. For example, for the command:

```
v decrement
```

the debugger will respond with:

```
Overloaded function; please choose one:
1 long ::decrement(short &);
2 long ::decrement(long &);
function number?
```

You can then respond with either 1 or 2 to indicate which one is desired.



---

## Breakpoint Commands

Breakpoint commands specifically for C++ are provided as well as extensions to other breakpoint capabilities to handle unique C++ functionality. These include:

- Extensions to the ability to set a breakpoint on a particular function.
- The ability to set a breakpoint on overloaded functions.
- The ability to set a breakpoint on all member functions of a class.
- The ability to set a breakpoint on one or all member functions of a particular class instance.

### Setting a Breakpoint on a Function

5

The debugger breakpoint command **b** can be used to set a breakpoint on a function. The syntax for doing this is as follows:

```
b [ [ class_name ] :: ] function_name [ \count ] [ { commands } ]
```

where *function\_name* is the name of the function where the breakpoint is being set.

It may be optionally qualified by a *class\_name* to indicate that it is a member function of the designated class. A *function\_name* prefixed by only the `::` operator indicates that the breakpoint is to be set on a global function of that name.

For example, this command:

```
b print
```

will set a breakpoint on a function called **print**. In this example:

```
b ::print
```

a breakpoint will be set on a global function called **print**, and for this one:

```
b A::print
```

the debugger will set a breakpoint at the beginning of a function called **print** which is a member of class **A**.

Should there be more than one class with the given name because of identically named local classes, the debugger will take scoping, based on the current viewing location, into consideration to resolve the ambiguity.

If the name is an overloaded function, the debugger will list all of the possible functions with each preceded by a number to allow the selection of a breakpoint. For example, if the following breakpoint command is executed:

```
b print
```

information similar to the following may be displayed:

```
Overload function; please choose one:  
1 long ::print(short)  
2 long ::print(long)  
3 long ::print(float)  
4 long ::print(complex)  
function number?
```

5

If number 2 is selected, the debugger will set a breakpoint at the `print(long)` function.

## Setting a Breakpoint on Overloaded Functions

The debugger breakpoint command `bpo` can be used to set breakpoints on overloaded functions. The syntax for doing this is as follows:

```
bpo [ [ class_name ] :: ] function_name [ { commands } ]
```

The breakpoint will be set at *function\_name*. A *class\_name* may be included to indicate that a breakpoint is to be set only on the designated functions of a particular class. A *function\_name* prefixed by only the `::` operator indicates that the breakpoint is to be set on global functions of that name. A set of commands to be executed when the breakpoint is hit may be included. For more information on breakpoint command lists, see the section “Breakpoint Commands” found in the chapter “HP Symbolic Debugger Commands.”

5

If the following breakpoint command is executed:

```
bpo print
```

all functions named `print`, both global and member, would have a breakpoint.

The following command:

```
bpo ::print
```

sets a breakpoint at the beginning of every overloaded function with the name `print` that is of global scope (that is, not a member function of a class).

This command:

```
bpo A::print
```

sets a breakpoint at the beginning of every overloaded function named `print` that is a member of class `A`. The debugger will take scoping into consideration when local classes are involved and, based on viewing location, will correctly resolve any ambiguity should there be more than one class with the given name.

## Setting a Breakpoint at all Member Functions of a Class

The debugger breakpoint command `bpc` can be used to set breakpoints on all member functions of a class. The syntax for doing this is as follows:

```
bpc  $\left[ \begin{array}{c} -c \\ -C \end{array} \right] class\_name$ 
```

If *class\_name* is the name of a local class and there is more than one class with this name, the debugger will resolve the ambiguity based on the scoping implied by the current viewing location.

If `-c` is given, breakpoints will be set only on member functions of the specified class and not of any base classes. If `-C` is given, breakpoints will also be set on member functions of base classes. The default behavior when neither `-c` or `-C` is given can be configured by setting or clearing a particular bit of a special variable called `$cplusplus`. If bit 1 of `$cplusplus` is cleared, the `bpc` command will act as if the `-c` option were given. To enable this default behavior execute:

```
p $cplusplus &= ~2
```

If bit 1 of `$cplusplus` is set, the `bpc` command will act as if the `-C` option were given. To enable this default behavior, execute:

```
p $cplusplus |= 2
```

The `$cplusplus` variable also contains bits to control behavior of other C++ features, so it is important to affect only bit 1 when using it to modify the behavior of the `bpc` command. By default, bit 1 of `$cplusplus` is cleared.

When the `bpc` command sets breakpoints on member functions of base classes, the debugger will indicate this when listing the breakpoint. For example:

```
1: Active   class functions: myclass and base classes
```

## Setting an Instance Breakpoint

Sometimes it is desirable to set a breakpoint on one or more member functions of a particular class but have that breakpoint recognized only when the member function is executed for a particular class instance.

The debugger provides a single command to set such a breakpoint which is called an *instance* breakpoint. There is the flexibility to set an instance breakpoint on a particular member function or on all member functions. If all member functions are chosen, there is the added capability to designate only those functions which are members of the instance's immediate class or are members of the immediate class and all its base classes.

If the instance breakpoint is to be for a particular member function, the syntax is:

5

```
bi instance_expression.member_function [\count] [{commands}]
```

or:

```
bi instance_expression_pointer->member_function [\count] [{commands}]
```

As implied by the names, *instance\_expression* must reduce to an instance and *instance\_expression\_pointer* must reduce to a pointer to an instance. An optional count and/or commands may be included with the breakpoint. Please refer to the section “Breakpoint Commands” found in the chapter “HP Symbolic Debugger Commands” for further explanation of these.

When this type of instance breakpoint is listed, it will be similar to this example:

```
1: count: 1 Active instance function (class myclass): object.func
```

This includes the count, the class to which the member function belongs, and the expression used to specify the instance and member function just as it was given by the user.

If the instance breakpoint is to be for all member functions of an instance, the syntax is:

```
bi [ -c ] instance_expression [ { commands } ]
```

Once again, the *instance\_expression* must reduce to an instance. In this case, it is not possible to specify a count with the breakpoint, but commands may be given which will be executed when the breakpoint is hit. Please refer to section “Breakpoint Commands” found in the chapter “HP Symbolic Debugger Commands” for more information on breakpoint command lists.

If `-c` is given, breakpoints will be set only on member functions of the instance’s class and not of any base classes. If `-C` is given, breakpoints will also be set on member functions of base classes. The default behavior when neither `-c` or `-C` is given can be configured by the user by setting or clearing a particular bit of a special variable called `$cplusplus`. If bit 2 of `$cplusplus` is cleared, this `bi` command will act as if the `-c` option were given. To enable this default behavior execute:

```
p $cplusplus &= ~4
```

If bit 2 of `$cplusplus` is set, this `bi` command will act as if the `-C` option were given. To enable this default behavior execute:

```
p $cplusplus |= 4
```

The `$cplusplus` variable also contains bits to control behavior of other C++ features, so it is important to affect only bit 2 when using it to modify the behavior of the `bi` command. By default, bit 2 of `$cplusplus` is cleared.

The listing for this type of breakpoint will be similar to this example:

```
1: Active instance functions (class myclass): object
```

Notice that the class name is included as well as the expression that was used to specify the instance.

5

When the `bi` command sets breakpoints on member functions of base classes, the debugger will indicate this when listing the breakpoint. For example:

```
1: Active instance functions (class myclass and base classes): object
```

Because class instances have limited lifetimes, it makes sense that instance breakpoints have lifetimes to match the instances themselves. When possible, the debugger attempts to delete an instance breakpoint automatically when the instance with which it is associated is destroyed. Sometimes this is not possible because the expression in the `bi` command involves dereferences, making it impossible to determine the lifetime of the instance. However, in all cases where the debugger can make a determination, the breakpoint is removed automatically. If any instance breakpoints remain when the program being debugged terminates, they are automatically removed.

---

## Handling Exceptions

The C++ language provides *exception handling* for dealing with special conditions produced by the programmer as well as for dealing with execution of invalid programming operations (for example, dividing by zero). The statements used by C++ to deal with exceptions are:

- |              |  |
|--------------|--|
| <b>try</b>   | Groups together statements where a set of exceptions can be handled.   |
| <b>throw</b> | Allows you to force an exception when a certain condition occurs, and passes the exception on to an exception handler. |
| <b>catch</b> | Designates where execution will continue when an exception of a specified type is thrown.                              |

For more information on **try**, **throw** and **catch**, see the *HP C++ Programmer's Guide* (Part Number: 92501-90005).

The topics covered in this section are:

- Using **throw** and **catch**
- Stopping on a **throw** statement
- Executing a **throw** command list
- Stopping on a **catch** statement
- Executing a **catch** command List
- Listing exceptions
- Inhibiting auto-destructors on **throw** and **catch**
- Exception command's effect on other commands

### Using **throw** and **catch**

The following program `divzero.C` tests for a divide by zero exception. If the divisor does equal zero, the **throw** statement passes a message to the **catch** statement which prints the message:

```
Division by zero is not legal.
```



If the divisor does not equal zero, then the result of the division is displayed on stdout. Here is the program `divzero.C`:

```
#include <stream.h>
#include <stdlib.h>

double divide(double,double);

main(void)
{
double i, j, result;

cout << "Enter the dividend and then the divisor:" << "\n";
cin >> i >> j;

5 try
  {
    result = divide (i,j);

    cout << "The result of dividing i by j is: " << result;
  }

catch (const char* v1)
  {
    cout << v1 << "\n";
    exit(1);
  }
}

double divide (double a1, double a2)
{
  if ( a2 == 0.0 )
    throw "Division by zero is not legal.";
  return a1/a2;
}
```

The program `divzero.C` will be referred to in subsequent sections in this chapter.

## Stopping on a throw Statement

By default, the debugger stops immediately prior to an exception throw. To toggle this behavior, execute either of these commands:

```
toggle exception throw
```

or

```
txt
```

When this toggle is enabled (which is the default), program execution will stop at any actual `throw` statement. You are then notified that a throw is about to occur, and you are either given an indication of where (what function and line number) the exception will be caught, or a warning if the exception will not be caught.

To try the stop-on-throw feature, compile the program `divzero.C` with the `-g` option, and execute the debugger command (`xdb`) with the `a.out` file. The content of the source file `divzero.C` can now be seen in the source window of the debugger.

To run the program found in the source window, execute:

```
run
```

At the prompt:

```
Enter the dividend and then the divisor:
```

type first the value `1.0` and a space and then the value `0.0` and press Return. Since, by default, the debugger stops at all `throw` statements, execution stops because the program has detected the invalid divisor and initiated a throw. The source window marker (`>`) is now pointing at the following line in the source file:

```
throw "Division by zero is not legal.";
```

You can now execute this command:

```
p a1
```

and the value of the variable `a1` (the numerator) is displayed in the command window.

## Executing a throw Command List

To define a debugger *command-list* to be executed when a stop on `throw` occurs, execute either of these commands:

```
exception throw command [ command-list ]
```

or

```
5 xtc [ command-list ]
```

When the `exception throw command` (`xtc`) is enabled and a stop on a `throw` occurs, the debugger executes the given *command-list*. The default *command-list* is empty (that is, execution is suspended).

In the section “Stopping on a throw Statement,” you executed the `print` command after stopping at the `throw` statement. If you would rather have the debugger print the numerator and then proceed with the `throw`, use the following command:

```
exception throw command {p a1;c}
```

or

```
xtc {p a1;c}
```

This command will stop the program at the `throw` statement and execute the commands shown in the *command-list*.

If the first command in the *command-list* is `Q`, the debugger will not print any messages normally printed upon stopping at a `throw` statement.

## Stopping on a catch Statement

By default, the debugger stops at the first statement of any `catch` clause. To turn this behavior *off*, execute either of these commands:

```
toggle exception catch
```

or

```
txc
```

When this toggle is enabled, program execution will stop at the first statement of any `catch` clause. You are then notified that a `catch` has occurred, and you are given an indication of where (what procedure or function and line number, if known) the exception was thrown from. The caught object behaves as if it were declared locally within the `catch` clause.

To try the stop-on-catch feature, compile the program `divzero.C` with the `-g` option, and execute the debugger command (`xdb`) with the `a.out` file. The content of the source file `divzero.C` can now be seen in the source window of the debugger. Next, in the command window, disable the stop-on-throw statement by executing the following command:

```
txt
```

To run the program found in the source window, execute:

```
run
```

At the prompt:

```
Enter the dividend and then the divisor:
```

type first the value `1.0` and a space and then the value `0.0` and press Return. Since, by default, the debugger stops at all `catch` statements, execution will stop because the program has detected the invalid divisor and initiated a throw. The throw has been ignored by the debugger (because the `toggle exception throw` was executed), but stopping on a `catch` statement is still enabled. The source window marker (`>`) is now pointing at the following line in the source file:

```
cout << v1 << "\n";
```

You can now execute this command:

```
p v1
```

and the value of the variable `v1` (error message) is displayed in the command window.

## Executing a catch Command List

To define a debugger *command-list* to be executed when a stop on `catch` occurs, execute either of these commands:

```
exception catch command [ command-list ]
```

or

```
xcc [ command-list ]
```

5

When the `exception catch command` (`xcc`) is enabled and a stop on a catch occurs, the debugger executes the given *command-list*. The default *command-list* is empty (that is, execution is suspended).

In the section “Stopping on a catch Statement,” you executed the `print` command after stopping at the statement just after the `catch` statement. If you would rather print the message and then continue through the catch, use the following command:

```
exception catch command {p v1;c}
```

or

```
xcc {p v1;c}
```

This command will stop the program at the `catch` statement and execute the commands shown in the *command-list*.

If the first command in the *command-list* is `Q`, the debugger will not print the messages normally issued upon stopping at a `catch` statement.

## Listing Exceptions

To list the current state of the `throw` and `catch` toggles, and command-list associated with them, execute either of the following commands:

```
list exceptions
```

or

```
lx
```

An exceptions listing looks like this:

```
Stop on throw is enabled.  
Throw command: {Q;p "hello\n";c}  
Stop on catch is enabled.  
Catch command: none.  
Destruction of auto-objects is disabled.
```

## Exception Command's Effect on Other Commands

The exception command affects the Step-into (**s**) and Step-over (**S**) commands. This section explains how these commands are affected by the exception commands.

### Step-Into (**s**)

Issuing an **s** command when stopped at a **throw** statement will cause the debugger to step into the first statement of the first member-function (compiled with the **-g** command-line option) implicitly called as a result of the **throw** statement. If a simple type is thrown (that is, no constructors are implicitly called), the debugger will step directly to the **catch** clause if it was compiled with the **-g** command-line option.

5

If a statement count is given with the **s** command, the debugger will proceed until either that many statements have been executed, a breakpoint is reached, or the **catch** clause is reached.

### Step-Over (**S**)

Issuing an **S** command when stopped at a **throw** statement will cause the debugger to step directly to the appropriate **catch** clause. The debugger will execute through any member-functions implicitly called as a result of the **throw** statement unless a breakpoint is encountered in one of those members.

If a statement count is given with the **S** command, the debugger will proceed until either that many statements have been executed, a breakpoint is reached, or the **catch** clause is reached.

---

## Debugging Parameterized Types

This section describes symbolic debugger commands that support C++ code that uses parameterized types. All of the features covered in this chapter require that you compile with the symbolic debug option (`-g` or `-g1`).

In general, class template names can be used anywhere a class name is valid. Likewise, function template names can be used anywhere a function name is valid. If the template name appears with arguments, the given operation is performed only on that particular instance of the template. On the other hand, if the template name appears without any arguments, the operation is performed on all instances of that template.

The HP symbolic debugger has the ability to provide the following support for parameterized types:

- Reference a class template or template class wherever a location-specifier is valid.
- Set breakpoints in any or all class template member functions (affecting all instances of that template).
- Set breakpoints in any or all member functions of a single instance of a class template (affecting only one instance).
- Reference a function template or template function wherever a location-specifier is valid. Any instances of a function template can be treated as any other non template function (for example, in a command-line procedure call). All template functions are included in all “all procedure” breakpoint commands.
- Set breakpoints at any location in a function template (affecting all instances of that template).
- Set breakpoints at any location in an instance of a function template (affecting only one instance).
- Print the definition (type) of any class template or a single instance of that template.
- Reference a function template instance wherever a non-template function can be referenced (for example, in a command-line procedure-call).
- List classes or class templates by name (or partial name).



- List function templates or template functions by name (or partial name).

For more information on parameterized types, see the *HP C++ Programmer's Guide* (Part Number: 92501-90005).

## Using Parameterized Types

The following program `stack.C` shows how a generic template for a class called `stack` can be created to handle various data types (that is, the type declaration `T` in the `template` can be: `int`, `char`, and so forth). Having a template like this allows you to keep your programs easier to maintain because you do not have to create separate `push(element)`, `pop()`, and `top()` functions for the various data types you might consider using in your program. This program creates a character stack and an integer stack, pushes a value onto each stack, and then displays the value at the top of these stacks.

5

```
#include <stream.h>
template<class T, int size> class stack
{
    int stack_pointer;
    T buffer[size];
public:
    stack() {stack_pointer = -1;}
    void push(T element) {buffer[++stack_pointer] = element;}
    T pop() {return buffer[stack_pointer--];}
    T top() {return buffer[stack_pointer];}
    T pop(int n) {stack_pointer -= n;
                 return buffer[stack_pointer + 1];}
};

main(void)
{
    stack<int, 40> stack_int;
    stack<char, 20> stack_char;
    int stack_int_top, stack_char_top;

    stack_int.push(100);
    stack_char.push('c');
```

```

stack_int_top = stack_int.top();
stack_char_top = stack_char.top();

cout << "Top of the integer stack is: " << stack_int_top;
cout << "Top of the character stack is: " << stack_char_top;
}

```

If you want to debug this program, you need to compile it using the `-g` and `+d` compile-line options and then run `xdb` with the `a.out` file that is generated.

The program `stack.C` will be referred to in subsequent sections in this chapter.

## Setting Breakpoints in Templates

This section covers setting breakpoints in:

- all member functions of a class template (all instances)
- all member functions of any single template class
- any single class template member function (all instances)
- any single class template member function instance
- function templates

5

### All Member Functions of a Class Template

Template names can be used as normal class names with the existing `bpc` command. The syntax for this command is:

```
breakpoint class class-template-name
```

or

```
bpc class-template-name
```

This causes a breakpoint to be set at the first statement of all member functions in the given class template in *all* instances of that template. For example, you can set a breakpoint at the first executable statement in the functions `stack()`, `push()`, `pop()`, and `top()` by using the following command:

```
bpc stack
```

Note that a breakpoint will be set in `stack<int>::pop()` and in `stack<char>::pop()` (as well as all other member functions).

A template name can be used in place of a class name in the `bpo` command. For example:

```
bpo stack::pop
```

sets a breakpoint at:

```
stack<int>::pop()
stack<int>::pop(int n)
stack<char>::pop()
stack<char>::pop(int n)
```

### All Member Functions of a Template Class

- 5 Template class names can be used as normal class names with the existing `bpc` command. The syntax for this command is:

```
breakpoint class template-class-name<args>
```

or

```
bpc template-class-name<args>
```

This causes a breakpoint to be set at the first statement of *all* member functions of the given template class.

To set a breakpoint in the member functions of `stack<int>`, but not in `stack<char>`, use the following command:

```
bpc stack<int>
```

A template class name can be used in place of a class name in the `bpo` command. For example:

```
bpo stack<int>::pop
```

sets a breakpoint at:

```
stack<int>::pop()
stack<int>::pop(int n)
```

## A Single Class Template Member Function

The following command causes a breakpoint to be set at the given line (default is the first statement) of the named member function in *all* instances of the named template:

```
breakpoint class-template-name::member-function-name[ :line ]
```

or

```
b class-template-name::member-function-name[ :line ]
```

If the current viewing location is within a member function of a class template, the **b** (breakpoint) command will take into consideration that a template member is being referred to, and the actual breakpoint will be set at the corresponding location in *all* instances of that template.

To set a breakpoint at only the `push()` function, use the following command:

```
b stack::push
```

This will set a breakpoint at the first statement in both `stack<int>::push()` and `stack<char>::push()`.

## A Single Class Template Member Function Instance

The following command causes a breakpoint to be set at the given line (default is the first statement) of the named member function in only the specific instance of the named template:

```
breakpoint class-template-name<args>::member-func-name[ :line ]
```

or

```
b class-template-name<args>::member-func-name[ :line ]
```

If you only want to set a breakpoint in a given member function of a given instance, you need to give the full class name. You can use the following command to do this:

```
b stack<int>::push
```

## Function Templates

Function templates work in much the same way as class templates. You can refer to one or all instances (instances) of the function. For example:

```
b function-template-name[:line]
```

If only one instance exists, this command sets a breakpoint at the given line (default is the first line) in that instance of the named function template.

If several instances exist, you will be given a menu of function templates from which you can choose one instance. Note that one of the menu options will allow you to set a breakpoint on all of the instances.

## Displaying Template Data

5 This section covers displaying:

- data member values in a template class
- calling a template function
- calling the type of an object declared as a template class
- the type of an template class

### Data Member Values in a Template Class

Template classes can be used anywhere a normal class can be used. The syntax for the command that displays the value of a data member in a template class is:

```
print class-template-name<args>::member-name
```

### Calling a Template Function

A template function can be used in a command-line procedure call. This works just like calling an overloaded routine, and you will be prompted with a menu to choose exactly which procedure or function to call. The syntax for calling a template function from the command line is:

```
print template-function-name(arguments)
```

### The Type of An Object Declared as a Template Class

The `t` format option to the `print` command recognizes objects which are instances of a template class. For example:

```
p object\t
```

shows the actual type of `object` as an instance of a class template. Actual arguments will appear in the appropriate places within the type.

### The Template Type of an Object

The `r` and `R` format options to the `print` command recognize objects which are instances of a template class. When either option is used, the class template will be shown (`R` also causes base classes to be printed). For example:

```
p object\r
```

or

```
p object\R
```

shows the actual type of `object` as the class template. No actual arguments relevant to `object` will appear.

### Listing Templates

This section covers the listing of:

- Classes
- Class templates
- Function templates
- Template functions

## Classes

The listing command in this section lists all classes known to the debugger. The optional *string* causes only classes whose names start with that *string* to be listed. The syntax for the command to list all classes is:

```
list classes [ string ]
```

or

```
lcl [ string ]
```

This command lists both regular classes and class templates.

To list instances of a template, use the following syntax:

```
lcl template-name<
```

5

## Class Templates

The listing command in this section lists all class templates known to the debugger. The optional *string* causes only templates whose names start with that *string* to be listed. The syntax for the command to list all class templates is:

```
list class templates [ string ]
```

or

```
lct [ string ]
```

## Function Templates

The listing command in this section lists all function templates known to the debugger. The optional *string* causes only templates whose names start with that *string* to be listed. The syntax for the command to list all function templates is:

```
list function templates [ string ]
```

or

```
lft [ string ]
```

## Template Functions

The listing command in this section lists all template functions templates known to the debugger. The optional *string* causes only template functions whose names start with that *string* to be listed. The syntax for the command to list all expansions of function templates is:

```
list template functions [string]
```

or

```
ltf [string]
```

Note that the lp (**list procedures**) command will list *all* functions, including function templates and template functions.



---

## Using Nested Classes

This chapter describes symbolic debugger support for nested classes. The topics covered in this section are:

- references to static members
- references to class names of an enclosed class

Here is an example of nested classes:

```
int x;

class A {
    static int x;

    class B {
        static int y;
        ...
        void funcB(...) // member function
    };

    void funcA(...) // member function
};
```

5

**Figure 5-1. Nested Classes**

### References to Static Members

References to static members or member functions of an enclosing class can be made without the qualification of the enclosing class name when execution is suspended within a member function of the enclosed class.

In Figure 5-1, if you are stopped in `funcB()`, you can reference `A::x` simply as `x`. The global `x` must be referenced as `::x`.

## References to Class Names of Enclosed Classes

References to class names of enclosed classes can be made when execution is suspended within a member function of the enclosing class.

In Figure 5-1, if you are stopped in `funcA()`, you can reference `funcB()` as `B::funcB()` instead of as `A::B::funcB()`.

---

## Customizing Default Debugger Behavior

It is possible to set the default behavior of certain debugger commands associated with C++. This is accomplished by setting or clearing certain bits in a special debugger variable called `$cplusplus`. The meaning of the bits in `$cplusplus` is shown in the following table. The specific commands to set or clear these bits are also included.

**Table 5-2. Bits Contained in the `$cplusplus` Variable**

Bit	Cleared	Set
bit 0	<p>When printing type or value information for a class object, any duplicate base class information will only be printed once.</p> <p><code>p \$cplusplus &amp;= ~1</code></p>	<p>When printing type or value information for a class object, all information will be printed wherever it logically appears in the object, even if this requires printing certain data more than once.</p> <p><code>p \$cplusplus  = 1</code></p>
bit 1	<p>The <code>bpc</code> command sets breakpoints only on member functions of the designated class, but not on any base classes it may have.</p> <p><code>p \$cplusplus &amp;= ~2</code></p>	<p>The <code>bpc</code> command sets breakpoints on member functions of the designated class and all of its base classes.</p> <p><code>p \$cplusplus  = 2</code></p>
bit 2	<p>The <code>bi</code> command sets breakpoints only on member functions of the instance's class type and not on functions of any base class.</p> <p><code>p \$cplusplus &amp;= ~4</code></p>	<p>The <code>bi</code> command sets breakpoints on member functions of the instance's class type and that class's base classes.</p> <p><code>p \$cplusplus  = 4</code></p>

5

---

## Sample C++ Debugging Sessions

There are two debugging sessions covered in this section. If these sessions are run on a Series 600/700/800 computer, the addresses will be similar to those shown in the example explanations for each session. These sessions demonstrate the enhanced debugging features of *xdb*. Each session gives a small C++ program, sample user input within *xdb*, and the debugger's actual output to the terminal. The first program uses a string class to print and concatenate character strings. The second program demonstrates class browsing and object identification using the C++ inheritance feature.

Note that there is also an online C++ demo found in the chapter "Getting Started." This demo guides you through some important debugger features that can be used to debug C++ programs.

### Session One

For this debugging session, the following source code will be used:

```
1: #include <stream.h>
2: #include <string.h>
3:
4:     const maxStringLen = 100;
5:
6: class String {
7:     int len;
8:     char str[maxStringLen];
9: public:
10:    String();
11:    String(char *);
12:    String operator + (String &);
13:    String operator + (char *);
14:    void print();
15: };
16:
17: String::String() {
18:     len = 0;
19: }
20:
21: String::String (char * s) {
```

```
22: len = strlen(s);
23: for (register int i = 0; i < len; i++)
24:     str[i] = s[i];
25: }
26:
27: void String::print() {
28:     for (int i = 0; i < len; i++)
29:         cout << str[i];
30:     cout << endl;
31: }
32:
33: String String::operator + (char * s) {
34:     String rslt(*this);
35:     int sLen = strlen(s);
36:     for (int i = 0; i < sLen; i++)
37:         rslt.str[rslt.len++] = s[i];
38:     return rslt;
39: }
40:
41: String String::operator + (String & t) {
42:     String rslt(*this);
43:     for (int i = 0; i < t.len; i++)
44:         rslt.str[rslt.len++] = t.str[i];
45:     return rslt;
46: }
47:
48: int stringLenCheck(int newlen) {
49:     if (newlen > maxStringLen) {
50:         cerr << "string length exceeded" << endl;
51:         return 0;
52:     }
53:     return 1;
54: }
55:
56: String s("Here's a global String.");
57:
58: main() {
59:     String s;
60:     cout << "\n--printing null String" << endl;
61:     s.print();
62:
```

```

63:  String t("!");
64:  cout << "\n--printing single character String" << endl;
65:  t.print();
66:
67:  String u = String("Hello world");
68:  cout << "\n--printing multi-character String" << endl;
69:  u.print();
70:
71:  s = u + t;
72:  cout << "\n--appending 1-char String to multi-char String" << endl;
73:  s.print();
74:
75:  cout << "\n--printing string append expression" << endl;
76:  (u+t).print();
77:
78:  cout << "\n--appending character string to String" << endl;
79:  u = u + ". Greetings from California!";
80:  u.print();
81:
82:  return 0;
83: }

```

5

The source code is assumed to be in the file `stringapp.C`. Once `stringapp.C` compiles without errors, recompile and automatically link the program using the `-g` option to generate debugging information tables to provide the debugger with the names and addresses of variables, labels, and source lines. For example, execute the following command:

```
CC -g -o stringapp stringapp.C
```

You are now ready to start the debugger. To do so, execute this command:

```
xdb stringapp
```

From this point on, enter commands on the line following the debugger's prompt `>`. To run the sample session, enter the commands shown in **computer font**. Note that the debugger's response is also shown.

To set a breakpoint at line 65, execute the following debugger command:

```
>b 65
```

The debugger displays the following information:

```
Overall breakpoints state: ACTIVE
Added:
  1: count: 1 Active ::main(): 65: t.print();
```

After the breakpoint has been entered, begin the program by executing the following command:

```
>r
```

The debugger displays a response similar to the following:

```
Starting process 28575: "stringapp"

--printing null String

5 --printing single character String

breakpoint at 0x00001510
```

To display the value of the `String` object `s` which is local to `main()`, enter the following debugger command:

```
>p s
```

At this breakpoint, the contents of object `s` are a null `String` and the results displayed are as follows:

```
s = class String {
private:
  len = 0;
  str = "";
}
```

To display the value of the global `String` object which is initialized at line 56, prefix the C++ scope operator `::` to the `String` object `s`. For example, executing this debugger command:

```
>p ::s
```

results in the following information being displayed:

```

s = class String {
private:
    len = 23;
    str = "Here's a global String.";
}

```

To display the type of object `s` as declared, use the `t` format specifier. This specifier displays the data members with their types as well as the member functions and their prototypes. For example, executing this debugger command:

```
>p ::s\t
```

results in the following information being displayed:

```

class String {
private:
    long len;
    char str[100];
public:
    String();
    String(char *);
    String operator+(String &);
    String operator+(char *);
    long print();
} s

```

5



To set a breakpoint at the constructor `String`, use the following debugger command:

```
>b String::String
```

Note again the use of the class scope operator to indicate that the function is a member of the class `String`. In this case, the constructor name is overloaded. When you enter the name of an overloaded function, the debugger lists all overloaded functions with the name `String` and asks you to select the one you want as shown:

```
Overloaded function; please choose one:
1 String::String();
2 String::String(char *);
function number? 2
```

5

At the prompt, enter your selection 1 or 2. For the purpose of this example, type 2 and press `(Return)`. This causes the following information to be displayed:

```
Overall breakpoints state: ACTIVE
Added:
2: count: 1 Active String::String(char *):21:String::String(char * s) {
```

Because it is known that the function is overloaded, a breakpoint can be set at each of the `String` constructors by executing the command:

```
>bpo String::String
```

This causes the following information to be displayed:

```
Overall breakpoints state: ACTIVE
Added:
3: Active overloaded functions: String::String
```

To continue executing the program at line 65, execute this debugger command:

```
>c
```

This causes the following information to be displayed:

```
!

breakpoint at 0x00001104
```

Note that the breakpoint located at the constructor `String::String (char *)` has been reached.

With the debugger stopped in the member function `String (char * s)`, execute this debugger command:

```
>p s
```

which displays the value of the function argument `s` that was passed to it in line 67. For a variable of type `char *`, the name of the variable and the value of the string pointed to are displayed.

```
s = "Hello world"
```

If the following step command, is executed twice:

```
>s
```

the program will stop at line 23.

To display the value of the data member `len`, execute this debugger command:

```
>p len
```

Note that the following command could also be used:

```
>p this->len
```

but since the debugger supports referring to the data members of a class object without qualification while within a member function, the first command is simpler. When either of these commands is executed, the following information will be displayed:

```
len = 11
```

The request for an immediate breakpoint upon return from the current function can be accomplished with the breakpoint `uplevel` command:

```
>bu
```

Once this command is executed, the following information is displayed:

```
Overall breakpoints state: ACTIVE
```

```
Added:
```

```
4: count: 1 Active ::main(): 68: cout << "\n--printing multi-character String" << endl;
```

To continue execution at line 23, execute this debugger command:

```
>c
```

When the above command is executed, the current function is exited and the breakpoint at line 68 is reached.

```
breakpoint at 0x00001528
```

To request a breakpoint at the member function for the overloaded operator in the class `String`, use the following debugger command:

```
>b String::operator+
```

The debugger prompts you to select the desired function as seen below.

```
Overloaded function; please choose one:
1 String String::operator+(char *);
2 static String String::operator+(String &);
function number? 1
```

5

In response to the above prompt, 1 is selected. The following information is displayed:

```
Overall breakpoints state: ACTIVE
Added:
5: count: 1 Active String::operator+(char *): 34: String rslt(*this);
```

Note that the `v` (**view**) command also supports overloaded functions. Therefore, when the following command is executed:

```
>v String::operator+
```

The debugger will give the same prompt as shown above for choosing the overloaded function. Upon selecting 1, the display will show the source code centered around line 34.

To return the viewing location to the point of execution, use this debugger command:

```
>V
```

To request an instance breakpoint at the member function `print`, execute the following debugger command:

```
>bi s.print
```

This results in the following information being displayed:

```
Overall breakpoints state: ACTIVE
```

```
Added:
```

```
6: count: 1 Active instance function (class String): s.print
```

To continue execution at line 68 and reach the instance breakpoint `String::print` at line 28, execute this debugger command:

```
>c
```

The following information is displayed:

```
--printing multi-character String  
Hello world
```

```
--appending 1-char String to multi-char String
```

5

```
breakpoint at 0x000011a8
```

The class object `s` can be printed using the following debugger command:

```
>p *this
```

The class object's contents are displayed as follows:

```
0x68ff3510 class String {  
private:  
    len = 12;  
    str = "Hello world!";  
}
```

The `V (View)` command displays the source at the current point of suspension at the depth in the program stack you specify. To view where the currently executing function will return, the following command will display the return point, in this case, line 75.

```
>V 1
```

To continue execution at line 28, execute this debugger command:

```
>c
```

The debugger displays:

```
Hello world!
```

```
--printing string append expression
```

```
Hello world!
```

```
--appending character string to String
```

```
breakpoint at 0x00001280
```

A breakpoint has now been reached at line 34 in:

```
String::operator+(char *s)
```

To print the value of the function argument `s`, passed from line 79, execute this debugger command:

```
5 >p s
```

The information printed is:

```
s = ". Greetings from California!"
```

To print the class object, execute the command given below. Note that for an overloaded binary operator such as `+`, this corresponds to the left-hand side of the operator expression.

```
>p *this
```

The class object information displayed is:

```
068ff3430 class String {
private:
    len = 11;
    str = "Hello world";
}
```

To continue execution at line 34, execute this debugger command:

```
>c
```

The program will finish running. Once the scope of `main()` is exited, the instance breakpoint associated with `s` becomes invalid and is removed and the following information is displayed:

```
Hello world.  Greetings from California!  
Child process terminated normally  
Deleted:  
  6: count: 1 Active   instance function (class String): s.print
```

To quit the debugger, execute this debugger command:

```
>q
```

and respond with `y` to the following prompt:

```
Really quit? y
```

The first session is ended.

5

## Session Two

For this debugging session, the following source code will be used:

```
1: /*****  
2: /*      Program to demonstrate class browsing      */  
3: /*      and object identification                  */  
4: /*****/  
5:  
6: #include <stream.h>  
7:  
8: class Base {  
9:     int base_i;  
10: public:  
11:     Base(int x);  
12:     virtual void print();  
13: };  
14:  
15: Base::Base(int x)  
16: {  
17:     base_i = x;  
18: }  
19:  
20: void Base::print()  
21: {  
22:     cout << "base_i = " << base_i << "\n";  
23: }  
24:  
25: class Inherit : public Base {  
26:     int inherit_i;  
27: public:  
28:     Inherit(int a , int x );  
29:     void print();  
30: };  
31:  
32: Inherit::Inherit(int a, int x) : Base(x)  
33: {  
34:     inherit_i = a;  
35: }  
36:  
37: void Inherit::print()
```

5

```

38: {
39:     Base::print();
40:     cout << "inherit_i = " << inherit_i << "\n";
41: }
42:
43: main()
44: {
45:
46:     Base x(10);
47:     Inherit y(20,30);
48:
49:     Base* bp = & x;
50:
51:     Inherit* ip = & y;
52:
53:     cout << "base pointer points to derived - invoking virtual print\n";
54:
55:     bp = ip;
56:
57:     bp->print();
58:
59:     return 0;
60: }

```

5

The steps described in “Session One” should be followed to compile the program and enter the debugger. From this point on, enter commands on the line following the debugger’s prompt >. To run the sample session, enter the commands shown in **computer font**. Note that the debugger’s response is also shown.



To set a breakpoint at line 51, execute the following debugger command:

```
>b 51
```

This causes the following information to be displayed:

```
Overall breakpoints state: ACTIVE
Added:
  1: count: 1 Active  ::main(): 51: Inherit* ip = & y;
```

Next, set a breakpoint at line 57 by executing the following command:

```
>b 57
```

The following information is displayed:

```
Overall breakpoints state: ACTIVE
Added:
  2: count: 1 Active  ::main(): 57: bp->print();
```

5

Execution of this debugger session is started by executing this debugger command:

```
>r
```

The process ID and breakpoint address are displayed as follows:

```
Starting process 4001: "a.out"
```

```
breakpoint at 0x00001264
```

Note that the breakpoint set at line 51 has been reached.

To display the type of the object pointed to by `Base`, execute this debugger command:

```
>p *bp\t
```

At this breakpoint, `bp` points to object `x` of class `Base`. Therefore, the debugger will display class `Base` with all its data and function members. The debugger also displays the virtual table pointer `__vptr` and a special hidden member function `Base *Base(const Base &)` which is used for copying across members.

```
class Base {
private:
    long base_i;
public:
    Base(long);
    inline Base *Base(const Base &);
    virtual long print();
private:
    __mptr *__vptr;
} <unnamed>
```

To display the object pointed to by pointer `bp`, execute this debugger command:

```
>p *bp
```

This will display the object `x` of class `Base` with current values of data members including the virtual table pointer.

```
0x68ff3288 class Base {
private:
    base_i = 10;
    __vptr = 0x40000078;
}
```

Next, continue execution at line 51 by executing this debugger command:

```
>c
```

The breakpoint is reached at line 57.

```
breakpoint at 0x00001288
```

Note that at line 55, the `Base` pointer `bp` is assigned the pointer `ip` which is a pointer to class `Inherit`. Thus, `bp` now points to object `y` which is an object of the derived class `Inherit`.

The following debugger command:

```
>p *bp\t
```

is the same command as was previously used to display the type of the object pointed to by `bp`. However, in this case, the debugger correctly identifies the type as class `Inherit` by using its object identification capability. After executing this command, the following information is displayed:

5

```
class Inherit: public Base {
private:
    long inherit_i;
public:
    Inherit(long, long);
    inline Inherit *Inherit(const Inherit &);
    virtual long print();
} <unnamed>
```

The format specifier `\t` only displays the members of the immediate class in an inheritance structure. To display members of all parent classes the format specifier `\T` needs to be specified with the `print` command as follows:

```
>p *bp\T
```

Executing this command causes the following information to be displayed:

```
class Inherit: public Base {  
  
    class Base {  
    private:  
        long base_i;  
    public:  
        Base(long);  
        inline Base *Base(const Base &);  
        virtual long print();  
    private:  
        __mptr *__vptr;  
    }  
  
private:  
    long inherit_i;  
public:  
    Inherit(long, long);  
    inline Inherit *Inherit(const Inherit &);  
    virtual long print();  
} <unnamed>
```

5

To print the object currently pointed to by `bp`, execute this debugger command:

```
>p *bp
```

The debugger again correctly recognizes that `bp` is currently pointing to object `y` of class `Inherit` and displays the current values of its data members. This command only prints the values of data members of the immediate class and not of the parent classes. The information displayed is as follows:

```
0x68ff3314 class Inherit: public Base {  
private:  
    inherit_i = 20;  
}
```

If in the above case the values of data members of parent classes are needed, the format specifier `\K` must be added to the `p (print)` command as follows:

```
>p *bp\K
```

Executing this command causes the following information to be displayed:

```
0x68ff327c  class Inherit: public Base {  
  
    class Base {  
    private:  
        base_i = 30;  
        __vptr = 0x40000060;  
    }  
  
    private:  
        inherit_i = 20;  
    }  
}
```

Execution of this session can be continued at line 57 by executing this debugger command:

5

```
>c
```

The output of the program is shown by the debugger and the program finishes running.

```
base pointer points to derived - invoking virtual print  
base_i = 30  
inherit_i = 20  
Child process terminated normally
```

To exit the debugger, execute this debugger command:

```
>q
```

and respond to the prompt with a **y** as shown:

```
Really quit? y
```

Note that the object identification capability of the HP C++ debugger demonstrated in this session will be very useful for debugging object oriented C++ applications. However, the debugger supports object identification only for classes with virtual functions. The debugger uses the address of the virtual table as a signature in identifying the correct class of an object. Object identification will not work for classes without virtual functions.

## Debugging Shared Libraries

---

Shared libraries are a feature of HP-UX that allow multiple running processes to share a single copy of common code, resulting in smaller executable files and reduced memory usage. By their very nature (run-time binding), they have the potential to improve the application or library developer's productivity by shortening the recompile/relink cycle.

In support of the library developer, the HP Symbolic Debugger has the ability to debug programs that have been linked with shared libraries, as well as the shared libraries themselves. Source-level debugging of shared libraries is fully supported, with a full set of debugger capabilities available to the shared library developer:

- View shared library sources
- Set breakpoints in a shared library
- Single-step library code in source or disassembly mode
- Set or examine data associated with the library
- Call shared-library procedures from the command line
- Debug shared libraries that are dynamically loaded with *shl\_load(3X)*.
- Examine core files produced by programs linked with shared libraries.
- Choose which libraries are of interest to minimize debugger overhead.

This chapter covers the following topics:

- Enabling the debugging of shared libraries
- How shared libraries are located by the debugger
- The static and run-time environments within the debugger
- Shared library symbols and how they are bound by the debugger
- Debugging shared libraries in an adopted process (`xdb -P`)
- Summary of extended debugger commands
- Special considerations

---

## Enabling the Debugging of Shared Libraries

This section covers how shared libraries are created and how the debugger invokes them.

### Creating the Library

Shared libraries are created by the linker with the *ld(1)* **-b** option. They are composed of one or more relocatable object files compiled with the **+z** or **+Z** (“PIC”) compiler options. The **-g** compiler option can also be used to create symbolic debug information for all or part of the shared library (See also “Creating a Program with Debugger Information” in Chapter 1, and “Preparing the Program” in Chapter 3).

---

#### Note

Debugging a program that uses shared libraries requires that the program be linked with `/usr/lib/end.o`, regardless whether any portion of the program was compiled with **-g**. Otherwise, the debugger cannot determine shared-library addresses or track library load/unload operations. Note that if *ld(1)* is used for the final link of a program, `/usr/lib/end.o` must be explicitly mentioned on the *ld(1)* command line. If a compiler is used for the final link, using **-g** is sufficient.

---

6

### Naming a Shared Library

Because of limitations in the debugger command parser, certain characters will not be recognized in a shared-library *basename*, although they may be valid HP-UX file names (see *glossary(9)* in the *HP-UX Reference*). Only the following non-alphanumeric characters are *recognized* correctly:

`. , : - ~ % ^ = +`

The full directory path of a shared library, excluding the *basename*, may be any legal HP-UX path name. For example:

```
/mnt/project/libs/lib-myshare.sl
```

where `/mnt/project/libs` is the legal HP-UX path name and `lib-myshare.sl` is the shared-library file name. Note that the full directory path is not used when referencing symbols.

## Locating Shared Libraries

A shared library is *attached* to a process shortly after the process is created, or when a shared library is programatically loaded with `shl_load(3X)`. This will be referred to as the *load time* for a given library, as opposed to *link time*, when the program is statically linked.

Shared libraries are located by the dynamic loader with either an absolute filename or a search path (ordered list of directories). For a more detailed discussion on library location and searching, refer to the section titled “Linking a Program with Shared Libraries” in the manual *Programming on HP-UX* (B2355-90026).

Briefly, load-time library search path information is initially provided to `ld(1)` by you when the program is linked:

- The library’s name which is one of the following:
  - `-llibrary` which takes into account a search path that includes all `-Ldirectory` arguments and the environment variable `LPATH`
  - A complete path name.
- The `+b path_list` option.
- The `+s` option, combined with the load-time value of the environment variable `SHLIB_PATH`.
- Both `+b` and `+s`, with their relative order on the `ld(1)` command-line defining their precedence.

The library search path used by the dynamic loader and the debugger is then found within the environment (`SHLIB_PATH`) or the program itself.

The `xdb -llibrary` option need only specify the name of the library as given to the linker (that is, without a full path, “lib” prefix, and trailing `.sl`). For example, `-lXm_debug`. The debugger will attempt to locate the library using the same information available to the dynamic loader.



Explicitly loaded libraries (*shl\_load(3X)*) may be abbreviated with the debugger `-l` option if they can be located through the search path information available in the program itself (as provided to the linker). Otherwise, an absolute path is required.

The debugger `lsl` (**list shared libraries**) or `mm` (**memory map**) commands can be used to verify what path is actually being used by the debugger to locate a library.

## Invoking the Debugger

The `xdb -l` command-line option enables full symbolic debugging of any or all of the shared libraries used by the program being debugged. If you choose not to use this option, the `-s` option enables minimal (disassembly) level debugging of all shared libraries. And it also minimizes debugger memory requirements. However, you may later enable source-level debugging for any library if you so choose (see “Explicit library references” below), as long as the `-s` option has been given.

If neither the `-s` or `-l` options are used, breakpoints and single-stepping are disallowed in any shared library and the debugger steps “over” shared-library calls as if they were system calls. However, shared-library disassembly code may still be viewed.

6

`-l library` Pre-loads the symbolic debug information (and linker symbols) in *library* into the debugger. *library* may be implicitly loaded by the program (linked in with the *ld(1)* `-l` option), or explicitly loaded by *shl\_load(3X)*.

If *library* is not a complete path name, it will be searched for using the same rules as the dynamic loader (see the previous section). The trailing `.sl` is optional in *library*, as well as the “lib” prefix (e.g. `/usr/lib/X11R4/libX11.sl` can be referred to as `-lX11`). The `.sl` suffix is assumed if it is not provided.

Note that the space between `-l` and *library* is optional.

**-1 ALL**            Pre-loads the symbolic debug information into the debugger for all shared libraries that are implicitly loaded by the program. Additional **-1** options are required for libraries that will be explicitly loaded with *shl\_load(3X)*.

Each use of the **-1 library** option loads the symbolic debug information for the named library into the debugger, making all symbols in that library (specifically, that portion of it that was compiled with **-g**) available to you when you are debugging the program.

---

## The Debugger Environment (Symbol Binding)

The debugger follows the rules used by the dynamic loader for referencing (binding) symbols. The “load” ordering of the shared libraries used by a program defines the mapping of symbols to locations in the address space, and at any given point in the program’s execution, any symbol normally has one and only one definition. The section titled “Linking a Program with Shared Libraries” in the manual *Programming on HP-UX* (B2355-90026) provides additional information on symbol binding semantics.

The debugger `ls1` (`list shared libraries`) command enumerates the set of all shared libraries known to the debugger, including:

- All implicit libraries (linked with `ld(1) -l`)
- All libraries that are dependents of implicit libraries (Series 600/700/800 only)
- All dynamically loaded (explicit) libraries that have already been loaded with `shl_load()`
- All other libraries listed with the debugger’s `-l` option (in anticipation of a `shl_load()`).

6

For each library, the `ls1` command also lists the following information:

- The library’s basename, without the trailing `.sl`, that must be used to qualify symbol names
- Whether the library is currently loaded into the process (mapped)
- Whether the library contains any symbolic debug information
- If symbolic debug information for the library has been loaded into the debugger.

When there is no child process executing, the load order for all implicit libraries is known to the debugger. Libraries that are explicitly loaded with `shl_load(3X)` are not known to the debugger unless they have been identified with the `xdb -l` option, in which case they are assumed to be at the “end” of the load ordering. The `ls1` command lists libraries with this ordering.

---

**Note**

To facilitate viewing disassembly code when no child process is running, temporary (“dummy”) load addresses are assigned to each library. This is also true of any library that has been `shl_unload()`ed, or otherwise is not currently mapped when a process is running. For this reason, numerical (absolute) addresses should not be used to set breakpoints in disassembly mode. Symbolic addresses (with an offset) should be used instead.

---

When a process is executing, the dynamic loader maintains a search (bind) order and assigns actual addresses to all shared libraries currently mapped into the process. This information is also known to the debugger, and is reflected by the `mm (memory map)` command. This command shows the true bind order. It also shows where each library was mapped in the address space.

Explicitly loaded shared libraries (`shl_load(3X)`), and their binding precedence, are automatically tracked by the debugger.

When you reference a symbol (without @-qualification; see “Shared Library Symbols” below), either the static load order or the run-time bind order, depending on whether the child process is running, is used to locate the definition (that is, the “meaning”) of that symbol. Should the bind order change during execution as a result of a `shl_load()` or `shl_unload()`, a symbol’s definition may also change.

---

**Note**

The debugger has no knowledge as to whether a given symbol is exported from a shared library, and all globally-scoped symbols in a shared library are visible to the debugger, regardless of whether they have been “hidden” with the `ld(1) -h` option or explicitly exported with the `ld(1) +e` or `-E` options.

---

---

## Shared Library Symbols

Symbols defined within a shared library are displayed by the debugger as:

*symbol@libname*

where *libname* is the basename of the library without the trailing `.so`. Symbols defined in this manner are considered to be “@-qualified.” For example, `printf@libc`. *libname* is always case sensitive, and is not affected by the `tc` (`toggle case`) command.

When referencing a symbol, the library qualification (`@libname`) is not usually necessary if the current bind order and other scoping rules currently in effect are sufficient to identify the symbol. The types of symbols that may be @-qualified in this manner are: globally scoped *vars*, *procedures*, and C++ *class names*. Most other symbols (such as local variables) are identified by the scoping rules at the current viewing location.

In circumstances where a symbol is defined in the global scope of more than one shared library, the user may override the normal binding rules by explicitly @-qualifying the symbol. If the main program contains the desired definition, the program name (as shown with the `nm` or `ls1` commands) may be used following the @ character.

Two symbols (`TMEM` and `DMEM`) are predefined for shared libraries that were not listed with the `-l` invocation option and have not been explicitly referenced. Their purpose is discussed in more detail in the section “Debugging Shared Libraries in Disassembly Mode” later in this chapter.

---

### Note

If one of the allowable non-alphanumeric characters (as listed previously in the section “Naming a Shared Library”) is present in the library’s *libname* or must be used as a language operator on a shared library symbol, a qualified reference must be delimited by parentheses to avoid conflict. For example:

```
>p (structvar@mylib-v1.1).flags = 0x104
```

## Explicit Library References

To minimize the affect on debugger performance and memory requirements, the debugger does not pre-load the symbolic debug tables (or linker symbol table) of a shared library unless you have listed it with the `-l` invocation option. Only the minimal information (name, base addresses, etc.) is initially available for each library.

If `-l` has not been used for a given library and `-l ALL` was not used, you may *at any time* during the debug session force the debugger to load symbolic debug information for the library by making a `@`-qualified reference to a symbol in that library. The `lsl` (**list shared libraries**) command can be used to verify that this is necessary. Libraries containing symbolic debug information that has not yet been loaded in to the debugger will be indicated with:

Name	Mapped	SymDebug	Path
<i>basename</i>	??	Not loaded	<i>full_path_name</i>

where *basename* is the file name of the shared library, *full\_path\_name* is the path name of the shared library, and ?? is either **Yes** or **No**.

Note that if no `-l` options were used, the `-s` option is required to enable single-stepping or setting breakpoints in any shared library.

---

### Note

If the user forces the loading of a library's symbolic debug tables, the debugger will attempt to run the debug preprocessor (`pxdb`) on the library if it has not already been done (`ld(1)` usually does this). This will fail if the library has already been mapped into the process' address-space. Should this failure occur, the user must end the debugger session and manually invoke `/usr/bin/pxdb` on the shared library in order to debug it at the source level.

---

If the program being debugged stops within a shared library for any reason or a code location within a shared library is viewed in disassembly mode, it will load the symbolic debug information for that library if it is available. This will happen even if the user has not used `-l` or an `@`-qualified symbol to previously reference the library.

## Debugging Shared Libraries in Disassembly Mode

Shared libraries that were not compiled with the `-g` compiler option may still be debugged in disassembly mode (see the `td (toggle disassembly)` command in Chapter 4).

Symbolic addresses (linker symbols) defined in a shared library may also be `@`-qualified with a library name, and the `-l` option will preload the linker symbol table.

(Series 600/700/800 only) Dynamic symbols defined with `shl_definesym(3X)` are also tracked by the debugger. They may be referenced like any other linker symbol. Note that no symbolic debug information is available for these dynamic symbols. However, the debugger `ll (list labels)` command can be used to list them. Their value when referenced is their address.

If the `-l` option was not used when invoking the debugger, and no explicit `@`-qualified references have been made to symbols in a library, the following (dummy) linker symbols are used by the debugger to denote symbolic addresses within the library:

6

```
TMEM [+offset] @libname
DMEM [+offset] @libname
```

`TMEM` and `DMEM` correspond to the base address of the text and data segments in each shared library, respectively. Note that the user cannot reference these symbols directly. They are used in circumstances where the symbols for a library have not been loaded by the debugger, but addresses in the library are to be displayed (such as in a stack trace). Should you see `TMEM@libname` and desire more information, use a command such as `ll @libname` to force the loading of symbols for that library.

---

## Summary of Extended Debugger Commands

The general syntax for *locations* (for example, procedures) includes the qualification with *@libname*. This applies to all debugger commands which accept a location as an argument, such as **b** (**breakpoint**), **v** (**view**), and **p** (**print**).

In addition, some commands also accept “*string@libname*” to only allow matching of *strings* within a specific library. Also, some commands accept “*@libname*” as an argument to allow reference to the entire library.

Commands which accept *@libname* will also accept *@progrname* to denote the main program itself. *progrname* is the basename of the program being debugged, as shown by the **mm** or **ls1** commands.

The following commands accept *@libname*:

- Breakpoint status commands: The commands **lb**, **db**, **ab**, and **sb** accept *@libname* to specify that all simple breakpoints in the named library are to be listed, deleted, activated, or suspended. This allows the ability to control all breakpoints in a given library as a single group.
- All-procedure breakpoint commands: the commands **bp**, **bpt**, and **bpx** accept *@libname* to indicate that all debuggable procedures with the named library are assigned an all-procedure breakpoint. The newly added all-procedure breakpoints are *in addition to* any all-procedure breakpoints currently set in the program or other shared-libraries.

Conversely, the **dp**, **Dpt**, and **Dpx** commands also accept *@libname* to delete the all-procedure breakpoints only in the named library.

- List commands: The following commands accept *@libname* to list only those objects within the named library: **lf**, **lg**, **lo**, **lp**, **lcl**, **lct**, **lft**, and **ltf**.
- Shared library specific commands: The **mm** command accepts *@libname* (or simply *libname*) to restrict the memory-map report to the named library.

The **s** command allows stepping into a procedure call to a shared library, even if the program was linked with

```
ld -B deferred ... (which is the default)
```

and the call has not yet been *bound*.



---

## Debugging Shared Libraries in an Adopted Process (`xdb -P`)

When the `xdb -s` or `-l` option is used, the debugger will normally use a private data switch to cause libraries to be mapped private to the process (unshared, writable). When the `r` or `s` command is used to start the program, the switch is set in the process after it is created. However, if a process is adopted with

```
xdb -P pid progname
```

the debugger has no opportunity to set the special switch before process initiation. Therefore the switch must be set in the file instead. The HP-UX command `/usr/bin/pxdb` is used to set this switch.

The primary function of `/usr/bin/pxdb` is to preprocess the debugging information provided by the compilers before `xdb` uses it. This command is normally invoked by `ld(1)` as part of a compilation with the `-g` option. It also provides the means for setting the “map private” switch in the program.

The syntax used to enable debugging of shared libraries in adopted processes is:

6

```
pxdb -s  $\left[ \begin{array}{l} \text{on | enable} \\ \text{off | disable} \\ \text{status} \end{array} \right] file$ 
```

where:

**on** or **enable** These options enable shared library debugging of the adopted process by setting private data switches within the file.

**off** or **disable** These options disable shared library debugging of the adopted process by clearing private data switches within the file.

**status** This option reports whether:

1. Shared-library debugging is enabled or disabled
2. Symbolic-debug information is present
3. The symbolic-debug information has already been preprocessed.

The file is not changed when this option is given. If all three conditions are true, an exit code of 0 is returned, otherwise 1.

With any of the toggle options (**on**, **enable**, **off**, **disable**), if the executable *file* contains symbolic-debug information which has not already been preprocessed, **pxdb** will process it as well as enabling or disabling shared library debugging. Note that for any of the toggle options, *file* must be writable by the user. In case of failure (such as *file* having been linked with an old version of **/usr/lib/end.o**), diagnostics are printed and a non-zero exit code is returned.

Once **pxdb -s enable *program*** has been performed, it may then be executed and later adopted with

```
xdb -P pid -l shlib ... program
```

---

**Note**

Enabling shared library debugging of adopted processes with

```
pxdb -s enable
```

causes all shared libraries to be mapped private rather than shared, regardless of whether the program is to be debugged or not. This affects the amount of swap required by the process. For this reason, large applications should be **disabled** if they are to be executed without the expectation of being debugged.

---

---

## Special Considerations

For more information on special considerations, read the section “Shared Library Limitations” in Appendix F in this manual.

- Debugging a program that uses shared libraries requires the program be linked with `/usr/lib/end.o`, regardless of whether any portion of the program was compiled with the `-g` compiler option. Otherwise, the debugger cannot determine shared library addresses or track library load/unload operations. Note that if `ld(1)` is used for the final link of a program, `/usr/lib/end.o` must be explicitly mentioned on the `ld(1)` command line. If a compiler is used for the final link, using `-g` is sufficient.
- Full shared library debugging capabilities, including core file support, require that the most current versions of the files `/lib/crt0.o` (for Series 300/400 FORTRAN use `/lib/frt0.o`) and `/usr/lib/end.o` be linked with the program. Consequently, programs linked on earlier releases may restrict the use of some debugger features relating to shared libraries.
- The debugger assumes that any shared library listed with the `-l` option ends with `.sl` unless a complete path name is provided.
- Using an `@`-qualified symbol in an expression where one of these characters  
`. , : - ~ % ^ = +`  
is in the library’s basename, or where one of these characters must be applied as an *operator*, requires the entire symbol to be delimited by parentheses: `(symbol@libname)`.
- Shared libraries support a versioning mechanism which allows older copies of procedures to be retained in a library, even when the procedure has been changed. The debugger can only support source-level debugging of the most recent version of a procedure in a shared library, although disassembly level debugging is possible on older versions.
- If a shared library is modified between successive invocations of the child process from the debugger (successive `r` commands), the debugger will print a warning, discard any breakpoints currently set within that library, and reload the symbolic debug information if it was previously loaded.
- Shared libraries that are loaded with `shl_load(3X)` can be located by the debugger before they are actually loaded if the user identifies the library to

the debugger with the `-l` option. If a complete path name is not provided, the debugger will attempt to locate the library using path information available from the main program. If the linker `+s` option was used, the environment variable `SHLIB_PATH` helps to locate shared libraries.

- Shared libraries that are loaded with the `BIND_FIRST` modifier to `shl_load(3X)` may not be properly bound by the debugger before a child process is run. Symbols in such libraries should always be `@`-qualified to ensure proper binding.



# A

## Messages

---

This appendix lists messages that you may encounter while using HP Symbolic Debugger. Self-explanatory messages and those which relate to syntax errors, such as missing or extraneous characters in commands, are not listed in this appendix.

To assist you in finding the solution to a problem, several messages may be displayed. Look up each message in this appendix to get complete information about the action to take.

Messages are preceded by unique reference numbers that indicate the error type. Messages, with their message reference numbers, are listed in this order:

UE42-UE2031                      User Errors

Internal error messages, which are in the range of IE500 to IE825, should not occur with normal debugger use. If they do occur, report them to your HP representative.

Child process (program) errors result in signals which are communicated to the debugger. If a program error occurs while executing a procedure call from the command line, it is handled like any other error (in other words, you can investigate the called procedure). To recover from this, or to abort a procedure call from the command line, press the shell interrupt key (usually **CTRL**C.)

The following example message has a reference number of UE312 and is listed below as it appears in this appendix:

```
UE312            MESSAGE            Invalid breakpoint type "TEXT"
```

A list of terms and abbreviations that are used throughout this appendix and their meanings follow. Note that in all explanations, commands are given in long form, but the short form may also be used. See the chapter “HP Symbolic Debugger Commands” for further details.

<u>TERM/ABBREVIATION</u>	<u>DEFINITION</u>
<i>ADDRESS</i>	A 32-bit hexadecimal number.
<i>CMD</i>	A debugger command.
<i>COREFILE</i>	The name of a file containing the core image of a terminated process.
<i>FILE</i>	The name of a file.
<i>FMT</i>	A single character print-format.
<i>NAME</i>	The name of a data object.
<i>NUM</i>	A number.
<i>OBJFILE</i>	A relocatable a.out file (“dot-oh”).
<i>PROC</i>	A user program or procedure name.
<i>PROGRAM</i>	The name of an executable program (or, in some cases, a shared library).
<i>SHARED-LIBRARY</i>	The path name of a shared library or the basename (without the trailing .sl).
<i>TEXT</i>	A text string; arbitrary user input.
<b>A</b> <i>UEnnn</i>	User-created error.

---

## User Errors (UE42 - UE2031)

User errors result from entering incorrect commands or from using the commands incorrectly. User errors cause the command that you entered to fail. You must correct the cause of the error and re-enter the command.

---

UE42	MESSAGE	WARNING: Modifying the breakpoint signal!
	CAUSE	The z (zignal) command has been used with signal parameter 5.
	ACTION	Modifying the disposition of signal 5 (SIGTRAP) will significantly affect the debugger's ability to control the program being debugged. This action is <i>not</i> recommended.
UE85	MESSAGE	WARNING: " <i>FILE</i> " does not appear to have line symbols.
	CAUSE	A program or shared-library contains symbolic debug information, but is missing the portion that contains source-line to address mappings. This may indicate a corrupt file.
	ACTION	Make sure the program or shared-library has been compiled properly.
UE86	MESSAGE	WARNING: " <i>FILE</i> " is younger than " <i>PROGRAM</i> ".
	CAUSE	The debugger has determined that the timestamp on <i>FILE</i> is more recent than the timestamp on the executable program (or shared-library) <i>PROGRAM</i> , which was compiled (in part) from <i>FILE</i> . This indicates that <i>FILE</i> has been modified (edited) since it was last compiled.
	ACTION	If <i>FILE</i> has indeed been modified, recompile <i>PROGRAM</i> . Otherwise, the <i>touch(1)</i> command can be used to adjust the timestamp on <i>PROGRAM</i> .

A



UE134	MESSAGE	Warning: <i>COREFILE</i> is older than <i>PROGRAM</i> ; ignoring <i>COREFILE</i>
	CAUSE	The time stamp on the core file is older than that on the object file. Usually this indicates that the core file is left over from an earlier program's failure. However, it can also occur if (for example): the object file has been copied, or processed by <i>pxdb</i> , after the core file was produced; the files are on NFS-mounted file systems and the system clocks are out-of-sync; etc.
	ACTION	If you are convinced the core file and executable go together, you can <i>touch(1)</i> the core file to make it more recent. The debugger will then do internal validity checks on the two files.
UE136	MESSAGE	Warning: <i>COREFILE</i> cannot be the core file for <i>PROGRAM</i> ; ignoring <i>COREFILE</i>
	CAUSE	The internal validity check which the debugger does to see if the core file and object file can be a valid pair has failed.
	ACTION	Either obtain a core file which does match the object file, or allow the debugger to ignore the invalid one (or rename it to something other than <i>core</i> to eliminate the error message).
UE142	MESSAGE	"D" command needs a directory name (in quotes)
	CAUSE	The D command was given with no argument or with an argument which is not a quoted string.
	ACTION	Given the path name of the desired directory to the D command enclosed in double-quotes ("path name").

A

UE143	MESSAGE	No labels
	CAUSE	The ll ( <i>list labels</i> ) command was used and either no linker symbol table was found, or it contained no external symbols.
	ACTION	Link the program without the <i>-s</i> linker option.
UE144	MESSAGE	No matching labels
	CAUSE	The ll ( <i>list labels</i> ) command was used with a string prefix, and no symbols in the linker symbol table matched the given prefix.
	ACTION	Verify an appropriate prefix is being used and re-enter the command. If no prefix is given, all symbols will be listed.
UE173	MESSAGE	Illegal indirection
	CAUSE	The argument list given with an r command includes an input or output redirection (> or <) without a target.
	ACTION	Leave out the < or >, or supply a target (perhaps <i>/dev/null</i> ).
UE202	MESSAGE	No linker symbol table in <i>PROGRAM</i> . Try linking without <i>-s</i>
	CAUSE	The program or shared library being debugged contains no linker symbol-table. It was either stripped (see <i>strip(1)</i> in the <i>HP-UX Reference</i> ) or linked with the <i>-s</i> option.
	ACTION	Re-link the program or shared library without the <i>-s</i> linker option, and do not strip it.

A

UE291	MESSAGE	No save state name specified
	CAUSE	The file name is missing in a save state command.
	ACTION	Re-enter the save state command with a file name for the new file.
UE300	MESSAGE	Attempt to read on non-word boundary
	CAUSE	The debugger cannot read on a non-word aligned address.
	ACTION	Do not try to read at a non-word boundary. An incorrect reference to a data item has probably been made. Note: Memory accesses are done word-at-a-time, regardless of how data is formatted in memory.
UE301	MESSAGE	Attempt to write to ODD address
	CAUSE	An attempt to write a value on a non-word or half-word boundary was made.
	ACTION	Do not try to write to an odd address. Note: Memory accesses are done word-at-a-time, regardless of how data is formatted in memory.
UE302	MESSAGE	Address not found
	CAUSE	The address is part of a command and is invalid. It is probably out of range.
	ACTION	Check the validity of the address and re-enter the command.

A

UE303	MESSAGE	Cannot read that location
	CAUSE	Access to the child process failed, possibly caused by an invalid address.
	ACTION	Check the validity of the address and re-enter the command.
UE304	MESSAGE	No child process
	CAUSE	The debugger attempted an operation that required a child process that does not exist (was not running).
	ACTION	To start a child process, use any of the <b>r</b> ( <b>run</b> ) or <b>s</b> ( <b>step</b> ) commands.
UE305	MESSAGE	No child process AND no corefile
	CAUSE	The debugger attempted an operation that required a child process or a core file.
	ACTION	Start a child process using any of the <b>r</b> ( <b>run</b> ) or <b>s</b> ( <b>step</b> ) commands, or restart the debugger on a valid core file.
UE306	MESSAGE	Attempt to write to non-word boundary.
	CAUSE	The debugger cannot write to a non-word aligned address.
	ACTION	Do not try to write to a non-word boundary. An incorrect reference to a data item has probably been made. Note: Memory accesses are done word-at-a-time, regardless of how data is formatted in memory.

**A**

UE307	MESSAGE	Cannot write that location
	CAUSE	Access to a child process failed; this may have been caused by an invalid address.
	ACTION	Check the validity of the address and re-enter the command.
UE308	MESSAGE	Bad access to child process
	CAUSE	Failed to read data from or write data to a child process. This may have been caused by an invalid address (for example, dereferencing an invalid pointer), or by an attempt to place a breakpoint in an unwritable child process code space. Other possible causes: <ul style="list-style-type: none"> <li>■ The executable file is already being debugged in a different debugging session.</li> <li>■ The process you were debugging exec'ed a different process.</li> </ul>
	ACTION	Check the validity of the data and re-enter the command. You can also: <ul style="list-style-type: none"> <li>■ Kill the other debugging session.</li> <li>■ If you need to debug the new process, adopt it with the -P option.</li> </ul>
UE310	MESSAGE	Can't set breakpoint (invalid address)
	CAUSE	The address of the specified breakpoint command was invalid or unknown.
	ACTION	Re-enter the breakpoint command with a correct address or location.

A

UE311	MESSAGE	Stack isn't that deep
	CAUSE	The debugger tried to set a breakpoint or view a procedure at an invalid depth. The child process stack was not that deep.
	ACTION	Use the trace command to list the child process stack. This will show you how deep the stack is and what procedure is at each depth on the stack.
UE312	MESSAGE	No symbols for that procedure
	CAUSE	The debugger tried to set a breakpoint using a stack depth, when the procedure at that stack depth was non-debuggable.
	ACTION	Try setting a ba (breakpoint address) using the name of the procedure; for example, ba xxx.
UE313	MESSAGE	Invalid breakpoint type " <i>TEXT</i> "
	CAUSE	<i>TEXT</i> was an invalid breakpoint type.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see valid <i>breakpoint</i> commands.
UE314	MESSAGE	Invalid command list, must be enclosed in {}
	CAUSE	The command list associated with a breakpoint or an assertion must be enclosed in {}.
	ACTION	Re-enter the breakpoint or assertion with the correct syntax.

A

UE315	MESSAGE	Invalid line number on "breakpoint" command
	CAUSE	The quantity given for a line number on a breakpoint command was an invalid numeric expression.
	ACTION	Re-enter the command with a valid expression.
UE317	MESSAGE	Can't toggle stubs OFF if current view is in a stub.
	CAUSE	(Series 600/700/800 only) The <code>tst</code> command was used when the view in the source window was at a stub.
	ACTION	Use the <code>t</code> command to determine which stack depths are not stubs, and then use the <code>up</code> , <code>down</code> , or <code>V</code> command to move the view to one of those depths. Then re-issue the <code>tst</code> command.
UE318	MESSAGE	Can't toggle stubs OFF if current <code>\$depth</code> is in a stub.
	CAUSE	(Series 600/700/800 only) The <code>tst</code> command was used when <code>\$depth</code> was set to the stack depth of a stub.
	ACTION	Use the <code>t</code> command to determine which stack depths are not stubs, and then use the <code>print</code> command to reassign <code>\$depth</code> to one of those values. Then reissue the <code>tst</code> command.
UE319	MESSAGE	Invalid line number on " <code>CMD</code> " command
	CAUSE	The quantity given for a line number on a <code>b</code> ( <code>breakpoint</code> ), <code>v</code> ( <code>view</code> ), or <code>c</code> ( <code>continue</code> ) command, was an invalid numeric expression.
	ACTION	Re-enter the command with a valid expression.

A

UE321	MESSAGE	Procedure " <i>PROC</i> " not found where specified
	CAUSE	The nesting of procedure <i>PROC</i> was not properly specified.
	ACTION	Use the trace command to list the stack and find where <i>PROC</i> is located.
UE322	MESSAGE	Can't go to negative stack levels
	CAUSE	An argument has been specified to the <i>up</i> command which is larger than the current stack depth, or <i>up</i> has been requested while the current depth is at the top of stack, or an equivalent action (for example, <i>V -2</i> ).
	ACTION	The <i>top</i> command will go to the highest possible stack level (the current top of stack).
UE323	MESSAGE	No count given for "breakpoint <i>CMD</i> " command
	CAUSE	The user failed to specify a breakpoint count (after the <i>\</i> ) for a breakpoint command.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see the correct syntax for breakpoint commands.
UE324	MESSAGE	No count given for "breakpoint" command
	CAUSE	The user failed to specify a breakpoint count (after the <i>\</i> ) for a breakpoint command.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see the correct syntax for breakpoint commands.

A



UE325	MESSAGE	No count given for "breakpoint address" command
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see the correct syntax for breakpoint commands.
UE326	MESSAGE	No count given for "breakpoint beginning" command
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see the correct syntax for breakpoint commands.
UE327	MESSAGE	No count given for "breakpoint count" command
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see the correct syntax for breakpoint commands.
UE328	MESSAGE	No count given for "breakpoint trace" command
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see the correct syntax for breakpoint commands.

A

UE329	MESSAGE	No count given for "breakpoint uplevel" command
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see the correct syntax for breakpoint commands.
UE330	MESSAGE	No count given for "breakpoint exit" command
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see the correct syntax for breakpoint commands.
UE331	MESSAGE	No count given for "CMD" command
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see the correct syntax for breakpoint commands.
UE332	MESSAGE	Count must be positive or negative
	CAUSE	A count of zero was given for a b (breakpoint) or bc (breakpoint count) command.
	ACTION	Re-enter the command with a non-zero count.

A

UE333	MESSAGE	Must specify a macro name
	CAUSE	The <code>def</code> command was entered without arguments.
	ACTION	Refer to the “Macro Facility Commands” section in Chapter 4 of the <i>HP-UX Symbolic Debugger User’s Guide</i> to see the correct syntax for the <code>def</code> command.
UE334	MESSAGE	<i>TEXT</i> is not a valid macro name
	CAUSE	An attempt was made to define a macro where the first argument to <code>def</code> was seen by the debugger as not being a name (for example: a number, an operator, etc.).
	ACTION	Use a valid name (beginning with a letter or <code>_</code> , not containing any operators) as the first argument of the <code>def</code> command.
UE335	MESSAGE	Must specify which macro to delete
	CAUSE	The <code>undef</code> command was entered to delete or undefine a macro without giving the name of the macro to delete.
	ACTION	Use the <code>lm</code> (list macros) command to list all defined macros.
UE336	MESSAGE	Unknown name or command " <i>CMD</i> "
	CAUSE	An unrecognized string ( <i>CMD</i> ) was encountered as a debugger command.
	ACTION	Refer to the <i>HP-UX Symbolic Debugger Quick Reference</i> to see tables of valid debugger commands.

A

UE337	MESSAGE	Unknown command " <i>CMD</i> " ( <i>NUM</i> )
	CAUSE	An unrecognized character ( <i>CMD</i> ) was encountered as a debugger command. ( <i>NUM</i> is the octal value of the character.)
	ACTION	Refer to the <i>HP-UX Symbolic Debugger Quick Reference</i> to see tables of valid debugger commands.
UE339	MESSAGE	Empty assertion not added
	CAUSE	The assertion command was given without an associated command list.
	ACTION	Re-enter the command and include a command-list within braces ( <code>{ }</code> ).
UE341	MESSAGE	No breakpoint set at current location
	CAUSE	An attempt was made to activate, delete, or suspend a breakpoint where no breakpoint was defined.
	ACTION	Use the <code>lb</code> ( <code>list breakpoints</code> ) command to see where breakpoints are set.
UE342	MESSAGE	Address is required after "breakpoint address"
	CAUSE	The <code>ba</code> ( <code>breakpoint address</code> ) command must be followed by a code address.
	ACTION	Use a valid code address (symbolic or numeric) with the command.
UE343	MESSAGE	Address is required after " <i>CMD</i> "
	CAUSE	The breakpoint command must be followed by a code address.
	ACTION	Use a valid code address (symbolic or numeric) with the command.

A

UE344	MESSAGE	Invalid depth given for "breakpoint <i>CMD</i> " command
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.
UE345	MESSAGE	Invalid depth given for "breakpoint beginning" command
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.
UE346	MESSAGE	Invalid depth given for "breakpoint trace" command
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.
UE347	MESSAGE	Invalid depth given for "breakpoint uplevel" command
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.
UE348	MESSAGE	Invalid depth given for "breakpoint exit" command
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.

A

UE349	MESSAGE	Invalid depth given for " <i>CMD</i> " command
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.
UE350	MESSAGE	Depth must be an integer
	CAUSE	An attempt was made to specify a stack depth that is not a number.
	ACTION	Re-enter the command and specify an integer depth.
UE354	MESSAGE	"da", "db", or "dp" is required
	CAUSE	d has been given as a command. Possible commands beginning with d are da, db, and dp.
	ACTION	Use the two letter command name, or some other command as appropriate.
UE355	MESSAGE	Must specify which assertion to delete
	CAUSE	The number of the assertion to delete was not specified.
	ACTION	Use the la (list assertions) command to find the number of the assertion to delete.
UE358	MESSAGE	Invalid expression for depth on "View" command
	CAUSE	The View command was given with an expression for a depth that the debugger cannot evaluate.
	ACTION	Use the t (trace) command to view the stack for the proper procedure and depth.

A

UE359	MESSAGE	Invalid expression for depth on "V" command
	CAUSE	The V command was given with an expression for a depth that the debugger cannot evaluate.
	ACTION	Use the <b>t</b> ( <b>trace</b> ) command to view the stack for the proper procedure and depth.
UE360	MESSAGE	"f" command needs a string argument
	CAUSE	An f command was given with an argument which is not a quoted string.
	ACTION	Give the format string argument to the f command enclosed in double-quotes ("format").
UE362	MESSAGE	"goto" must be followed by #label, line-number or offset
	CAUSE	The g ( <b>goto</b> ) command was followed by an invalid location specifier.
	ACTION	Re-enter the command with a valid program label, line-number within the same procedure, or offset.
A UE363	MESSAGE	"g" must be followed by #label, line-number or offset
	CAUSE	The g ( <b>goto</b> ) command was followed by an invalid location specifier.
	ACTION	Re-enter the command with a valid program label, line-number within the same procedure, or offset.

UE364	MESSAGE	Missing "{"
	CAUSE	The <code>i</code> ( <code>if</code> ) command did not have a brace ( <code>{</code> ) following the conditional expression. Or, the expression might have been entered incorrectly.
	ACTION	Re-enter the expression, enclosing the command-lists in braces.
UE369	MESSAGE	Unknown name " <i>NAME</i> "
	CAUSE	An unrecognized string (procedure or variable name) was encountered in an expression.
	ACTION	Use the <code>lp</code> ( <code>list procedures</code> ), <code>lg</code> ( <code>list globals</code> ), <code>l</code> ( <code>list</code> ), <code>lc</code> ( <code>list commons</code> ), or <code>ll</code> ( <code>list labels</code> ) command to list all known procedures, globals, locals, commons, or labels.
UE372	MESSAGE	Must specify which assertion to suspend
	CAUSE	The number of the assertion to suspend was not specified.
	ACTION	Use the <code>la</code> ( <code>list assertions</code> ) command to find the number of the assertion to suspend.
UE373	MESSAGE	Invalid expression given for "suspend assertion" command
	CAUSE	The <code>sa</code> ( <code>suspend assertion</code> ) command was given with an expression that the debugger cannot evaluate.
	ACTION	Use an expression which evaluates to a number.

A



UE374	MESSAGE	Invalid expression given for "sa" command
	CAUSE	The sa (suspend assertion) command was given with an expression that the debugger cannot evaluate.
	ACTION	Use an expression which evaluates to a number.
UE375	MESSAGE	Bad magic number <i>NUM.NUM</i>
	CAUSE	The file you are trying to debug is not a valid executable file.
	ACTION	Specify a valid executable file for the program to be debugged.
UE378	MESSAGE	Invalid expression given for "step" command
	CAUSE	A non-numeric expression was entered as part of the s (step) command.
	ACTION	Re-enter the command with a correct numeric expression.
UE379	MESSAGE	Invalid expression given for "Step" command
	CAUSE	A non-numeric expression was entered as part of the S (Step) command.
	ACTION	Re-enter the command with a correct numeric expression.
UE380	MESSAGE	Invalid expression given for "CMD" command
	CAUSE	A non-numeric expression was entered as part of the s (step), S (Step), t (trace), T (Trace), or sa (suspend assertion) command.
	ACTION	Re-enter the command with a correct numeric expression.

A

UE382	MESSAGE	Invalid expression given for "trace" command
	CAUSE	A non-numeric expression was entered as part of the t (trace) command.
	ACTION	Re-enter the command with a correct numeric expression.
UE383	MESSAGE	Invalid expression given for "Trace" command
	CAUSE	A non-numeric expression was entered as part of the T (Trace) command.
	ACTION	Re-enter the command with a correct numeric expression.
UE384	MESSAGE	Invalid window size
	CAUSE	The numeric expression given for the new window size on the window command was not a valid numeric expression or was outside a range that is acceptable for you screen size.
	ACTION	Re-enter the command with a valid numeric expression within the range of 1 to the number of lines on your screen minus 3.
UE387	MESSAGE	Invalid expression for mode on "exit" command
	CAUSE	The x (exit) command was given with an expression for mode that the debugger could not evaluate.
	ACTION	Replace the mode expression with a valid numeric expression.

A

UE388	MESSAGE	Invalid expression for mode on "x" command
	CAUSE	The x (exit) command was given with an expression for mode that the debugger could not evaluate.
	ACTION	Replace the mode expression with a valid numeric expression.
UE389	MESSAGE	Signal "TEXT" unknown
	CAUSE	The debugger didn't recognize the parameter to the z (zsignal) command as a valid signal.
	ACTION	Enter a signal number documented in <i>signal(5)</i> of the <i>HP-UX Reference</i> manual.
UE390	MESSAGE	Unknown name or command "CMD CMD"
	CAUSE	Your command is not recognized by the debugger.
	ACTION	Enter a valid debugger command.
UE391	MESSAGE	No playback name specified
	CAUSE	The file name is missing in a playback command.
	ACTION	Re-enter the playback command with a valid playback file name.
UE392	MESSAGE	Can't open <i>FILE</i> as playback file
	CAUSE	<i>FILE</i> does not exist or is unreadable.
	ACTION	Enter a valid file name, or change the file permission if it exists already.

A

UE393	MESSAGE	Can't open <i>FILE</i> as record file
	CAUSE	You don't have write permission in the specified directory, or a non-writable file with the same name already exists.
	ACTION	Enter a different file name, remove the old file, or change the write permission for the directory.
UE394	MESSAGE	Operand stack overflow
	CAUSE	An expression was too complicated for the expression handler to parse. A combination of more than 15 nested parentheses and/or pending operators may be the cause.
	ACTION	Re-enter the expression, using less than 15 nested parentheses.
UE396	MESSAGE	Data too big to put in the child process
	CAUSE	A string constant or other data was larger than the total size of the buffer in <i>/usr/lib/end.o</i> .
	ACTION	Re-enter a smaller string constant or data item, if applicable.
UE397	MESSAGE	Can't store into a constant
	CAUSE	The left side of an assignment statement was found to be a constant; it cannot be modified.
	ACTION	Use the <i>\t</i> display format for information on the assigned variable.

A

UE398	MESSAGE	Attempt to write to read-only register
	CAUSE	An attempt was made to change the value of a privileged register, such as a Series 600/700/800 floating-point status or exception register (\$f0 - \$f3).
	ACTION	Verify that an appropriate debugger special variable is used to reference the register in the expression, and re-enter the command.
UE399	MESSAGE	String too long for assignment
	CAUSE	An attempt was made to assign a string over 1024 bytes to an HP FORTRAN 77 CHAR*, HP Pascal string, or HP Pascal packed array of char.
	ACTION	Use the \t display format for type information of the string assigning to, and re-enter the command with an appropriately sized string.
UE400	MESSAGE	Incompatible operands for string assignment
	CAUSE	An attempt was made to assign to an HP FORTRAN 77 CHAR*, HP Pascal string, or HP Pascal packed array of char, something other than an HP FORTRAN 77 CHAR*, HP Pascal string, HP Pascal packed array of char, a string constant, or a character constant.
	ACTION	Re-enter the command with a proper assignment.
UE402	MESSAGE	Can't take the address of a constant
	CAUSE	The operand of a &, \$addr, or addr operator is marked as a constant type.
	ACTION	Use the \t display format to find the type of the operand.

A

UE403	MESSAGE	Can't take the address of a register
	CAUSE	The operand of a <code>&amp;</code> , <code>\$addr</code> , or <code>addr</code> operator is marked as a register type.
	ACTION	Use the <code>\t</code> display format to find the type of the operand.
UE404	MESSAGE	Prefix "++" not supported
	CAUSE	An attempt was made to use an unsupported <code>++</code> prefix operator.
	ACTION	Make sure there is a space between a <code>+</code> and a unary <code>+</code> operator (for example <code>2+ +5</code> ). <code>+=1</code> can be used to increment.
UE405	MESSAGE	Prefix "--" not supported
	CAUSE	An attempt was made to use an unsupported <code>--</code> prefix operator.
	ACTION	Make sure there is a space between a <code>-</code> and a unary <code>-</code> operator (for example <code>2- -5</code> ). <code>-=1</code> can be used to decrement.
UE406	MESSAGE	Invalid combination of operator and operands
	CAUSE	The debugger tried to perform a numeric operation on one or more non-numeric operands.
	ACTION	Re-enter the command with a valid expression.

**A**

UE407	MESSAGE	Unknown operator ( <i>NUM</i> )
	CAUSE	An unsupported operator, with internal value <i>NUM</i> , was pushed on the operator stack.
	ACTION	Re-enter the command using an operator known to the current language or reset <b>\$lang</b> to the language in which the operator is valid.
UE408	MESSAGE	Misformed expression
	CAUSE	An expression was entered incorrectly. The debugger attempts to show you where the error was detected in the command line. The error token might be one token beyond the actual error.
	ACTION	Re-enter the expression using operators and operands known to the current language or reset <b>\$lang</b> to the language in which the operator or operand is valid.
UE409	MESSAGE	Two operators in a row
	CAUSE	The expression handler detected an improper construct in an expression.
	ACTION	Re-enter the command with a valid expression.
UE410	MESSAGE	Postfix "++" not supported
	CAUSE	An attempt was made to use an unsupported ++ postfix operator.
	ACTION	Make sure there is a space between a + and a unary + operator (for example 2+ +5). +=1 can be used to increment.

A

UE411	MESSAGE	Postfix "--" not supported
	CAUSE	An attempt was made to use an unsupported -- postfix operator.
	ACTION	Make sure there is a space between a - and a unary - operator (for example 2- -5). -=1 can be used to decrement.
UE412	MESSAGE	FORTRAN variable not pure array
	CAUSE	An attempt was made to dereference an array that had pointer or function qualifiers, while the current language was set to FORTRAN, which does not support them.
	ACTION	Try again with \$lang set to a different language.
UE413	MESSAGE	Invalid real number
	CAUSE	The specified numeric expression was not a real number.
	ACTION	See the appropriate language reference manual, or Table 4-4 in this manual, for the format of real numbers.
UE414	MESSAGE	Misformed global name
	CAUSE	A : or :: must be followed by a variable name (string).
	ACTION	Refer to the "Entering Variable Names" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see how to specify global variables.

A



UE415	MESSAGE	Unknown global
	CAUSE	The variable specified with <code>:var</code> or <code>::var</code> was not a recognized global variable name.
	ACTION	Use the <code>lg (list globals)</code> command to list all known global variables.
UE416	MESSAGE	Need a ":" after the number
	CAUSE	In specifying a variable, <code>proc:depth:var</code> was entered incompletely ( <code>:var</code> was missing).
	ACTION	Refer to the <code>l (list)</code> command listing in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see a list of valid expression for variables.
UE417	MESSAGE	Invalid local name
	CAUSE	In specifying a variable, <code>proc[:depth]:var</code> was entered incorrectly. The variable <code>var</code> must be a valid variable name in the specified procedure, at the specified depth.
UE418	MESSAGE	Unknown local
	CAUSE	The variable specified with <code>proc[:depth]:var</code> was not a recognized local variable of <code>proc</code> .
	ACTION	Use the <code>l (list)</code> command to list all known local variables of the current <code>proc</code> , or use the <code>T (Trace)</code> command to list the locals, variables, and procedures on the stack.

A

UE419	MESSAGE	Procedure " <i>PROC</i> " not found at stack depth <i>NUM</i>
	CAUSE	In <i>proc:depth</i> , the procedure <i>PROC</i> was not on the child process stack at depth <i>NUM</i> . Either the stack was not that deep, or the procedure at that depth was not <i>PROC</i> .
	ACTION	Use the <b>t</b> ( <b>trace</b> ) command to list the stack.
UE420	MESSAGE	Unknown language
	CAUSE	An attempt was made to modify the current language by assigning an invalid language designator to the special variable <b>\$lang</b> . The valid language designators are Pascal, FORTRAN, C, C++ and default.
	ACTION	Re-enter the command with Pascal, FORTRAN, C, C++ or default as the designator.
UE421	MESSAGE	Local is not active
	CAUSE	A local variable name was recognized but the procedure it belongs to was not currently active on the child process stack.
	ACTION	Re-enter the command after its procedure has been called.
UE422	MESSAGE	Two operands in a row
	CAUSE	The expression handler detected an improper construct in an expression.
	ACTION	Refer to the "Entering Expressions" section in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> .

**A**

UE423	MESSAGE	No source file for current address
	CAUSE	The given child process address did not map to a known, debuggable source file.
	ACTION	Use the <code>lf</code> ( <code>list files</code> ) command to view the files the debugger recognizes, and re-enter the command with an appropriate address expression.
UE424	MESSAGE	No search pattern
	CAUSE	The search command ( <code>/</code> , <code>?</code> , <code>n</code> ( <code>next</code> ), or <code>N</code> ( <code>Next</code> )) was given without a search pattern (in the case of <code>n</code> ( <code>next</code> ) and <code>N</code> ( <code>Next</code> ), the previous search command <code>/</code> or <code>?</code> was provided without a pattern).
	ACTION	Refer to the individual command listings in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> for more information about search commands.
UE425	MESSAGE	No match for " <code>TEXT</code> "
	CAUSE	The search pattern ( <code>TEXT</code> ) for the <code>/</code> , <code>?</code> , <code>n</code> , ( <code>next</code> ) or <code>N</code> ( <code>Next</code> ) command was not found in the current viewing file. Note that the pattern is a literal, not a regular expression.
	ACTION	Try another pattern or view another file and search for the pattern.
UE426	MESSAGE	Invalid display format " <code>TEXT</code> "
	CAUSE	Given the data display format, or a portion of it, the <code>TEXT</code> contained invalid syntax.
	ACTION	Refer to Table 4-4 in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see valid data viewing formats.

A

UE427	MESSAGE	Format is missing "\"
	CAUSE	Because the command ends with a \, the debugger expects a format.
	ACTION	Re-enter the command with a format or without the ending \.
UE428	MESSAGE	Length not allowed with "FMT" format
	CAUSE	The data display format <i>FMT</i> does not allow the data length specification because it is irrelevant or implicit in the format.
	ACTION	Refer to Table 4-3 in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> to see valid data viewing formats.
UE429	MESSAGE	This does not appear to be a record or union
	CAUSE	The debugger tried and failed to dump the contents of a data object that was not a record or union.
	ACTION	Use the \t display format for more information.
UE430	MESSAGE	This does not appear to be a struct or union
	CAUSE	The debugger tried and failed to dump the contents of a data object that was not a struct or union.
	ACTION	Use the \t display format for more information.
UE431	MESSAGE	No count given for b command
	CAUSE	The debugger expected a breakpoint count after the \.
	ACTION	Re-enter the command with a breakpoint count, or with no \.

**A**

---

UE433	MESSAGE	No current procedure
	CAUSE	The debugger tried to list locals for the current viewing procedure when the procedure was undefined.
	ACTION	Use the <code>lp</code> ( <b>list procedures</b> ) command to list all the debuggable procedures.

---

UE434	MESSAGE	No such procedure " <i>PROC</i> "
	CAUSE	An attempt to list locals of a non-existent, or non-debuggable procedure <i>PROC</i> was made.
	ACTION	Use the <code>lp</code> ( <b>list procedures</b> ) command to list all known debuggable procedures.

---

UE435	MESSAGE	Unrecognized " <code>l</code> " command
	CAUSE	The <code>l</code> ( <b>list</b> ) command was given with a second part that was neither a known procedure name, nor a valid option.
	ACTION	Refer to the <code>l</code> ( <b>list</b> ) command listing in Chapter 4 of the <i>HP-UX Symbolic Debugger User's Guide</i> for more information.

**A**

---

UE438	MESSAGE	Exiting command line procedure call
	CAUSE	The command line procedure call environment terminated for an unusual reason, such as encountering a breakpoint during program execution, or an error was reached before the procedure was called.
	ACTION	Check the procedure call for errors and re-enter the command line procedure call.

UE439	MESSAGE	Can't pass more than <i>NUM</i> arguments to called procedure
	CAUSE	A large limit ( <i>NUM</i> ) exists on how many parameters can be passed to a procedure called from the command line.
	ACTION	Check the number of parameters for the procedure you are attempting to call. If the limit ( <i>NUM</i> ) is less than the number of parameters in the procedure, that procedure cannot be called from the command line.
UE440	MESSAGE	Argument list too long
	CAUSE	Arguments to the run command exceeded 1024 bytes.
	ACTION	Re-enter the run command with fewer arguments.
UE441	MESSAGE	Can't goto a location in another procedure
	CAUSE	The line number given to the <i>g</i> ( <i>goto</i> ) command was not an executable source line in the top procedure on the child process stack. This is not always the same as the current viewing procedure.
	ACTION	Re-enter the <i>g</i> ( <i>goto</i> ) command with a line number within the procedure on the top of the child process stack.
UE442	MESSAGE	Signal <i>NUM</i> unknown
	CAUSE	The debugger didn't recognize the parameter to the <i>z</i> ( <i>signal</i> ) command as a valid signal.
	ACTION	Enter a signal number documented in <i>signal(5)</i> of the <i>HP-UX Reference</i> manual.

A

UE443	MESSAGE	Signal actions are "i", "r", "s", "Q"
	CAUSE	An invalid signal action was given.
	ACTION	Re-enter the command with a valid action: i (ignore), r (report), s (stop), or Q (quietly change signal action).
UE444	MESSAGE	Unknown name
	CAUSE	An unrecognized string (procedure or variable name) was encountered in an expression.
	ACTION	Use the lp (list procedures), lg (list globals), l (list), lc (list commons), or ll (list labels) command to list all known procedures, globals, locals, commons, or labels.
UE445	MESSAGE	It appears that there's no debugging information in <i>PROGRAM</i>
	CAUSE	The program you are trying to debug doesn't contain debug information.
	ACTION	Recompile the program with the debugging directive (-g compiler option), or debug the program at the assembly language level.
UE446	MESSAGE	Misformed hex number
	CAUSE	0x or 0X was given without digits following.
	ACTION	Re-enter the command with a valid hexadecimal number.
UE447	MESSAGE	Misformed octal number
	CAUSE	An octal number starting with 0 contains an 8 or 9.
	ACTION	Re-enter the command with the correct octal number.

A

UE448	MESSAGE	Character constant is missing ending '
	CAUSE	Token parsed as a character constant is missing a trailing single quotation mark ('). This applies to a single quotation mark followed by a single character or an equivalent backslash sequence.
	ACTION	Re-enter the command enclosing the character constant in single quotation marks (').
UE449	MESSAGE	String constant is missing ending "
	CAUSE	Token parsed as a string constant was missing a trailing double quotation mark before the end of the command line.
	ACTION	Re-enter the string with a beginning and ending double quotation marks.
UE450	MESSAGE	Macros nested too deeply
	CAUSE	A user specified macro has caused the evaluation of over 20 macro definitions during its evaluation. The debugger cannot evaluate macros nested this deep. This error can also be caused by a recursive macro definition.
	ACTION	Redefine the macro using fewer than 20 macro definitions, or remove the recursive definition.
UE451	MESSAGE	Macros processing overflow
	CAUSE	While evaluating a user specified macro, the buffer used to hold the resulting definition for this macro was about to overflow, and the processing for this macro terminated unsuccessfully.
	ACTION	Undefine the unnecessary macros and redefine the macro.

A



UE452	MESSAGE	Sorry, you can't access a naked field
	CAUSE	An attempt was made to refer to a field by name without specifying the qualifying structure (for example, union, record, pointer, etc.).
	ACTION	Use the \t display format on the structure object to examine its type information.
UE453	MESSAGE	Too many subscripts
	CAUSE	An attempt was made to dereference an array with more dimensions than it was declared to have. However, HP C does allow you to dereference pointers in this manner.
	ACTION	Use the \t display format for on the array object to examine its type information.
A UE454	MESSAGE	It appears that there's no debugging information in <i>SHARED-LIBRARY</i>
	CAUSE	The -l invocation option was used with <i>SHARED-LIBRARY</i> , but it contains no symbolic debug information.
	ACTION	Don't use -l <i>SHARED-LIBRARY</i> when invoking the debugger, or compile all (or some) of <i>SHARED-LIBRARY</i> with the -g compiler option.
UE455	MESSAGE	Invalid field access: " <i>NAME</i> "
	CAUSE	An attempt was made to do a field dereference of an object ( <i>NAME</i> ) that was not a structure or union.
	ACTION	Use the \t display format to determine the characteristics of the object ( <i>NAME</i> ).

UE456	MESSAGE	No such field name " <i>NAME</i> " for that record
	CAUSE	The record did not contain a field of that <i>NAME</i> .
	ACTION	Use the \t display format for more information.
UE457	MESSAGE	No such field name " <i>NAME</i> " for that struct
	CAUSE	The struct did not contain a field of that <i>NAME</i> .
	ACTION	Use the \t display format for more information.
UE458	MESSAGE	No such field name " <i>NAME</i> " for that union
	CAUSE	The union did not contain a field of that <i>NAME</i> .
	ACTION	Use the \t display format for more information.
UE459	MESSAGE	Illegal cast
	CAUSE	The expression contains an illegal cast.
	ACTION	Re-enter the command with a valid expression. When casting with a class, structure, or union type, the keyword <code>class</code> , <code>struct</code> , or <code>union</code> must be given.
UE460	MESSAGE	Mismatched parenthesis around name: <i>NAME</i>
	CAUSE	The debugger could not parse an expression containing a symbol.
	ACTION	Re-enter the expression, making sure parenthesis are correctly nested.

A

UE461	MESSAGE	No child process or corefile
	CAUSE	The debugger attempted an operation that required an active child process or a core file.
	ACTION	Start a child process using any of the <b>r</b> ( <b>run</b> ) or <b>s</b> ( <b>step</b> ) commands, or restart the debugger on a valid core file.
UE463	MESSAGE	Program died in unknown location, pc = <i>ADDRESS</i> Stack trace will not be possible.
	CAUSE	The corefile being used was created on an older system which does not support shared-library corefiles, or the program that aborted was not linked with a current version of <i>/lib/crt0.o</i> (for Series 300/400 FORTRAN use <i>/lib/frt0.o</i> ). In either case, the debugger could not relate the program-counter recorded in the corefile back to a shared-library address in the program.  (Series 600/700/800 only) Because stack-unwind information is not available for shared-libraries in the corefile, a stack trace is not possible.
	ACTION	Relink the program using a current version of <i>/lib/crt.o</i> , and re-execute the program on a current version of HP-UX.
UE464	MESSAGE	Operator stack overflow
	CAUSE	An expression was too complicated for the expression handler to parse. A combination of more than 15 nested parentheses and/or pending operators may be the cause.
	ACTION	Re-enter the expression, using less than 15 nested parentheses.

A

UE465	MESSAGE	Can't execute child program
	CAUSE	The debugger could not execute the object file given.
	ACTION	Check to see that the file is executable and writable by the user.
UE466	MESSAGE	Window mode required for this command
	CAUSE	The debugger was probably invoked with the -L option.
	ACTION	Verify that you are using a terminal that supports window mode and rerun the debugger without the -L option.
UE475	MESSAGE	Count must be positive
	CAUSE	The count argument given to the c (continue) command is negative or 0.
	ACTION	Re-enter the command with a positive count (or none).
UE476	MESSAGE	Too many characters in wide-character constant
	CAUSE	More than one valid (possibly multi-byte) character was entered.
	ACTION	Re-enter the expression with one character constant.
UE477	MESSAGE	Wide string constant too long; truncating to <i>NUM</i> wide-characters
	CAUSE	Not enough buffer space was available in the user process to store the entire string constant (maximum is 127).
	ACTION	Enter a shorter string constant.

A

UE478	MESSAGE	Wide string constant contained unmappable chars in the current locale
	CAUSE	A wide-character string contains an element that cannot be mapped back to the external character set with <i>wctomb(3C)</i> .
	ACTION	Re-enter a valid string, and/or restart the debugger with a correct locale setting (environment variable <i>LC_CTYPE</i> ).
UE479	MESSAGE	Empty hex escape sequence
	CAUSE	An invalid ANSI C hexadecimal escape sequence was entered.
	ACTION	Replace the invalid escape sequence with a valid one of the form <i>\xhh</i> .
UE480	MESSAGE	Long double function calls are not supported
	CAUSE	There was an attempt to call from the command line a function whose return type is long double.
	ACTION	This is not supported.
A UE481	MESSAGE	Long double parameters are not supported in command line function calls
	CAUSE	There was an attempt to call from the command line a function which expects a long double parameter.
	ACTION	This is not supported.
UE482	MESSAGE	Unknown print-mode
	CAUSE	There was an attempt to assign an illegal value to the <i>\$print</i> debugger variable.
	ACTION	Assign one of these values: <i>ASCII</i> , <i>native</i> , <i>raw</i> .

UE483	MESSAGE	Misformed binary number
	CAUSE	A misformed binary number was found in an expression.
	ACTION	Replace the misformed number with a valid one. (Ob or OB followed by one or more 0's or 1's)
UE484	MESSAGE	Can't open " <i>FILE</i> " as state file
	CAUSE	The file already exists and is not writable, or the directory has the wrong permissions.
	ACTION	Remove the old file, or make the directory writable and executable.
UE486	MESSAGE	Can't open " <i>FILE</i> " as restore file
	CAUSE	The file doesn't exist or the directory is not readable.
	ACTION	Enter a valid file name or add read permission to the directory.
UE488	MESSAGE	No restore name specified
	CAUSE	No file name was specified with the -R option.
	ACTION	Invoke the debugger with a restore file name or don't provide the -R option.
UE490	MESSAGE	Wrong objectfile for this statefile
	CAUSE	The save file specified was not created with the object file you are trying to debug.
	ACTION	Specify a valid state file, or if you must use the one originally specified, start the debugger and use the file as a playback file. Be sure to read the warnings related to state files before doing this.

A

UE578	MESSAGE	exec of PXDB failed: <i>IMMEDIATE CAUSE</i>
	CAUSE	The debugger's attempt to invoke PXDB to preprocess the debug information in the program failed due to <i>IMMEDIATE CAUSE</i> . Either <i>/usr/bin/pxdb</i> is not installed properly on your system, or the environment variable <i>ST_PXDB</i> is set to an improper value.
	ACTION	Check that <i>/usr/bin/pxdb</i> is correct on your system, or unset <i>ST_PXDB</i> , and try again. If necessary, re-install the DEBUGGER'S fileset.
UE582	MESSAGE	No matching instance. No breakpoint set.
	CAUSE	The user attempted to set a breakpoint on a class template member function or on a function template. No instance could be found matching the given arguments. If no arguments were specified, the given template has no instances, or all instances have been inlined.
	ACTION	Use the <i>lcl template-name&lt;</i> command to verify that an object was declared with the desired arguments.
UE583	MESSAGE	Class <i>CLASS</i> has no member function <i>PROC</i>
	CAUSE	The user attempted to set a breakpoint on a class member function which doesn't exist.
	ACTION	Use the <i>lcl (list classes)</i> or the <i>lf (list functions)</i> command to verify that <i>PROC</i> exists as a member of <i>CLASS</i> . Verify the spelling of the member function.
UE584	MESSAGE	Unexpected class name
	CAUSE	A class name of the form <i>name&lt;arguments&gt;</i> is missing its closing <i>&gt;</i> .
	ACTION	Re-enter the correct class name.

A

UE585	MESSAGE	<code>/usr/lib/end.o</code> not linked. No exception support.
	CAUSE	The file <code>/usr/lib/end.o</code> was not linked with the program or an older version (lacking support for exception handling debugging) has been linked with the program.
	ACTION	Relink the program with the correct (current) <code>/usr/lib/end.o</code> .
UE586	MESSAGE	<code>/usr/lib/end.o</code> out of date. No exception support.
	CAUSE	An older version of the file <code>/usr/lib/end.o</code> lacking support for exception handling debugging has been linked with the program.
	ACTION	Relink the program with the correct (current) <code>/usr/lib/end.o</code> .
UE587	MESSAGE	C++ library not linked. No exception support.
	CAUSE	The C++ run-time library ( <code>libC</code> or <code>libC.ansi</code> ) was not linked with the program, or an older version (lacking debugger support for exception handling) has been linked with the program.
	ACTION	Relink the program with the correct C++ library.
UE588	MESSAGE	C++ library out of date. No exception support.
	CAUSE	An older version of the C++ run-time library ( <code>libC</code> , <code>libC.ansi</code> ) lacking debugger support for exception handling has been linked with the program.
	ACTION	Relink the program with the correct C++ library.

A



UE590	MESSAGE	function or static member expected
	CAUSE	A non-function or non-static class member was given in a context where a member function or a static member is required.
	ACTION	Retype the command with a member function or a static member or use a different command.
UE593	MESSAGE	Warning: exception will not be caught; program will abort.
	CAUSE	An exception throw has occurred, and there is no corresponding catch clause for the object being thrown.
	ACTION	No action is required. The program will terminate if allowed to continue.
UE601	MESSAGE	exec of <i>A_SHELL</i> failed: <i>IMMEDIATE CAUSE</i>
	CAUSE	The debugger's attempt to invoke <i>A_SHELL</i> to execute an ! (shell escape) command failed due to <i>IMMEDIATE CAUSE</i> . Either the environment variable <i>SHELL</i> is not set properly or <i>/bin/sh</i> could not be executed.
	ACTION	Check that <i>SHELL</i> is set to an appropriate command interpreter, or if <i>SHELL</i> is unset, that <i>/bin/sh</i> is properly installed on your system.
UE605	MESSAGE	Incompatible debug information
	CAUSE	The debugger was invoked on a file linked on a older version of the operating system.
	ACTION	Try relinking your program. If that doesn't solve the problem, you will have to recompile the program.

A

UE626	MESSAGE	Attempt to read from ODD address
	CAUSE	An attempt to read from a non-word or half-world boundary was made.
	ACTION	Do not try to read from an odd address. Note: Memory accesses are done word-at-a-time, regardless of how data is formatted in memory.
UE629	MESSAGE	No Files
	CAUSE	The <code>list files</code> command was given with a pattern for which there was no match.
	ACTION	Make sure the pattern is valid, and re-issue the command.
UE631	MESSAGE	Character constant too long
	CAUSE	A C or C++ quoted character constant contains too many characters.
	ACTION	Re-enter the expression with a correct character constant.
UE632	MESSAGE	Wide-character constant not allowed ( <code>\$lang</code> must be 'C')
	CAUSE	Attempt to use a wide character constant while the language is not C.
	ACTION	Set <code>\$lang</code> to C and re-enter the expression.

A

UE633	MESSAGE	Does not map to a wide-character in the current locale
	CAUSE	A character constant (possibly multi-byte) was entered that cannot be mapped to a wide-character ( <code>wchar_t</code> ) with <code>mbtowc(3C)</code>
	ACTION	Re-enter a valid character, and/or restart the debugger with a correct locale setting (environment variable <code>LC_CTYPE</code> ).
UE642	MESSAGE	No child process AND no corefile registers
	CAUSE	The debugger attempted an operation that required an active child process or a core file.
	ACTION	Start a child process using any of the <code>r</code> ( <code>run</code> ) or <code>s</code> ( <code>step</code> ) commands, or restart the debugger on a valid core file.
UE644	MESSAGE	Registers bad in core file
	CAUSE	The core file is corrupt or incomplete.
	ACTION	Obtain a proper core file, or run the program under the debugger to the point of failure.
UE645	MESSAGE	Exec area bad in core file
	CAUSE	Unexpected exec area size. The core file might be corrupted.
	ACTION	Create a new core file.

A

UE646	MESSAGE	Error trying to read " <i>FILE</i> "; ignoring it
	CAUSE	Some error occurred while attempting to interpret <i>FILE</i> as a core file. This message will be accompanied by a specific error message unless <i>FILE</i> is empty or truncated.
	ACTION	Verify that <i>FILE</i> is the correct core file, or create a new core file.
UE654	MESSAGE	Breakpoint count ignored
	CAUSE	A count is meaningless for class, overload, or instance breakpoints on multiple member functions.
	ACTION	None required. The count was ignored but the breakpoint was set.
UE655	MESSAGE	This does not appear to be a struct, union, or class
	CAUSE	The S display format was specified but the type of the object to print is not a struct, union, or class.
	ACTION	If you want to do a formatted dump of an address, cast the address to some struct, union, or class.
UE656	MESSAGE	No such field name " <i>NAME</i> " for that class
	CAUSE	The class did not contain a field of the <i>NAME</i> .
	ACTION	Use the \t display format for more information.
UE657	MESSAGE	No such class " <i>NAME</i> "
	CAUSE	Use of a non-class name in a context that requires a class name (e.g., bpc (breakpoint class)).
	ACTION	Re-enter the command with the name of a valid class.

A

UE658	MESSAGE	No overloaded functions
	CAUSE	There are no overloaded functions to list.
	ACTION	Use the <code>lp</code> ( <code>list procedure</code> ) command to see a list of functions.
UE659	MESSAGE	No functions
	CAUSE	There are no functions to list starting with the provided prefix.
	ACTION	Re-enter the command with a valid function prefix, or just use the <code>lp</code> ( <code>list procedure</code> ) command with no prefix to see a list of all the functions.
UE661	MESSAGE	Cannot view (no debug information for file)
	CAUSE	A location was specified as <i>file:procedure</i> and the file is not in the debugger's list of files for which it has debugging information.
	ACTION	Re-enter the command with a valid file name.
UE662	MESSAGE	Cannot set breakpoint (no debug information for file)
	CAUSE	The breakpoint location was specified as <i>file:procedure</i> and the file is not in the debugger's list of files for which it has debugging information.
	ACTION	Re-enter the command with a valid file name. Use the <code>lf</code> ( <code>list files</code> ) command to list all valid source files and the path name you must use.

A

UE663	MESSAGE	Invalid file on "breakpoint" command
	CAUSE	A file specified as part of a breakpoint location is not known to the debugger.
	ACTION	Re-enter the command with a valid file name. Use the <code>lf</code> ( <code>list files</code> ) command to list all valid source files and the path name you must use.
UE664	MESSAGE	Invalid procedure on "breakpoint" command
	CAUSE	A procedure specified as part of a breakpoint location is not known to the debugger.
	ACTION	Re-enter the command with a valid procedure name. Use the <code>lp</code> ( <code>list procedures</code> ) command to see a list of all valid procedures.
UE665	MESSAGE	Invalid label on "breakpoint" command
	CAUSE	A label specified as part of a breakpoint location is not known to the debugger.
	ACTION	Re-enter the command with a valid label name.
UE666	MESSAGE	Invalid class on "breakpoint" command
	CAUSE	A class specified as part of a breakpoint location is not known to the debugger.
	ACTION	Re-enter the command with a valid class name.
UE668	MESSAGE	Ambiguous function name on "breakpoint" command
	CAUSE	A location was specified as <code>file:function</code> , and there are several C++ functions with the same name.
	ACTION	Use a class specifier instead of a file specifier to identify the desired function.

A

UE669	MESSAGE	Invalid file on "continue" command
	CAUSE	A file specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid file name. Use the lf (list files) command to list all valid source files and the path name you must use.
UE670	MESSAGE	Invalid procedure on "continue" command
	CAUSE	A procedure specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid procedure name. Use the lp (list procedures) command to see a list of all valid procedures.
UE671	MESSAGE	Invalid line number on "continue" command
	CAUSE	A line specified as part of a continue location is out of range for the associated file.
	ACTION	Re-enter the command with a valid line number.
A UE672	MESSAGE	Invalid label on "continue" command
	CAUSE	A label specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid label name.
UE673	MESSAGE	Invalid class on "continue" command
	CAUSE	A class specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid class name.

UE675	MESSAGE	Ambiguous function name on "continue" command
	CAUSE	A location was specified as <i>file:function</i> , and there are several C++ functions with the same name.
	ACTION	Use a class specifier instead of a file specifier to identify the desired function.
UE676	MESSAGE	Invalid file on "Continue" command
	CAUSE	A file specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid file name. Use the <b>lf</b> ( <b>list files</b> ) command to list all valid source files and the path name you must use.
UE677	MESSAGE	Invalid procedure on "Continue" command
	CAUSE	A procedure specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid procedure name. Use the <b>lp</b> ( <b>list procedures</b> ) command to see a list of all valid procedures.
UE678	MESSAGE	Invalid line number on "Continue" command
	CAUSE	A line specified as part of a continue location is out of range for the associated file.
	ACTION	Re-enter the command with a valid line number.
UE679	MESSAGE	Invalid label on "Continue" command
	CAUSE	A label specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid label name.

A



UE680	MESSAGE	Invalid class on "Continue" command
	CAUSE	A class specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid class name.
UE682	MESSAGE	Ambiguous function name on "Continue" command
	CAUSE	A location was specified as <i>file:function</i> , and there are several C++ functions with the same name.
	ACTION	Use a class specifier instead of a file specifier to identify the desired function.
UE683	MESSAGE	Invalid file on "view" command
	CAUSE	A file specified as part of a view location is not known to the debugger.
	ACTION	Re-enter the command with a valid file name. Use the <b>lf (list files)</b> command to list all valid source files and the path name you must use.
UE684	MESSAGE	Invalid procedure on "view" command
	CAUSE	A procedure specified as part of a view location is not known to the debugger.
	ACTION	Re-enter the command with a valid procedure name. Use the <b>lp (list procedures)</b> command to see a list of all valid procedures.
UE685	MESSAGE	Invalid line number on "view" command
	CAUSE	A line specified as part of a view location is out of the range of the associated file.
	ACTION	Re-enter the command with a valid line.

A

UE686	MESSAGE	Invalid label on "view" command
	CAUSE	A label specified as part of a view location is not known to the debugger.
	ACTION	Re-enter the command with a valid label name.
UE687	MESSAGE	Invalid class on "view" command
	CAUSE	A class specified as part of a view location is not known to the debugger.
	ACTION	Re-enter the command with a valid class name.
UE689	MESSAGE	Ambiguous function name on "view" command
	CAUSE	A location was specified as <i>file:function</i> , and there are several C++ functions with the same name.
	ACTION	Use a class specifier instead of a file specifier to identify the desired function.
UE690	MESSAGE	Invalid file on <i>CMD</i> command
	CAUSE	A file specified as part of a <i>CMD</i> location is not known to the debugger.
	ACTION	Re-enter the command with a valid file name. Use the <i>lf</i> ( <b>list files</b> ) command to list all valid source files and the path name you must use.
UE691	MESSAGE	Invalid procedure on <i>CMD</i> command
	CAUSE	A procedure specified as part of a <i>CMD</i> location is not known to the debugger.
	ACTION	Re-enter the command with a valid procedure name. Use the <i>lp</i> ( <b>list procedures</b> ) command to see a list of all valid procedures.

A

UE692	MESSAGE	Invalid label on <i>CMD</i> command
	CAUSE	A label specified as part of a <i>CMD</i> location is not known to the debugger.
	ACTION	Re-enter the command with a valid label name.
UE693	MESSAGE	Invalid class on <i>CMD</i> command
	CAUSE	A class specified as part of a <i>CMD</i> location is not known to the debugger.
	ACTION	Re-enter the command with a valid class name.
UE695	MESSAGE	Ambiguous function name on <i>CMD</i> command
	CAUSE	A location was specified as <i>file: function</i> , and there are several C++ functions with the same name.
	ACTION	Use a class specifier instead of a file specifier to identify the desired function.
UE696	MESSAGE	Must specify breakpoint to delete
	CAUSE	Although there is a breakpoint at the current viewing location, a breakpoint number must be given with the <i>db (delete breakpoint)</i> command.
	ACTION	Use the <i>lb (list breakpoints)</i> command to find the number of the breakpoint you want to delete and re-enter the <i>db (delete breakpoint)</i> command with the breakpoint number.
UE697	MESSAGE	Must specify function name
	CAUSE	The <i>bpo (breakpoint overload)</i> command was invoked without a function name.
	ACTION	Re-enter the command with a function name.

A

UE698	MESSAGE	Function not found
	CAUSE	No function matching the function name argument given to the <code>bpo</code> ( <code>breakpoint overload</code> ) command was found.
	ACTION	Re-enter the command with a valid function name. Use the <code>lp</code> ( <code>list procedures</code> ) command to see a list of all valid procedures.
UE699	MESSAGE	Must specify class name
	CAUSE	No class name argument was given to the <code>bpc</code> ( <code>breakpoint class</code> ) command.
	ACTION	Re-enter the command with a class argument.
UE701	MESSAGE	Class not found
	CAUSE	No class matching the class name given to the <code>bpc</code> ( <code>breakpoint class</code> ) command was found.
	ACTION	Re-enter the command with a valid class name.
UE702	MESSAGE	Class has no member functions
	CAUSE	The class argument given to the <code>bpc</code> ( <code>breakpoint class</code> ) command has no member functions.
	ACTION	None—This kind of breakpoint cannot be set for this class.
UE703	MESSAGE	No count given for "breakpoint instance" command
	CAUSE	The user failed to specify a breakpoint count (after the <code>\</code> ) for a breakpoint command.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the HP Symbolic Debugger User's Guide to see the correct syntax for breakpoint commands.

**A**

UE706	MESSAGE	Function is not class member
	CAUSE	The function argument to the <b>bi (breakpoint instance)</b> command is not a class member.
	ACTION	Use the <b>b (breakpoint)</b> command to set breakpoints at non-member functions
UE707	MESSAGE	No static data members
	CAUSE	This class has no static data members to print.
	ACTION	None—This message is for information purposes only.
UE708	MESSAGE	Class member required
	CAUSE	The name following the <b>::</b> is not a valid identifier for a class member.
	ACTION	Use a valid class member name after the <b>::</b> .
UE709	MESSAGE	Must specify breakpoint to suspend
	CAUSE	Although there is a breakpoint at the current viewing location, a breakpoint number must be given with the <b>sb (suspend breakpoint)</b> command.
	ACTION	Use the <b>lb (list breakpoints)</b> command to find the number of breakpoint you want to suspend and re-enter the <b>sb (suspend breakpoint)</b> command with the breakpoint number.

A

UE710	MESSAGE	Must specify breakpoint to activate
	CAUSE	Although there is a breakpoint at the current viewing location, a breakpoint number must be given with the <code>ab</code> (activate breakpoint) command.
	ACTION	Use the <code>lb</code> (list breakpoints) command to find the number of breakpoints you want to activate and re-enter the <code>ab</code> (activate breakpoint) command with the breakpoint number.
UE711	MESSAGE	Field not found
	CAUSE	There was an attempt to access a C++ class member through an invalid member or member function pointer.
	ACTION	Make sure the pointer is initialized before using it.
UE712	MESSAGE	Improper pointer conversion
	CAUSE	There was an attempt to assign the value of a pointer to a class to a pointer to another class.
	ACTION	If this type of assignment is needed, get the value of the first pointer by using the <code>p</code> ( <code>print</code> ) command, then assign the obtained value directly to the second pointer. This will bypass type checking.

A

---

UE713	MESSAGE	Static data member required
	CAUSE	There was an attempt to access a non-static class member through the use of class scope operator outside of a member function for the class.
	ACTION	Outside a member function the class scope operator <i>class::name</i> is used to access static class members only. To access a non-static member, use the <i>.</i> operator with an object, or the <i>-&gt;</i> operator with a pointer to an object.

---

UE721	MESSAGE	Pointer to member dereferenced
	CAUSE	There was an attempt to use a pointer to class member as a regular pointer.
	ACTION	A pointer to member can only be used in the following context:  <i>class:*pointer</i> <i>object.*pointer</i> <i>pointer_to_object-&gt;*pointer</i>  as it is directly connected with all objects of a specific type and does not contain an absolute address.

A

---

UE722	MESSAGE	Illegal use of pointer to member
	CAUSE	There was an attempt to use a pointer to member in an expression of the form <i>name.*pointer</i> where <i>name</i> is not the name of a class object.
	ACTION	Re-enter the expression with a valid object name.

UE723	MESSAGE	Illegal member pointer assignment
	CAUSE	There was an attempt to assign some illegal expression to a member pointer.
	ACTION	Re-enter the expression with a valid expression, that is, <i>&amp;class::member</i> .
UE724	MESSAGE	Instance not specified for function call
	CAUSE	There was an attempt to call a non-static member function as <i>class::function()</i> .
	ACTION	Call the function through an object or an object pointer, that is, <i>object.function()</i> or <i>objptr-&gt;function()</i> .
UE725	MESSAGE	Class member not found
	CAUSE	There was an attempt to print a non-existing class member.
	ACTION	Re-enter the expression with a valid class/member combination.
UE726	MESSAGE	Line not found in body of procedure
	CAUSE	There was an attempt to get the address of a <i>line</i> using the notation <i>function#line</i> where <i>line</i> is not in the body of the function.
	ACTION	Re-enter the expression with a valid function/line number combination. Use the <b>lp (list procedures)</b> command with the procedure's name. The range of valid line numbers will be displayed with the procedure.

A



UE728	MESSAGE	Warning: breakpoint not set on inlined function invocations.
	CAUSE	The last breakpoint command was been prevented from setting a breakpoint on all targeted member functions because some member functions were inlined by the compiler.
	ACTION	Compile with the <i>CC(1)</i> +d option to prevent member functions from being inlined.
UE729	MESSAGE	Invalid structure access
	CAUSE	There was an attempt to use a non-pointer or a pointer to a class member as a pointer, that is, <i>p-&gt;i</i> where <i>p</i> is not of type pointer.
	ACTION	Re-enter the expression with a valid pointer, or use the address of <i>p</i> if you need it, that is, <i>&amp;(p)-&gt;i</i>
UE730	MESSAGE	Operations on classes are not supported
	CAUSE	There was an attempt to use a class in an expression in a way not supported by the debugger, for example, trying to add two class objects.
	ACTION	If an operator was overloaded to perform the desired function, you must use the <i>operator&lt;op&gt;()</i> form of the function call. For example, the debugger won't allow <i>A + B</i> , but will accept <i>A.operator+(B)</i> .
UE731	MESSAGE	Cannot assign to function
	CAUSE	There was an attempt to assign a value to a function.
	ACTION	This is not supported by the debugger.

A

UE732	MESSAGE	Nil character constant
	CAUSE	There was an attempt to use ' ' as a character.
	ACTION	Re-enter the expression with a valid character constant. 'c', or '\value'
UE733	MESSAGE	Invalid procedure given for "breakpoint trace" command
	CAUSE	The debugger could not find a procedure with the specified name.
	ACTION	Use the lp ( <b>list procedures</b> ) command to find what procedures are known to the debugger, and re-enter the command with the corrected name. Alternatively, if the procedure you supplied was not compiled with the debug flag, you can still set a breakpoint at its entry point by using the ' <b>ba address</b> ' command.
UE734	MESSAGE	Invalid procedure given for "bt" command
	CAUSE	The debugger could not find a procedure with the specified name.
	ACTION	Use the lp ( <b>list procedures</b> ) command to find what procedures are known to the debugger, and re-enter the command with the corrected name. Alternatively, if the procedure you supplied was not compiled with the debug flag, you can still set a breakpoint at its entry point by using the ' <b>ba address</b> ' command.

A

UE735	MESSAGE	Class instance or member function required for "breakpoint instance" command
	CAUSE	There was an attempt to set an instance breakpoint on something that the debugger doesn't recognize as a class instance or a member function.
	ACTION	Re-enter the command with a valid class instance ( <i>object</i> ) or a member function ( <i>object.function</i> or <i>object_pointer-&gt;function</i> ).
UE736	MESSAGE	Class instance or member function required for "bi" command
	CAUSE	There was an attempt to set an instance breakpoint on something that the debugger doesn't recognize as a class instance or a member function.
	ACTION	Re-enter the command with a valid class instance ( <i>object</i> ) or a member function ( <i>object.function</i> or <i>object_pointer-&gt;function</i> ).
UE738	MESSAGE	Use "breakpoint instance" for instance breakpoints
	CAUSE	There was an attempt to use the regular b (breakpoint) command for an instance breakpoint.
	ACTION	Use the bi (breakpoint instance) command instead.
UE739	MESSAGE	Use "bi" for instance breakpoints
	CAUSE	There was an attempt to use the regular b (breakpoint) command for an instance breakpoint.
	ACTION	Use the bi (breakpoint instance) command instead.

A

UE756	MESSAGE	No registers in core file -- registers required
	CAUSE	The core file has a format not recognized by the debugger.
	ACTION	Obtain a new core file on the same system as the debugger you are running.
UE757	MESSAGE	Modifier is not allowed before <i>CMD</i> command
	CAUSE	In <i>cdb</i> , <i>fdb</i> , or <i>pdb</i> , a modifier was entered before a command that does not take a modifier.
	ACTION	Re-enter the command without a modifier in front of it.
UE759	MESSAGE	No code for function
	CAUSE	There was an attempt to view or set a breakpoint in a procedure that contains no code.
	ACTION	If the C++ function is pure virtual or declared but not defined, no action can be taken. If the function is inlined it can be recompiled with the <i>+d</i> option.
UE760	MESSAGE	Cannot call pure virtual function
	CAUSE	There was an attempt to call a pure virtual C++ function from the debugger command line.
	ACTION	Call the virtual function of a derived object instead.

A

UE761	MESSAGE	Cannot call inlined function
	CAUSE	There was an attempt to call a C++ inlined function from the debugger command line.
	ACTION	If such a function needs to be debugged, recompile your program with the <code>+d</code> option. This will cause the compiler to force a non-inlined version of the function to be emitted. This function can then be debugged regularly.
UE762	MESSAGE	Cannot set instance breakpoint on static member function
	CAUSE	There was an attempt to set an instance breakpoint on a static member function.
	ACTION	Use the regular <code>b (breakpoint)</code> command on static member functions.
UE763	MESSAGE	Class has only static member functions
	CAUSE	Use of the <code>bi (breakpoint instance)</code> command on a class which has only static member functions.
	ACTION	Use the <code>bpc (breakpoint class)</code> command instead.
UE764	MESSAGE	Breakpoints set only for non-static member functions
	CAUSE	The <code>bi (breakpoint instance)</code> command was used on a class that has static member functions. No breakpoint was set on the static member functions.
	ACTION	If you need to set a breakpoint on all the members, use the <code>bpc (breakpoint class)</code> command instead. Alternatively, use the <code>bi (breakpoint instance)</code> command so that you get instance breakpoints on the regular members, and set regular breakpoints on static members.

A

UE765	MESSAGE	Pure virtual function in expression not supported
	CAUSE	A function used in an expression is a pure virtual function.
	ACTION	Use the function from a derived object instead.
UE766	MESSAGE	Calls via function expressions not supported
	CAUSE	An expression contains a call to a member function through a member function pointer.
	ACTION	This is not supported.
UE767	MESSAGE	Function calls returning class objects are not supported
	CAUSE	An expression contains a call to a member function whose return type is a class object.
	ACTION	This is not supported.
UE768	MESSAGE	Warning: constructors will not be implicitly executed
	CAUSE	An expression contains a call to a member function and the process being debugged has not been started yet or has died. If there are any static objects in your program, their constructors will not be called before the function is called.
	ACTION	If the member function you want to call accesses any static objects, you need to start the child process first (use the <code>s</code> or <code>S</code> command).

**A**

---

UE769	MESSAGE	Inlined function in expression not supported
	CAUSE	The name of an inlined C++ function has been used in an expression.
	ACTION	To be able to use this function, your program must be recompiled with the +d option. This will cause the compiler to force a non-inlined version of the function to be emitted.

---

UE770	MESSAGE	No breakpoint set; all functions have been inlined
	CAUSE	There was an attempt to set an instance or class breakpoint (bi or bpc) on a class whose member functions have all been inlined.
	ACTION	Recompile your C++ program with the +d option. This will prevent the compiler from inlining member functions.

---

UE771	MESSAGE	Cannot resolve overloaded function "PROC" while running assertions
	CAUSE	A command used in an assertion command list involves overloaded functions. Usually the debugger presents a menu of functions and asks you for your choice to resolve ambiguities. This is not possible from inside an assertion.
	ACTION	If you are trying to set a breakpoint on an overloaded function, set the breakpoint at a line number so that there is no possible ambiguity.

A

UE772	MESSAGE	Cannot resolve overloaded function " <i>PROC</i> " when executing breakpoint
	CAUSE	A command used in a breakpoint command list involves overloaded functions. Usually the debugger presents a menu of functions and asks you for your choice to resolve ambiguities. This is not possible from inside a breakpoint.
	ACTION	If you are trying to set a breakpoint on an overloaded function, set the breakpoint at a line number so that there is no possible ambiguity.
UE773	MESSAGE	Class object parameters are not supported in command line function calls
	CAUSE	An expression contains a call to a function that has an argument that is a class object.
	ACTION	This is not supported.
UE774	MESSAGE	Unsupported member pointer assignment
	CAUSE	There was an attempt to assign the address of a member function to a pointer to a member function of a different class.
	ACTION	This is not supported.
UE775	MESSAGE	Function calls returning a pointer to a member function are not supported
	CAUSE	An expression contains a call to a member function whose return type is a pointer to member function.
	ACTION	This is not supported.

A



---

UE776	MESSAGE	Parameter of type pointer to member function not supported in command line call
	CAUSE	There was an attempt to call a function that has an argument which is a pointer to member function.
	ACTION	This is not supported.

---

UE779	MESSAGE	Breakpoint command processing overflow
	CAUSE	A macro processing overflow occurred while evaluating a breakpoint command list (see UE451).
	ACTION	Shorten the macro being processed, or manually substitute the reference of the macro in the breakpoint command-list with the actual command.

---

UE782	MESSAGE	Invalid argument on "continue" command
	CAUSE	The <code>continue</code> command was given with a location the debugger could not evaluate or apply in this context (such as a filename).
	ACTION	Replace the location specified with valid location (line number, procedure name, label).

A

---

UE783	MESSAGE	Invalid argument on "Continue" command
	CAUSE	The <code>Continue</code> command was given with a location the debugger could not evaluate or apply in this context (such as a filename).
	ACTION	Replace the location specified with valid location (line number, procedure name, label).

UE784	MESSAGE	Invalid argument on " <i>CMD</i> " command
	CAUSE	The <i>CMD</i> command was given with a location the debugger could not evaluate or apply in this context (such as a filename).
	ACTION	Replace the location specified with valid location (line number, procedure name, label).
UE785	MESSAGE	Address is required after "va"
	CAUSE	The va command was entered with no parameter.
	ACTION	Re-enter the command with an address argument.
UE786	MESSAGE	Unrecognized option
	CAUSE	An unrecognized option was given to a bi (breakpoint instance) or bpc (breakpoint class) command.
	ACTION	Consult Chapter 4 in this manual for valid options to be used with the bi (breakpoint instance) or bpc (breakpoint class) commands.
UE830	MESSAGE	Count ignored on break on template member functions.
	CAUSE	A count was specified with a breakpoint set on a template member function and has been ignored. Setting a count on breakpoints on template member functions is not supported.
	ACTION	No action is needed.

A

UE835	MESSAGE	Cannot restore PC space register to continue
	CAUSE	The program was linked with an older version of <code>/usr/lib/end.o</code> , and the debugger does not have the ability to use hooks in that file to restore the PCSQ register after a command-line procedure call.
	ACTION	Relink the program using a current version of <code>/usr/lib/end.o</code> , and re-invoke the debugger.
UE836	MESSAGE	WARNING: <code>/usr/lib/end.o</code> was not linked with this program
	CAUSE	This file must be linked with your program for the debugger to support many operations.
	ACTION	If the linker ( <code>ld(1)</code> ) was used to link you program, explicitly list <code>/usr/lib/end.o</code> on the linker command-line. If a compiler was used for the final link, make sure <code>-g</code> is use. If the message persists, make sure <code>/usr/lib/end.o</code> is installed on your system.
UE837	MESSAGE	Shared-library debugging cannot be made available
	CAUSE	The file <code>/usr/lib/end.o</code> must be linked with your program for the debugger to effectively support debugging of shared-libraries or programs that use them.
	ACTION	See UE836.

A

---

UE838	MESSAGE	WARNING: Cannot find 'main' entry-point in <i>PROGRAM</i>
	CAUSE	The debugger could not locate the symbol which marks the "start" address of the program.
	ACTION	Make sure the program was linked with the standard /lib/crt0.o (for Series 300/400 FORTRAN use /lib/frt0.o). The linker -v option can provide this information.

---

UE839	MESSAGE	WARNING: Enclosing procedure not on stack
	CAUSE	A command-line procedure call was made to a Pascal procedure which is scoped (nested) within another procedure, which is not currently active on the execution stack. The debugger cannot construct a <i>static link</i> for the procedure being called.
	ACTION	If this error occurs, the debugger will prompt for continuation. Answering yes will cause a static-link of 0 will be used. Otherwise, make sure the outer procedure is active on the stack before attempting to call any procedure nested within it.

---

UE841	MESSAGE	Invalid \$lang value. Resetting to Default
	CAUSE	The \$lang special variable was set to an invalid value, perhaps by an action such as p \$lang=C where C is also an identifier in the program being debugged.
	ACTION	Reset \$lang to a valid value. Choices are C (0), FORTRAN (1), Pascal (2), C++ (4), and default. If the desired language name coincides with a program identifier, the numerical values may be used instead.

A

UE842	MESSAGE	WARNING: Cannot locate main entry point.
	CAUSE	The main program body (e.g. <code>main()</code> for the C language) cannot be located by the debugger. Presumably it is within a shared-library for which the <code>-l</code> option is not specified.
	ACTION	Re-invoke the debugger with <code>-l libname</code> , where <i>libname</i> identifies the shared-library containing the main program body.
UE843	MESSAGE	WARNING: Can't set breakpoint at main entry; try invoking with <code>-s</code> .
	CAUSE	The main program body (e.g. <code>main()</code> for the C language) is within a shared-library, but the debugger cannot set a breakpoint there because the library is mapped into the process as read-only.
	ACTION	Re-invoke the debugger with the <code>-s</code> or <code>-l</code> option.
UE846	MESSAGE	Corefile created on older system.
	CAUSE	An attempt was made to use a corefile that was created on an older version of HP-UX which does not support shared-library corefiles. Consequently, if the program aborted in a shared-library, the debugger cannot convert the actual address where the program aborted to a symbolic address, and the current shared-library <i>load map</i> at the time the program aborted cannot be reconstructed.
	ACTION	Examine the corefile on the system on which it was created, or attempt to create a valid corefile by running the program on a newer version of HP-UX. Make sure the program is linked with current versions of <code>/lib/crt0.o</code> (for Series 300/400 FORTRAN use <code>/lib/frt0.o</code> ) and <code>/usr/lib/end.o</code> (see also UE436, UE836, and UE838).

A

---

UE847      MESSAGE      Program linked on older system.

            CAUSE          (Series 300/400 only) A corefile was used, but the program being debugged was linked with an old version of */bin/ld*, and the debugger cannot determine the shared-library *load map* at the time the program aborted.

            ACTION        Relink the program using a current linker, as well as current versions of the files */lib/crt0.o* (for Series 300/400 FORTRAN use */lib/frt0.o*) and */usr/lib/end.o*. Then attempt to re-create the corefile on a current version of HP-UX.

---

UE848      MESSAGE      Program linked with older version of *OBJFILE*.

            CAUSE          (Series 600/700/800 only) A corefile was used, but the program being debugged was linked with an old version of either */lib/crt0.o* (for Series 300/400 FORTRAN use */lib/frt0.o*) or */usr/lib/end.o*, and the debugger cannot determine the shared-library *load map* at the time the program aborted.

            ACTION        Relink the program using current versions of the files */lib/crt0.o* and */usr/lib/end.o*. Then attempt to re-create the corefile on a current version of HP-UX.

---

UE850      MESSAGE      No valid entry for *PROC*

            CAUSE          A command-line procedure-call to *PROC* was attempted (*PROC* was not compiled with -g). No callable address for *PROC* was found.

            ACTION        Use the *ll PROC (list labels)* command to verify the symbol-type and location of *PROC*.

A

UE851	MESSAGE	<i>PROC</i> is not TYPE_PROCEDURE
	CAUSE	A command-line procedure-call to <i>PROC</i> was attempted ( <i>PROC</i> was not compiled with -g). <i>PROC</i> is a dynamic symbol, but was not defined as TYPE_PROCEDURE (see <i>shl_definesym(3X)</i> ).
	ACTION	Use the <code>ll PROC (list labels)</code> command to verify the symbol-type of <i>PROC</i> .
UE852	MESSAGE	Dynamic-loader cannot locate <i>PROC</i>
	CAUSE	A command-line procedure call was attempted to a procedure which the dynamic-loader cannot locate. The debugger uses <i>shl_findsym(3X)</i> to properly bind procedures called on the command-line.
	ACTION	Make sure the procedure is spelled correctly, and that it is defined in an active shared library. The <code>list procedures</code> or <code>list labels</code> command can help.
UE854	MESSAGE	No such shared-library <i>NAME</i>
	CAUSE	A reference was made to a symbol using @ qualification (e.g. <i>symbol@NAME</i> ), and the debugger cannot properly identify the shared-library referred to by <i>NAME</i> .
	ACTION	Use the <code>ls1</code> command to list all shared-libraries and the abbreviated names that are legal in symbol references. If the shared library denoted by <i>NAME</i> was not linked with the program, but is expected to be loaded with <i>shl_load(3X)</i> , use the <code>-l</code> invocation option to specify the library.

A

UE856	MESSAGE	syntax: apm {oldpath   \"\\"} [newpath]
	CAUSE	The apm command was given without any arguments.
	ACTION	Supply the proper arguments for the desired mapping.
UE857	MESSAGE	path map ignored (2 empty paths)
	CAUSE	The command apm "" or apm "" "" was given.
	ACTION	Supply an apm command with at least one non-empty path.
UE858	MESSAGE	Path substitution list is empty
	CAUSE	The dpm command was given when no path maps have been defined with the apm command.
	ACTION	No action is needed.
UE859	MESSAGE	Path substitution stack is not that deep
	CAUSE	The dpm command was given with an argument which is greater than the number of path maps currently defined.
	ACTION	Use the lpm command to see which and how many path maps are currently defined.
UE1026	MESSAGE	<i>SHARED-LIBRARY</i> is not an active shared-library
	CAUSE	The user attempted an mm @shared-library command, and the named library is not currently mapped into memory.
	ACTION	No action is needed.

A



---

UE1030	MESSAGE	WARNING: <i>OBJFILE</i> does not contain the required support for <i>-s</i> .
	CAUSE	The program being debugged was linked with an old version of either <i>/lib/crt0.o</i> (for Series 300/400 FORTRAN use <i>/lib/frt0.o</i> ) or <i>/usr/lib/end.o</i> , and debugging of shared-libraries cannot be supported.
	ACTION	Relink the program using current versions of the files <i>/lib/crt0.o</i> and <i>/usr/lib/end.o</i> , and re-invoke the debugger.

---

UE1031	MESSAGE	Cannot set breakpoint at <i>ADDRESS</i> in <i>SHARED-LIBRARY</i> . Try invoking with <i>-lSHARED-LIBRARY</i> or <i>-s</i> .
	CAUSE	No <i>-s</i> or <i>-l</i> command-line option was given to the debugger, resulting in <i>SHARED-LIBRARY</i> being loaded as truly shared (read-only).
	ACTION	Re-run the debugger with the <i>-s</i> or <i>-l</i> command-line option.

---

A UE1042	MESSAGE	Ignoring <i>-s</i> or <i>-l</i> option. You must invoke the command: " <i>pxdb -s enable PROGRAM</i> " before executing <i>PROGRAM</i> .
	CAUSE	An attempt was made to adopt <i>PROGRAM</i> using <i>xdb -P</i> , but the flags that cause shared-libraries to be loaded as writable were not properly pre-set. Consequently, setting a breakpoint in any shared-library loaded by <i>PROGRAM</i> is disallowed.
	ACTION	Run the given command before executing <i>PROGRAM</i> . This statically sets the appropriate flags in the <i>a.out</i> file, causing the dynamic-loader to load private (writable) images of all shared-libraries used by <i>PROGRAM</i> .

UE1044	MESSAGE	Warning: Cannot locate dependent library: <i>SHARED-LIBRARY</i>
	CAUSE	The program being debugged was linked against a shared-library which the debugger cannot locate using information available to it in the executable itself.
	ACTION	Use the <i>-l</i> option to specify the complete path to the debugger.
UE1045	MESSAGE	Warning: Cannot locate library for <i>-lNAME</i>
	CAUSE	An abbreviated shared-library name was used with the <i>-l</i> option, but the debugger cannot locate the library using information available to it in the executable itself.
	ACTION	Re-issue the <i>-l</i> option with a complete path.
UE2003	MESSAGE	Warning: PA-RISC 1.1 executable on PA-RISC 1.0 system.
	CAUSE	The program being debugged was compiled on a PA-RISC 1.1 system or with the <i>+DA1.1</i> compiler option, and is being debugged on a PA-RISC 1.0 system. The debugger will abort with a UE375.
	ACTION	Recompile the program with <i>+DA1.0</i> , or debug it on a PA-RISC 1.0 system.
UE2004	MESSAGE	Executables linked with <i>-N</i> are not supported on Series 600/800
	CAUSE	The <i>-N</i> option to <i>ld(1)</i> was used when the program was linked. This option is only supported on Series 300/400/700.
	ACTION	Either link the object file without the <i>-N</i> option or debug the program on a Series 700.

A

---

UE2007	MESSAGE	Warning: no such shared-library <i>SHARED-LIBRARY</i>
	CAUSE	The program was linked with, or a corefile indicates that the program was executed with, a shared-library which cannot be located by the debugger.
	ACTION	Make sure the program will execute stand-alone (outside the debugger), or if a corefile was used, make sure all shared-libraries in use by the program when it aborted are available on the current system.

---

UE2011	MESSAGE	WARNING: Shared-library linked more than once, ignoring duplicate: <i>SHARED-LIBRARY</i>
	CAUSE	(Series 300/400 only) The program was linked with more than one copy of a shared-library. Most likely, a language's default library was inadvertently listed on the compiler command-line (For example: <code>cc ... -lc</code> ). The debugger can only accommodate the first one seen.
	ACTION	Make sure that each shared-library used by the program is only linked in once.

---

<b>A</b>	UE2012	MESSAGE	WARNING: ignoring <code>shl_load()</code> of library that is already loaded: <i>SHARED-LIBRARY</i>
		CAUSE	(Series 300/400 only) The program being debugged did a <code>shl_load(3X)</code> of a shared-library that had either been implicitly linked in, or previously <code>shl_load</code> 'ed. The debugger can only accommodate the library already loaded.
		ACTION	Make sure that each shared-library used by the program is only linked in once, and any library implicitly linked in is not <code>shl_load</code> 'ed.

UE2017	MESSAGE	Too many debuggable shared-libraries
	CAUSE	The program was linked with more than 512 shared-libraries that contain symbolic debug information, and the user requested that all such libraries be debugged.
	ACTION	Don't use the <code>-lALL</code> invocation option, or limit the number of <code>-llibname</code> arguments given to the debugger.
UE2021	MESSAGE	Cannot bind unbind address <i>ADDRESS</i> (symbol <i>NAME</i> ) to inactive image <i>SHARED-LIBRARY</i>
	CAUSE	A reference was made to an address which was mapped to shared-library which is no longer active (mapped into the process).
	ACTION	Verify the symbol or address is correctly specified, and reissue the command. The <code>mm</code> (memory map) command can be used to list all active images.
UE2029	MESSAGE	WARNING: <i>NAME</i> debug table(s) in <i>FILE</i> are too large
	CAUSE	The program or shared-library <i>FILE</i> contains more symbolic-debug information than the debugger can handle from a single file.
	ACTION	Compile all/some portions of <i>FILE</i> without the <code>-g</code> compiler option, or split the shared-library into multiple libraries.

**A**

---

UE2031      MESSAGE      Image not active: *SHARED-LIBRARY*

             CAUSE        Attempt to print a variable or call a function from  
                             the command line where the variable or the function  
                             is in a shared library that has been unloaded by the  
                             program.

             ACTION       Print the variable or call the function before the  
                             program unloads the shared library.

**A**

## **HP C and C++ Language Operators**

---

This appendix lists and describes operators for the HP C and C++ programming languages that the debugger expression evaluator recognizes.

---

### **HP C and C++ Language Operators**

The following table lists the supported HP C and C++ operators. Operators are listed in order of precedence, from highest to lowest. All operators listed in the same box are of equal precedence. Associativity of operators in the following table is from left to right, unless otherwise stated. Assignment is treated by the debugger as an operation.

For HP C and C++, the operators `&&` and `||` are not short circuited as is done by the compilers; all portions of an expression involving these operators are evaluated. Also, pointer arithmetic in the debugger is unsupported.

Full support of *struct* and *class* objects is provided.

## HP C and C++ Language Operators

**Table B-1. Language Operators for HP C and C++**

Operator	Operation
::	scope resolution operator (C++ only)
( )	parenthesis (group elements)
[ ]	array member selection
->	member selection of pointer to structure
.	member selection of structure
! (order is right to left)	unary logical negation
~ (order is right to left)	unary logical one's complement
- (order is right to left)	unary negation
* (order is right to left)	unary indirection (pointer or address dereferencing)
& (order is right to left)	unary address of an object
\$addr (order is right to left)	unary address of an object
\$sizeof (order is right to left)	unary size of an object
\$in (order is right to left)	unary suspended in named routine
sizeof (order is right to left)	unary size of an object
*	multiplication
/	division
%	modulus - mod function
+	addition
-	subtraction

B

**Table B-1. Language Operators for HP C and C++ (continued)**

Operator	Operation
<<	bit-wise logical left shift; fill with 0
>>	bit-wise arithmetic right shift; unsigned fill with 0, else fill with sign bit
<	relational less than
<=	relational less than or equal to
>	relational greater than
>=	relational greater than or equal to
==	relational equal to
!=	relational not equal to
&	bit-wise logical and
^	bit-wise logical exclusive or
	bit-wise logical inclusive or
&&	logical and
	logical or
= (order is right to left) op= (order is right to left)	assignment assignment operators of the form: e1 op= e2 which means (e1) = (e1) op (e2). Where op may be any one of the mathematical or bit-wise operators (*, /, %, +, -, <<, >>, &, ^,  )

**B**





## HP FORTRAN 77 Language Operators and VMS Record Support

---

This appendix lists and describes operators for the HP FORTRAN 77 programming language that the debugger expression evaluator recognizes.

---

### HP FORTRAN 77 Language Operators

The following table lists the supported HP FORTRAN 77 operators. Operators are listed in order of precedence, from highest to lowest. All operators listed in the same box are of equal precedence. All operators of equal precedence evaluate left to right, unless otherwise stated. Assignment is treated by the debugger as an operator.

Complex variables in HP FORTRAN 77 are not supported except as a pair of two separate reals or doubles. Any HP C language operators that do not clash with supported HP FORTRAN 77 operators can be used in HP FORTRAN 77 expressions, with the corresponding C interpretation. The only exception to this is the unary operator `sizeof`.

## HP FORTRAN 77 Language Operators

Table C-1. Language Operators for HP FORTRAN 77

Operator	Operation
( )	parentheses (grouping), array member selection
.	member selection of record
- (order is right to left)	unary negation
\$addr (order is right to left)	unary address of an object
\$sizeof (order is right to left)	unary size of an object
\$in (order is right to left)	unary suspended in named routine
*	multiplication
/	division
+	addition
-	subtraction
.LT.	relational less than
.LE.	relational less than or equal to
.EQ.	relational equal to
.GE.	relational greater than or equal to
.NE.	relational not equal to
.GT.	relational greater than
.NOT.	logical negation
.AND.	logical and
.OR.	logical or

C

**Table C-1. Language Operators for HP FORTRAN 77 (continued)**

Operator	Operation
.EQV.	logical equivalence
.NEQV.	logical nonequivalence
= (order is right to left)	assignment

## VMS FORTRAN Records

HP Symbolic Debugger provides support for VMS FORTRAN records. There are four associated types:

- structures
- records
- unions
- maps

A *structure* defines record field types, as in the following example:

```
structure /date/
  integer a
  union
    map
      integer b
      real c
      character*8 d
      integer e
    union
      map
        logical f
        integer g
      end map
      map
        character*3 h
      end map
      map
        real i
      end map
    end union
  end map
end union
real j
integer f
end structure
```

C

A *record* corresponds to an instance of that record structure.

For example, given the previous structure, you can define a record with that structure:

```
record /date/ rec1
```

In HP Symbolic Debugger, HP FORTRAN 77 *records* are treated as HP FORTRAN 77 *structures* from the debugger. This means that if you use the `print` command with the `\t` format to look at a record, you will see the record's *structure* rather than the record definition, `record /date/ rec1`.

For example, if you type:

```
>p rec1\t
```

you will get:

```
structure /date/
  integer a
  union
    map
      integer b
      real c
      character*8 d
      integer e
    union
      map
        logical f
        integer g
      end map
      map
        character*3 h
      end map
      map
        real i
      end map
    end union
  end union
```

C

```
        end map
    end union
    real j
    integer f
end structure rec1
```

You can access any element within a record. Because maps and unions are unnamed, they are ignored in naming subelements. For example, field `h` in the previous example must be accessed as:

```
rec1.h
```

If there is any ambiguity among field names, the first one appearing by a given name is chosen, just as it is in HP FORTRAN 77. For example, field `rec1.f` in the example above is of type *logical*, not integer.

When the value or type of any field in a record is displayed, its individual format is identical to what it would be if it were not within a record. For the records, unions, and maps themselves, these keywords are used identically to the way they are used in HP FORTRAN 77 except:

- When printing the type of a structure, its name will follow the entire structure instead of preceding it.

For example:

```
>p rec\t
```

gives you this:

```
structure
  integer*4 i
end structure rec
```

- When printing the value of a structure, its name and an equal sign (=) precede its value.

For example:

```
>p rec
```

gives you this:

```
rec = structure  
  i = 3  
end structure
```





# D

## HP Pascal Language Operators

---

This appendix lists and describes operators for the HP Pascal programming language that the debugger expression evaluator recognizes.

---

### HP Pascal Language Operators

The following table lists the supported HP Pascal operators. Operators are listed in order of precedence, from highest to lowest. All operators listed in the same box are of equal precedence. All operators of equal precedence evaluate left to right, unless otherwise stated. Assignment is treated by the debugger as an operator.

Any HP C language operators that do not clash with supported HP Pascal operators can be used in HP Pascal expressions, with the corresponding C interpretation.

There are two restrictions with the language operators for HP Pascal:

- Variables qualified by the **WITH** statement in an HP Pascal program must be fully qualified in HP Symbolic Debugger expressions. The HP Pascal **WITH** construct is not recognized as a debugger command.
- The debugger does not support HP Pascal set constants and does not support operations on sets.

## HP Pascal Language Operators

Table D-1. Language Operators for HP Pascal

Operator	Operation
( )	parenthesis, group elements
[ ]	array member selection
.	member selection of record
^ (order is right to left)	pointer (address) dereferencing
not (order is right to left)	unary logical negation
addr	unary address of an object <sup>1</sup>
\$addr (order is right to left)	unary address of an object
\$sizeof (order is right to left)	unary size of an object
\$in (order is right to left)	unary suspended in named routine
sizeof (order is right to left)	unary size of an object
*	multiplication
/	real division
div	integer division with truncation
mod	modulus
+	addition
-	subtraction

<sup>1</sup> The debugger does not allow Pascal's optional second argument to `addr`.



**Table D-1. Language Operators for HP Pascal (continued)**

<b>Operator</b>	<b>Operation</b>
<	relational less than
>	relational greater than
<=	relational less than or equal to
>=	relational greater than or equal to
=	relational equal to
< >	relational not equal to
:= (order is right to left)	assignment
and	logical and
or	logical or



## Special and Environment Variables Used by the Symbolic Debugger

---

This appendix covers special variables and environment variables (that affect the behavior of the debugger).

### Special Variables

Table E-1. Special Variables

Variable	Description
<code>\$var</code>	Creates or references user-defined variables. User-created special variables are of type <code>long</code> , and their names are defined when they are first used. The variable names are limited to 100 characters.
<code>\$pc</code> , <code>\$sp</code> , <code>\$r7</code> , etc.	These are the names of the program counter, the stack pointer, the CPU general registers, etc. (see the appendix "Registers Displayed by the HP Symbolic Debugger in Disassembly Mode"). All registers act as type <code>integer</code> .
<code>\$fpa</code>	If this variable is set to a non-zero value, any sequence of machine instructions that constitute a single floating-point accelerator instruction will be treated as a single instruction for machine-level single-stepping and display (Series 300 only).

Table E-1. Special Variables (continued)

Variable	Description
<code>\$fpa_reg</code>	Indicates the address register used in floating point accelerator instruction sequences if <code>\$fpa</code> is set to a non-zero value (Series 300 only). A 0 corresponds to register a0, 1 to a1, etc. The default value is 2.
<code>\$result</code>	References the return value from the last command-line procedure call. Note that <code>\$short</code> and <code>\$long</code> are available as alternate ways of looking at <code>\$result</code> .
<code>\$signal</code>	Contains the current child process signal number (can be modified).
<code>\$lang</code>	Contains the current language (can be modified). The current language determines the operators that can be used in expressions, and the format in which variables are displayed.
<code>\$print</code>	Alters the behavior of the <code>print</code> command when printing character data. Values are <code>ASCII</code> , <code>native</code> , and <code>raw</code> . Default is <code>ASCII</code> .
<code>\$line</code>	Contains the current source line number, which is also settable with a number of different commands.
<code>\$malloc</code>	Allows you to see the current amount of memory (bytes) allocated at run-time for use by the debugger itself.
<code>\$step</code>	Contains the number of machine instructions the debugger will step while in a non-debuggable procedure before setting an up-level breakpoint and free-running to it (can be modified). The number of machine instructions the debugger will step by default for Series 300/400 computers is 12 and for Series 600/700/800 computers is 24.
<code>\$cplusplus</code>	A set of flags to control behavior of certain C++ capabilities. For information on this special variable, read the section "Customizing Default Debugger Behavior" in the chapter "C++ and the Symbolic Debugger."

**Table E-1. Special Variables (continued)**

Variable	Description
<b>\$depth</b>	Contains the default stack level for viewing local variables. It is set by the <b>V</b> , <b>up</b> , <b>down</b> , and <b>top</b> commands. It is reset to 0 (top of the stack) by the commands <b>r</b> , <b>R</b> , <b>c</b> , <b>C</b> , <b>s</b> , <b>S</b> , <b>g</b> , and <b>k</b> and adjusted by the <b>tst</b> command (PA-RISC only). Higher depth numbers correspond to procedures further down the stack (greater stack depth). Setting this variable directly ( <b>p \$depth = n</b> ) sets the local context to the specified depth, but does not update the source window.



Table E-2. Environment Variables

Variable Name	Description	Default <sup>1</sup>
Display Interface:		
TERM	Terminal type	none <sup>2</sup>
LINES	Terminal or window height	\$TERM, 24 <sup>3</sup>
COLUMNS	Terminal or window width	\$TERM, 80 <sup>3</sup>
Command-line editing (see <i>ksh(1)</i> ):		
XDBHIST	History file	\$HOME/.xdbhist
HISTSIZE	Maximum commands in history	128
XDBEDIT	Editing mode ( <i>vi</i> , <i>emacs</i> , <i>gmacs</i> )	\$VISUAL, \$EDITOR, none
Native Language Support:		
LANG	Locale for message	"C"
LC_TYPE	Locale for interpreting textual data	\$LANG, "C"

1 If alternate defaults are listed, they are checked in the order given.

2 If TERM is not set, the debugger uses "dumb" mode (equivalent to the -L command-line option).

3 In an X Window, if LINES and COLUMNS are not set, values are taken from the window. Otherwise, if TERM is set, LINES and COLUMNS may be determined by the `terminfo` entry for the terminal type given.

## Limitations and Hints

---

This appendix lists some limitations of HP Symbolic Debugger and gives some hints for debugger usage.

---

## Limitations and Hints

### Source Limitations

- Code that is not compiled debuggable or does not have a corresponding source file is dealt with in a limited manner. The debugger shows “unknown” for unknown file and procedure names, cannot show source or interpret parameter lists, etc. However, the linker symbol table (viewable with the `ll (list labels)` command) provides procedure names for most procedures, even if they are not debuggable (see the section “Disassembly Mode Limitations”).
- Some compilers only issue source line symbols at the end of each logical statement or physical line, whichever is greater. This means that, if you are accustomed to saying `a = 0; b = 1;` on one line, you cannot set a breakpoint after the assignment to `a` and before the assignment to `b`.
- Some statements do not emit code where you would expect it. For example, `assume:`

```
99:   for (i = 0; i < 9; i++) {  
100:       xyz (i);  
101:   }
```

A breakpoint placed on line 99 will be hit only once in some cases. The code for incrementing is placed at line 101. Each compiler is a little different; you must get used to what your particular compiler does. A good way of

finding out is to use single stepping to see in what order the source lines are executed.

- The output of some program generators, such as *yacc(1)*, have compiler line number directives in them that can confuse the debugger. It expects source line entries in the symbol table to appear in sorted order. Removal of line directives fixes the problem, but makes it more difficult to find error locations in the original source file. The following script, run after *yacc(1)* and before *cc(1)*, comments out line number changes in C programs:

```
sed "/# *line/s/^.*$/\/*&*\\/" y.tab.c >temp.c
```

The *yacc(1)* command will leave out line directives if invoked with the *-l* option. In general, line number directives (or compiler options) are safe so long as they never emit line number directives out of sequence.

## Process Limitations

- The debugger will not be usable on systems that have been booted from something other than */hp-ux* (for example, *SYSBCKUP* was booted instead). Note that this applies only to Series 300/400 computers.
- The debugger has no knowledge about, or control over, child processes forked in turn by the process being debugged. Programs being debugged should not execute a different program via *exec(2)*.
- Child process output may be (and usually is) buffered. Hence it may not appear immediately after you step through an output statement such as *printf(3S)*. It may not appear at all if you kill the process.
- If the address given to a *ba* command is not a code address in the child process, strange results or errors may ensue.
- If you single step or run with assertions through a call to *longjmp()* (see *setjmp(3C)*), the child process will probably take off free-running as the debugger sets but never hits an up-level breakpoint.
- Programs which use the *set-user-ID* facility do not have that capability when run under the debugger, as setting of the *effective-user-ID* is disabled when executing a traced process. Such programs can be debugged via adoption by superuser. See *ptrace(2)* and *set-user-ID* under *glossary(9)* in the *HP-UX Reference* for more information.

## Single Step Limitations

- The default value of `$step` may be insufficient for some applications. Note that large values for `$step` may impact single-step performance when stepping into non-debuggable procedure calls.
- (Series 600/700/800 only) The `S (Step)` command can be used to step over procedure calls in disassembly mode *only* when used at a branching instruction (for example, `BLE`). The current location after performing the step will be the first instruction following the delay slot instruction (the second instruction following the branch). The `S` command behaves like `s` when used at the delay slot instruction, and the call is stepped into.

## Signals Restrictions

- The debugger does not terminate on an interrupt (`SIGINT`); instead it jumps to its main loop and awaits another command. However, this does not imply that sending the debugger an interrupt is harmless. It can result in internal tables being left in an inconsistent state that could produce incorrect behavior.
- Do not use the `z` command to manipulate the `SIGTRAP` signal. This signal is used by the debugger to synchronize with and control the traced process, and unpredictable results may occur if it is manipulated in a different manner. A result of this is that applications that make use of the `SIGTRAP` signal will at best be difficult to debug.

## Operators Limitations

- The C operators `++`, `--`, and `?:` are not available. The debugger always understands all the other C operators, except `sizeof` if the default language is FORTRAN. Users should use `$sizeof` which works in any language.

The C operators `&&` and `||` are not short-circuit-evaluated as in the compiler. All parts of expressions involving them are evaluated, with any side-effects, even if it's not necessary.

The debugger does not understand C pointer arithmetic. `*(a+n)` is not the same as `a[n]` unless `a` has an element size of 1.

- The only operations that are allowed on entire C++ class objects during expression evaluation are taking the address of them (with the `&` operator) and taking the size of them (with the `sizeof` or `$sizeof` operators.)

## F Object Type Limitations

- Assignments from debugger special variables into objects greater than four bytes in size will give invalid results.
- When you try to display a variable which is a FORTRAN format label, a Pascal file-of-text, or a Pascal set, with no display format or with normal format (`\n`), the value is shown as `{format-label}`, `{file-of-text}`, or `{set}`, respectively. You can use other formats, such as `\x`, to display the contents of such variables.
- When a C parameter is declared as an array of anything, the highest type qualifier (array) shows up as a pointer instead. For example, `int x[]` looks like `int *x`, and `char (*x)[]` looks like `char **x`, but `char *x[]` is treated correctly as “pointer to array of char”.

When a compiler does not know array dimensions, such as for some C and FORTRAN array parameters, it uses `0:MAXINT` or `1:MAXINT`, as appropriate. The `\t` format shows such cases with `[ ]` (no bounds specified), and subscripts from 0 (or 1) to `MAXINT` are allowed in expressions.

There is no support for Pascal packed arrays where the element size is not a whole number of bytes. Any reference into such an array may produce garbage or a bad access error.

- The debugger does not know about `void` as a type. All objects of type `void` are reported as being of type `int`. After a command-line call to a function of type `void`, `$result` will contain a meaningless value.
- The debugger interprets `COMPLEX` variables to be of type `REAL`; therefore, when you try to print the value of a `COMPLEX` variable from within the debugger, a `REAL` value is displayed. To print both the real and imaginary parts of a `COMPLEX` variable, you need to enter a command with the following syntax:

```
p variable\2n
```

The following program (`mytest.f`) prints the real and imaginary parts of the `COMPLEX` variable `cplx8`.

```
PROGRAM main ()

COMPLEX*8 cplx8

cplx8 = (3.5, 5.4)
PRINT *, cplx8

STOP
END
```

To test the previously given print syntax, compile the above program as follows:

```
f77 -g -o mytest mytest.f
```

Next, execute the debugger command with the program name `mytest` as follows:

```
xdb mytest
```

Set a breakpoint at line 6 in the program:

```
>b 6\t
```

and run the program in the debugger using this command:

```
>r
```

To display the real and imaginary parts of the `COMPLEX` variable `cplx8`, execute this command:

```
>p cplx8\2n
```

Results displayed are similar to this:

```
0x7b0333d0          3.5          5.4
```

To change the values of the the `COMPLEX` variable `cplx8`, you would enter and execute commands similar to this:

```
>p cplx8=2.7
cplx8 = 2.7
>p *($addr(cplx8)+$sizeof(cplx8))=7.6
```

```
0x7b0333d4 7.6
>p cplx8\2n
0x7b0333d0          2.7          7.6
```

To view the values with an alternate floating-point format, you must take into consideration the size of the data items:

```
>p cplx8\2e
0x7b0333d0          3.500000e+00  5.400000e+00
```

Note that the “E” format will not work here.

- Two types of string formats are supported in addition to null-terminated C strings. FORTRAN character variables consist of a string of bytes (no null terminator). Pascal string variables consist of a length byte, followed by the string characters. The `\s` and `\a` formats will display these types correctly only if the current language is FORTRAN or Pascal.

## Files Restrictions

- Do not modify any file while the debugger has it open. If you do, the debugger gets confused and may display garbage.
- Although the debugger tries to do things reasonably, it is possible to confuse the recording mechanism. Be careful about trying to play back from a file currently open for recording, or vice versa; strange things can happen.
- Command lines longer than 1024 bytes are broken into pieces of that size. This may be relevant if you run the debugger with playback or with input redirected from a file.

For backwards compatibility, a blank line in a record file is interpreted as ten lines when played back.

## Naming Restrictions

- The debugger does not support identically-named procedures except for overloaded functions in C++. In all other cases, it will always use the first procedure it finds with the given name. In Pascal, identically-named procedures are legal if the procedures are in different scopes and are referenced with the appropriate qualification.

- Pascal **WITH** statements are not understood. To access any variable you must specify the complete “path” to it.
- Case-insensitive searches are done in a crude way which equates some non-letters with other non-letters. For example, [ and { are equal, as are @ and ‘.
- Procedures in FORTRAN and Pascal may have alias names in addition to normal names. Aliases are shown by the **lp** (**list procedures**) command. They can be used in place of the normal name, as desired.

The procedure name **\_MAIN\_** is used as the alias name for the main program (main procedure) in all supported languages. Do not use it for any debuggable procedures.

FORTRAN “ENTRY” points are flagged **ENTRY** by the **lp** command.

- Some variables are indirect, so a child process must exist in order for the debugger to know their addresses. When there is no child process, the address of any such variable is shown as **0xffffffffe**.
- Symbol names in the debugger’s name table are never preceded by underscores, so the debugger never bothers to search for names of that form. The only time a prefixed underscore is expected is when searching the linker symbol table for names of non-debuggable procedures. (Series 300 only)
- There is no support for Pascal intermediate variables. To reference a variable local to an enclosing procedure, you must specify the procedure name and stack depth in the usual way (*proc:depth:var*).

## Command-Line Procedure Call Limitations

- The debugger supports call-by-reference only for known parameters of known (debuggable) procedures. You can fake such a call by passing **&object** (that is, the address of the object).
- Array parameters are always passed to command-line procedure calls by address. This is correct except for Pascal call-by-value parameters. Structure parameters are passed by address or value, as appropriate, but only a maximum of eight bytes is passed, which may totally confuse the called procedure.



- Functions which accept complex (real) arguments are not called correctly; only the first number of a complex pair is passed as a parameter. Functions which return complex numbers are not called correctly; insufficient stack space is allocated for the return area, which can lead to overwriting the parameter values.
- There is limited support for command-line calls of functions which return structures. The debugger interprets the start of heap as a structure of the return type. However, a call such as `abc()\t` displays the return type correctly.
- `$short` and `$long` are available in addition to `$result`. If a command-line procedure call returns a double, `$result` is set to the value cast to a long.

## Shared Library Limitations

- Use of the `-s` or `-l` option causes all shared libraries used by the application to be transparently loaded as private (unshared) copies. For large applications, this can significantly increase the amount of swap space allocated to the process. If the user only needs to debug the application, but not the libraries it requires, use of the `-s` or `-l` option is unnecessary.
- Only certain non-alphanumeric characters are allowed in the basenames of shared libraries which the user may wish to reference with an `@`-qualification:

. , : - ~ % ^ = +

Shared libraries with basenames containing any other non-alphanumeric characters cannot be referenced in the debugger.

- Programs which load the same shared library more than once, either by linking with `-l` or with `shl_load(3X)`, will cause the debugger to print a warning and ignore all but the first instance of the shared library. Consequently, the debugger cannot properly map addresses within a duplicate library to their symbolic counterparts.

(Series 300/400 only) Listing a language's default library on the compiler command line will cause that library to be linked and loaded twice, since the compiler will also specify that library when invoking the linker. For example:

```
CC ... -lc -ldld
```

will cause both `/usr/lib/libC.sl` and `/usr/lib/libdld.sl` to be linked and loaded twice, although only one of each would normally be accessible and usable by the program.

- The total number of shared libraries the debugger can debug at the source level is 512. The size of symbolic debug tables allowable in a program or shared library has a finite limit, although considerable. Support for debugging of shared libraries has caused these limits to be reduced somewhat. For example, the maximum number of source statements plus 3 times the number of procedures compiled with `-g` must be no greater than approximately 8.4 million for a given library or program. The previous limit was approximately 4.2 trillion.

The maximum number of debuggable procedures allowed in any one shared library is 32,767. This limit only applies to all-procedure breakpoints.

- (Series 600/700/800 only) Attempting to print a data item defined in a shared library, but for which debug info has not been loaded (wasn't listed with the debugger `-l` option), will show the item as an integer value. This is because the symbol also exists in the linker symbol table, has type `(const) int`, and its value is an address. Once symbolic debug info for the object has been loaded (i.e. through an explicit reference), the symbol will take on its proper type and value.
- Attempting to print a data item defined in both a shared library and the main program will show an incorrect value if:
  - The symbol was actually exported from the main program.
  - The main program was compiled non-debuggable (without `-g`).
  - Any library referencing (importing) the symbol is compiled debuggable (with `-g`) and the symbolic debug information for the library has been loaded.
- (Series 600/700/800 only) If a global object is defined in a shared-library which is unloaded with `shl_unload(3X)`, both the value and the type of the object will change as it is unloaded, since a symbol naming the object may still be present and visible in the linker symbol table. The type of such a symbol will be `(const) int`, and its value will be an address. For example, suppose we have a global variable containing a floating point value:

```
>p realnum\t
```

```
double realnum
>p realnum
realnum = 6.02257e+23
```

F If the library that defines `realnum` is subsequently unloaded, and execution is again suspended, the object doesn't become undefined, but its nature changes:

```
>p realnum\t
(const) int <unnamed>
>p realnum
-1074683528
>p realnum\X
0xbff1a178
```

This problem will not occur on Series 300/400, as the linker symbol corresponding to a global object will always be prefixed with an underscore (-).

- The debugger may show mismatched value and type information for multiply defined global data symbols when both shared and archived libraries are used. For more information on this, read the section “Relying on Undocumented Linker Behavior” in the chapter “Linking and Running Programs” in the *Programming on HP-UX* manual.
- Any shared library with basename `ALL` or `libALL` cannot be listed with the `-l` option. An explicit reference to the library must be made to force loading of symbolic debug information for that library (for example, `list procedures @libALL`).

## Disassembly Mode Limitations

- (Series 300/400 only) The debugger disregards all compiler-generated linker symbols of the form `Lnnnnn` (where `nnnnn` is any number of digits). They will not be visible in disassembly mode, even if the `as(1)` `-L` option was used. Symbols of this form should not be used in user-written assembly source.
- (Series 300/400 only) The disassemble-instruction format character (`i`) cannot be applied to constant expressions (for example, `p 0x4e71\i`). It can only be applied to dereferenced address expressions (for example, `p *($a4+6)\4i`).

- Single stepping floating-point instructions may show delayed results for operations that are actually emulated via exception traps (for example, `fsin` on the Series 300/400 MC68040 processor). Actual results may not be apparent until the next floating-point operation is performed.
- Debugging dynamically loaded code is inherently difficult, since no symbols within it are known to the debugger. On Series 600/700/800 (PA-RISC) implementations, stack traces are not possible from within dynamically loaded code.

### Save State Limitations

- When the debugger writes a “save state” file, it makes certain assumptions about the initial state of the debugger. If you have an `.xdbrc` file, those assumptions may not be valid when the `-R` file is read. For example, if the `.xdbrc` file defines assertions or breakpoints or toggles the global breakpoint, assertion, or macros state, the restored state may have different activations than when the `ss` file was created. If the `.xdbrc` file defines macros, they may have duplicate definitions after the `-R` file is read.

If a “save state” file is used as a playback file, the recorded locations of breakpoints may not correspond to meaningful locations in the *objectfile* (if it has changed). In addition, the other considerations about debugger state mentioned in the preceding paragraph apply here (regardless of how the state was established).

- The `ss` command saves the current value of the count for breakpoints. When restored, this is the value to which the count for permanent breakpoints is reset when the break occurs. This may differ from the initial count assigned to the breakpoint before the state was saved.
- The `ss` command does not save instance breakpoints (set with the `bi` command).

## Pointer Limitation

Symbolic debugging information is not always emitted for objects which are not directly referenced. For instance, if a pointer to an object is used but no fields are ever referenced, HP C++ only emits symbolic debug information for the pointer type and not for the type of object that the pointer points to. For instance, use of `Widget *` will only emit debug information for the pointer type `Widget *` and not for `Widget`. If you wish such information, you can create an extra source file which references an object of that type (`Widget`) and link it into the executable program.

## Address Format Restriction

If you set the address printing format to something `printf(3S)` does not like, you may get an error (usually memory fault) each time you try to print an address, until you fix the format with another `f` command.

## Hints for Using Assertions

Since assertions are executed before each instruction, they slow down the program execution considerably. A good practice is to narrow down where you think the problem is occurring and use assertions only on small sections of code.

Suppose you have some location (call it address `0x12345678`) that gets mysteriously overwritten some time during execution, but you do not know when. All you know is the value that is *supposed* to be there (call it `0x98765432`). A typical method for tracking this down is to set up an assertion before running the program:

```
a {if (*0x12345678 != 0x98765432) {toggle assertions; exit 0}}
```

This will make the comparison against the known value *at every instruction* during execution, and suspend the program when the location gets overwritten. Doing this with a non-trivial program can take practically *forever*.

A more effective way to do this would be to make the test only at the entry and exit of each procedure call. This speeds up execution tremendously, and isolates somewhat the area of code you need to give closer attention to:

```
bpt {Q; if (*0x12345678 == 0x98765432) {c}}
```

This sets a “procedure trace” breakpoint at the entry and exit of each debuggable procedure. This in turn makes the test and suspends the program when the location changes value (that is, at the first call or return *following* the point where it changed).

Now that the questionable segment of code has been isolated, an assertion needs to be toggled on and off to pinpoint the instruction that is at fault. If it is a large segment of code that is known to be executed many times before the error occurs, running an assertion through it can still take forever. However, a counter can be set that counts the number of times through that segment of code before the “procedure trace” test fails.

The following breakpoint command:

```
b {Q; pq $mycounter = $mycounter + 1; c}
```

can be used to silently count the number of times the suspected segment of code executes before the location changes its value. To display the value of `$mycounter`, execute this command:

```
p $mycounter
```

The passes through that segment of code can then be re-counted upon re-running the program and the assertion enabled only on the *n*th time through. This might take the form:

```
b \ $mycounter {ta; c}
```

This breakpoint is not taken until it has been encountered `$mycount` times. At that time, it turns on assertions (`ta`) and continues (`c`).

It is not necessary to have an assertion enabled for any longer than absolutely necessary.

## Window Mode Requirements

In order to implement the window mode, the debugger requires the following `terminfo` resources:

<code>cup</code>	Screen relative cursor motion.
<code>ed</code>	Clear to end of display.

`e1` Clear to end of line.

Also, the debugger requires either:

`meml` Lock memory above cursor, `meml`, and unlock memory above cursor,  
`memu` `memu` (both of these resources are available on many HP terminals).

or

`csr` Change scrolling region (available on most `vt`-compatible terminals).

If the above resources are not available, the debugger will use line mode.

The debugger also uses the following resources if available, but does not require them:

`k11` Home down or last line (as available on HP terminals, this allows the debugger to make better use of the command window);

`i11` Insert one line (used when changing the size of the source window);

`d11` Delete one line (used when changing the size of the source window);

`rev` Reverse video, `rev` (to indicated regions on the screen), and turn off all attribute modes, `srg0`;

`sms0` Enter standout mode, `sms0` (for location line/changed registers if no  
`rms0` inverse video), and exit standout mode, `rms0`.

## Installed Files

---

This appendix lists the installed files for the HP Symbolic Debugger.

---

### Debugger Installation

These are the files needed to use the HP Symbolic Debugger on your system.

- The file `end.o` must be linked at the end of the user program to give the debugger private data space in the user process. This is done automatically if linking occurs at the same time as compilation and the `-g` option is given to the compiler.

`/usr/lib/end.o`

- These are the executable program files for the HP Symbolic Debugger (only `/usr/bin/xdb` is available on Series 600/700/800 computers).

`/usr/bin/xdb`

`/usr/bin/cdb`

`/usr/bin/fdb`

`/usr/bin/pdb`

- The file `pxdb` (the preprocessor) processes the executable file the first time the debugger is invoked on it. On some releases, this step may be performed by the linker. It produces quick-lookup tables to increase the performance of the debugger and removes duplicate global definitions.

`/usr/bin/pxdb`



- The file `xdb.help` contains the database for the `help` facility, which is a summary of HP Symbolic Debugger commands. A similar file, `cdb.help`, is for use with `cdb`, `fdb`, and `pdb` and is only needed on Series 300/400 architectures.

```
/usr/lib/xdb.help
/usr/lib/cdb.help
```

The following files can be processed with `nroff(1)` to make a copy of the `help` text suitable for printing.

```
/usr/lib/xdb.help.nro
/usr/lib/cdb.help.nro
```

- The file `xdb.cat` contains the message catalog for `xdb`. For `cdb`, `fdb`, and `pdb` Series 300/400 architectures, the message catalog is `cdb.cat`. The file `pxdb.cat` contains the message catalog for `pxdb`.

```
/usr/lib/nls/C/xdb.cat
/usr/lib/nls/C/cdb.cat
/usr/lib/nls/C/pxdb.cat
```

- The following files constitute the demo package for the debugger (see the chapter in this manual entitled “Getting Started”).

```
/usr/lib/xdb_demos/README
/usr/lib/xdb_demos/RAINFALL
/usr/lib/xdb_demos/Makefile
/usr/lib/xdb_demos/demo.C
/usr/lib/xdb_demos/demo.c
/usr/lib/xdb_demos/demo.p
/usr/lib/xdb_demos/demo.f
/usr/lib/xdb_demos/C.demo
/usr/lib/xdb_demos/c.demo
/usr/lib/xdb_demos/p.demo
/usr/lib/xdb_demos/f.demo
/usr/lib/xdb_demos/gen_core.c
```

- The *HP-UX Symbolic Debugger 9.0 Release Notes* for the debugger are in the file `Debugger`.

```
/etc/newconfig/90RelNotes/Debugger
```

## G-2 Installed Files

## HP Symbolic Debugger Commands

---

This section describes command syntax and gives a description of all the HP Symbolic Debugger commands. Note that the syntax column of the tables found in this appendix provides the short form of the command and that the description column provides the long form of the command if there is one.

---

### Invocation Options

Enter the following command to start the debugger:

```
xdb [ -d dir
      -r file
      -R file
      -p file
      -P process ID
      -L
      -i file
      -o file
      -e file
      -S num
      -s
      -l library
      -l ALL ] [ objectfile [ corefile ] ]
```

The options for the `xdb` command are described as follows:

- objectfile* Is an executable program file with zero or more of its components compiled with the `-g` option. The default for *objectfile* is `a.out`.
- corefile* Is a core image from a failed execution of *objectfile* (see *core(4)* in the *HP-UX Reference*). The default for *corefile* is `core`.
- `-d dir` Specifies an alternate directory for source files. Alternate directories are searched in the order given. If a source file is not found in any alternate directory, the current directory is searched last. When searching for the source *file* in an alternate directory *altdir*, where *file* is composed of a directory and a base file name (i.e., *dirname/basename*), `xdb` first attempts to open *altdir/dirname/basename*. If this fails, `xdb` attempts to open *altdir/basename* (see *basename (1)* in the *HP-UX Reference*).
- H** `-r file` Specifies a record *file*, which is invoked immediately for overwrite, rather than for append (see the section “Record and Playback Commands” in the chapter “HP Symbolic Debugger Commands”).
- `-R file` Specifies a restore state *file*, which is processed before the `-p` option (if any) and after the `-r` option (if any). The *file* must have been created previously with the `ss` command while debugging the same *objectfile* (see the section “Save State Command” in this appendix), which the debugger attempts to verify when the `-R` option is used.
- `-p file` Specifies a playback *file*, which is invoked immediately (see the section “Record and Playback Commands” in the chapter “HP Symbolic Debugger Commands”).
- `-P process-ID` Specifies the *process-ID* of an existing process that the user wishes to debug (see the section “Adopting a Running Process” in the chapter “Using the HP Symbolic Debugger” in this manual).
- `-L` Forces the line-oriented interface, even if `xdb` can support the window-oriented interface on the terminal type specified by environment variable `TERM`.
- `-i file` Redirects standard input to the child process from the designated file or character device.
- `-o file` Redirects standard output from the child process to the designated file or character device.

## H-2 HP Symbolic Debugger Commands

- e** *file*                    Redirects standard error from the child process to the designated file or character device.
- S** *num*                    Sets the size of the string cache to *num* bytes (default is 1024, which is also the minimum). The string cache holds data read from the *objectfile*.
- s**                            Causes all shared libraries used by an application to be loaded as private (unshared) copies. This option or the **-1** option (which implies **-s**) is required if breakpoints will be set or single stepping will be done in shared libraries.
- 1** *shared-library*        Pre-loads the symbolic debug information (and linker symbol table) into the debugger so that the user can view code, set breakpoints, and do other debugging operations prior to running the program. If the **-1** option is not used for a given library, no symbolic information concerning the library will be available, and you will not be able to debug that library at the source level, unless

- You explicitly make a reference to a symbol in that library (e.g. *symbol@shared-library* as opposed to just *symbol*), or
- The debugger stops execution at some location within that library.

*shared-library* may be implicitly loaded by the program (linked in with the *ld(1)* **-1** option), or explicitly loaded by *shL\_load(3X)*.

If *shared-library* is not a complete path name, it will be searched for using the same search rules used by the dynamic loader (see the section “Locating Shared Libraries” in Chapter 6, the *ld(1)* **+b** and **+s** options, and the section “Library Location and the Dynamic Loader” in the manual *Programming on HP-UX*). If the library is not located, any directories previously specified with the **-d** option will also be searched, followed by the current directory. If it is still not located, the symbolic debug information will still be available once the library has been mapped in (loaded), and an explicit reference to a symbol within it has been made.

The trailing *.sl* is optional in *shared-library*.

- 1** **ALL**                    Pre-loads the debug information (and linker symbol table) into the debugger for all shared-libraries used by the program, with the exception of libraries loaded with *shL\_load(3X)*, which the user must list using a separate **-1** option for each.



There can only be one *objectfile* and one *corefile* per debugging session (activation of the debugger). The program (*objectfile*) is not invoked as a child process until you give an appropriate command (see the section “Job Control Commands” in this appendix). The same program may be restarted, as different child processes, many times during one debugging session.

**H**

## Window Mode Commands

Table H-1. Window Mode Commands

Cmd	Syntax	Description
fr	fr	<b>floating point registers</b> Displays the Series 600/700/800 (PA-RISC) or Series 300/400 (MC680x0) floating point registers in the register window when the debugger is in disassembly mode.
tf	tf	<b>toggle float</b> Toggles the display of floating point registers between the single- and double- precision modes.
gr	gr	<b>general registers</b> Displays the Series 600/700/800 (PA-RISC) or Series 300/400 (MC680x0) general registers in the register window when the debugger is in disassembly mode.
sr	sr	<b>special registers</b> Displays the Series 600/700/800 (PA-RISC) special registers (space and control) when the debugger is in disassembly mode.
td	td	<b>toggle disassembly</b> Toggles the source window between disassembly mode and source mode.
ts	ts	<b>toggle screen</b> Toggles the source window between all source or all assembly and split-screen mode.
u	u	<b>update</b> Updates the source and location windows to show the current location of the user program.
U	U	<b>Update</b> Clears the screen of data and redraws the screen.

H

## Window Mode Commands

Table H-1. Window Mode Commands (continued)

Cmd	Syntax	Description
w	w <i>number</i>	<b>window</b> If your terminal supports windowing, this command changes the size of the source window to the number of lines that you specify. Enter a number from 1 to the screen size minus 3.
+r	+r	Scroll the Series 300/400 or Series 600/700/800 floating-point register display forward four lines.
-r	-r	Scroll the Series 300/400 or Series 600/700/800 floating-point register display back four lines.

H

## File Viewing Commands

Table H-2. File Viewing Commands

Cmd	Syntax	Description
+	+ [ <i>number</i> ]	Moves forward in the current file the specified number of lines (or the specified number of instructions in disassembly mode). If you do not enter a number, the next line (or instruction) becomes the current line (or instruction).
-	- [ <i>number</i> ]	Moves the specified number of lines (or the specified number of instructions in disassembly mode) backward in the current file and updates the windows. The default is one line (or instruction) before the current line (or instruction).
/	/ [ <i>string</i> ]	Searches forward in the file for the specified string. Searches wrap around the end of the file. If you do not enter a string, the last one that you entered is used again. The string must be literal; wild cards are not supported.
?	? [ <i>string</i> ]	Searches backward in the current file for a specific pattern. Searches wrap around the beginning of the file. If you do not enter a string, the last search string is used again. The string must be literal; wild cards are not supported.

H



## File Viewing Commands

**Table H-2. File Viewing Commands (continued)**

Cmd	Syntax	Description
D	D " <i>dir</i> "	<b>Directory</b> Adds the directory that you specify to the end of the list of directory search paths for source files.
ld	ld	<b>list directories</b> Lists all the alternate directories that are searched when the debugger tries to locate the source files. The list order is the same as the search order.
lf	lf [ <i>string</i> ] [ <i>@shared-library</i> ]	<b>list files</b> Lists all source files containing executable statements that were compiled to build the executable file. If a string is specified, only those files beginning with the string are listed. <i>@shared-library</i> restricts the listing to files that were used to build the named shared library.
L	L	<b>Location</b> Displays in the command window the current file, procedure, line number and the source line (text) for the current point of execution.
n	n	<b>next</b> Repeats the previous search (/ or ?) command.
N	N	<b>Next</b> Repeats the previous search (/ or ?) command, searching in the opposite direction.

H

Table H-2. File Viewing Commands (continued)

Cmd	Syntax	Description
v	v [ <i>location</i> ]	<p><b>view</b></p> <p>Displays one source window forward from the current source window if no <i>location</i> is given. One line from the previous window is preserved for context. If your terminal does not support windowing, only the new source line is displayed. Using the <i>location</i> option causes the specified location to become the current location, and the source at the specified location is then displayed in the source window.</p>
va	va <i>address</i>	<p><b>view address</b></p> <p>Displays in the source window assembly code at the specified address. A specified address can be an absolute address or symbolic code label with an optional offset (for example, <code>_start + 0x20</code>).</p>

H

## Data Viewing and Modification Commands

Table H-3. Data Viewing and Modification Commands

Cmd	Syntax	Description
l	<p>l [<i>proc</i>[:<i>depth</i>]]</p> <p>l [<i>class</i>][:[:<i>proc</i>[:<i>depth</i>]]]</p>	<p><b>list</b></p> <p>Lists all parameters and local variables of the current procedure. You can optionally specify any active procedure and its depth on the stack.</p>
lc	lc [ <i>string</i> ]	<p><b>list common</b></p> <p>Used when debugging an HP FORTRAN 77 program, this command displays HP FORTRAN 77 common blocks and their associated variables (Series 600/700/800 computers). If a string is specified, only those common blocks whose names begin with that string are printed; otherwise, all common blocks within the current subroutine/function are printed.</p>
lcl	lcl [ <i>string</i> ][ <i>@shared-library</i> ]	<p><b>list classes</b></p> <p>Lists all classes (regular classes and templates) known to the debugger. If a string is specified, only those classes whose names begin with that string are listed.</p> <p><i>@shared-library</i> restricts the search to the named shared library.</p>

H

## Data Viewing and Modification Commands

**Table H-3. Data Viewing and Modification Commands (continued)**

Cmd	Syntax	Description
lct	lct [ <i>string</i> ] [ @ <i>shared-library</i> ]	<p><b>list class templates</b>  Lists all class templates known to the debugger. If a string is specified, only those class templates whose names begin with that string are listed. @<i>shared-library</i> restricts the search to the named shared library.</p>
lft	lft [ <i>string</i> ] [ @ <i>shared-library</i> ]	<p><b>list function templates</b>  Lists all function templates known to the debugger. If a string is specified, only those function templates whose names begin with that string are listed. @<i>shared-library</i> restricts the search to the named shared library.</p>
lg	lg [ <i>string</i> ] [ @ <i>shared-library</i> ]	<p><b>list globals</b>  Lists all global variables and their values. If a string is specified, only those global variables whose names begin with that string are listed. @<i>shared-library</i> restricts the search to the named shared library.</p>

H

## Data Viewing and Modification Commands

Table H-3. Data Viewing and Modification Commands (continued)

Cmd	Syntax	Description
ll	ll [ <i>string</i> ] [ <i>@shared-library</i> ]	<b>list labels</b> Lists all external labels and program entry points known to the linker. If a string is specified, only those external labels (symbols) which begin with this prefix are used. <i>@shared-library</i> restricts the search to the named shared library.
lm	lm [ <i>string</i> ]	<b>list macros</b> Displays all user-defined macros and their definitions. If a string is specified, only those macros whose names begin with this string are listed.
lo	lo [[ <i>class</i> ][:][ <i>string</i> ] [ <i>@shared-library</i> ]	<b>list overload</b> List overloaded C++ functions. If <i>string</i> is present, only those with the same initial characters are listed. This can also be qualified by a <i>class</i> . <i>@shared-library</i> restricts the search to the named shared library.

H

## Data Viewing and Modification Commands

**Table H-3. Data Viewing and Modification Commands (continued)**

Cmd	Syntax	Description
lp	<p>lp [ <i>string</i> ] [ @<i>shared-library</i> ]</p> <p>lp [ <i>class</i> ] :: [ <i>string</i> ] [ @<i>shared-library</i> ]</p>	<p><b>list procedures</b>  Lists all procedure names and their aliases, locations in memory, file names, and line numbers. If a string is specified, only those procedures whose names begin with this string are listed. @<i>shared-library</i> restricts the search to the named shared library.</p>
lr	<p>lr [ <i>string</i> ]</p>	<p><b>list registers</b>  Lists all registers and their contents. If a string is specified, only those registers beginning with this string are listed. The leading \$ is significant.</p>
ls	<p>ls [ <i>string</i> ]</p>	<p><b>list specials</b>  Lists all special variables and their values. Registers are not listed. If a string is specified, only those special variables whose names begin with this string are listed. The leading \$ is significant.</p>
lsl	<p>lsl</p>	<p><b>list shared libraries</b>  List all shared libraries known to the debugger.</p>

H

## Data Viewing and Modification Commands

Table H-3. Data Viewing and Modification Commands (continued)

Cmd	Syntax	Description
ltf	ltf [ <i>string</i> ] [ <i>@shared-library</i> ]	<b>list template functions</b> Lists all template functions known to the debugger. If a string is specified, only those template functions whose names begin with this string are listed. <i>@shared-library</i> restricts the search to the named shared library.
lx	lx	<b>list exceptions</b> Lists the current state of the <b>throw</b> and <b>catch</b> toggles and command-list associated with them.

H

## Data Viewing and Modification Commands

**Table H-3. Data Viewing and Modification Commands (continued)**

Cmd	Syntax	Description
mm	mm [ [ @ ] <i>shared-library</i> ]	<p><b>memory map</b> Shows a memory-map of all currently loaded shared libraries and the main program. If <i>shared-library</i> is present, only the memory-map for the named library is listed. The memory-map provides the following information for each loaded region: basename of the library (as used in symbolic names; for example, <code>libc</code>), upper and lower bounds of both text and data addresses, the handle (see <i>shl_load(3X)</i>), the complete path name, and whether the region is writable (debuggable) or read-only (shared).</p> <p>Note that libraries explicitly loaded with <i>shl_load(3)</i> are visible to the debugger until they are unloaded.</p>

H



## Data Viewing and Modification Commands

**Table H-3. Data Viewing and Modification Commands (continued)**

Cmd	Syntax	Description
p	$p \left\{ \begin{array}{l} \text{expr} \left[ \left\{ \begin{array}{l} \backslash \\ ? \end{array} \right\} \text{format} \right] \\ \text{class}:: \\ \left[ \begin{array}{l} + \\ - \end{array} \right] \left[ \left[ \backslash \right] \text{format} \right] \end{array} \right\}$	<p><b>print</b>            Displays program data in the formats shown in tables 1-5 and 1-6 of chapter 1, "Reference Tables". A <i>format</i> has the syntax:</p> $[ \text{count} ] \{ \text{formchar} \} [ \text{size} ]$ <p><i>Formchar</i>, which is required, is the actual format in which you choose to display the data. <i>Count</i> is the number of times to apply the format. <i>Size</i> is the number of bytes that are formatted for each data item, and overrides the default size for the given format. <b>p+</b> prints the next element. <b>p-</b> prints the previous element. Use the <i>\format</i> option to display the value of the expression in a specific format. Use the <i>?format</i> option to print the address of the evaluated expression in the selected format. The <b>p</b> command is also used to modify the value of a variable when <i>expr</i> contains an assignment operator. <i>class::</i> prints the values of all static data members of <i>class</i>. (See Table 4-4 for the data viewing formats.)</p>

H

## Data Viewing and Modification Commands

**Table H-3. Data Viewing and Modification Commands (continued)**

Cmd	Syntax	Description
pq	$pq \left\{ \begin{array}{l} expr \left[ \left\{ \begin{array}{l} \backslash \\ ? \end{array} \right\} format \right] \\ class:: \\ \left[ \begin{array}{l} + \\ - \end{array} \right] \left[ \left[ \backslash \right] format \right] \end{array} \right\}$	<p><b>print quietly</b>            Does not print anything unless an error occurs. Otherwise, the action is the same as for <b>p</b>.            The <b>pq</b> command can be used to do assignments without causing output to the command window. This is useful in breakpoint and assertion command lists.</p>

H

## Source Directory Mapping Commands

Table H-4. Source Directory Mapping Commands

Cmd	Syntax	Description
apm	$\{ \text{apm} \} \left\{ \begin{array}{l} \text{old\_path} \\ \dots \end{array} \right\} [ \text{new\_path} ]$	<b>add path map</b> Allows you to modify the path the debugger will use to locate a set of source files.
lpm	lpm	<b>list path maps</b> Lists the path maps in the order in which they will be searched.
dpm	$\{ \text{dpm} \} \left[ \begin{array}{l} n \\ * \end{array} \right]$	<b>delete path map</b> Removes the latest path map entered if used with no arguments. If a positive integer <i>n</i> is given, the <i>n</i> th path map will be removed. If a * is given, all the path maps will be removed.

H

## Stack Viewing Commands

Table H-5. Stack Viewing Commands

Cmd	Syntax	Description
t	t [ <i>depth</i> ]	<b>trace</b> Prints a stack trace. You can optionally specify a <i>depth</i> . The default depth is 20 levels. If an optional depth is supplied, only the procedures up to this depth in the stack are displayed.
T	T [ <i>depth</i> ]	<b>Trace</b> Prints a stack trace. You can optionally specify a <i>depth</i> . The default depth is 20 levels. If an optional depth is supplied, only the procedures up to this depth in the stack are displayed. Displays everything the <b>t (trace)</b> command displays, plus all local variables and their values in \n format.
V	V [ <i>depth</i> ]	<b>View</b> Displays the text for the procedure at the depth on the program stack that you specify. If you do not enter a depth, the current active procedure is used.
up	up [ <i>n</i> ]	Moves up <i>n</i> (default one) levels toward the top of the stack.
down	down [ <i>n</i> ]	Moves down <i>n</i> (default one) levels toward the bottom of the stack.

H

## Stack Viewing Commands

**Table H-5. Stack Viewing Commands (continued)**

Cmd	Syntax	Description
top	top	Moves to the top of the stack. It is shorthand for the debugger command <code>V 0</code> , which moves you to the top of the stack.
tst	tst	<b>toggle stubs</b> Toggles the visibility of inter-procedural stubs in stack traces. (PA-RISC only)

H

---

## Status Viewing Command

Table H-6. Status Viewing Command

Cmd	Syntax	Description
I	I	<b>Inquire</b> Prints the current state of the debugger. The output contains information such as the version number of the debugger, program name, number of source files and procedures, process-ID of the child process, number of breakpoints, record and playback information, etc.

H |

## Job Control Commands

Table H-7. Job Control Commands

Cmd	Syntax	Description
c	c [ <i>location</i> ]	<b>continue</b> Resumes execution after a breakpoint or a signal has been encountered, ignoring the signal, if any. If a location is specified, a temporary breakpoint is set at that location.
C	C [ <i>location</i> ]	<b>Continue</b> Resumes execution after a breakpoint or a signal has been encountered, allowing the signal, if any, to be received by the child process. If a location is specified, a temporary breakpoint is set at that location.
g	g $\left[ \begin{array}{l} \textit{line} \\ \# \textit{label} \\ + \textit{lines} \\ - \textit{lines} \end{array} \right]$	<b>goto</b> Moves the current point of execution suspension to the specified <i>line</i> or <i>label</i> . The specified <i>line</i> or <i>label</i> must be within the same procedure where execution is currently suspended (at depth zero on the stack). The program counter will change so that the given line number or the <i>line</i> that <i>#label</i> appears on becomes the next executable line. Execution does not automatically resume. The + and - move the program counter the specified number of source or assembly lines from the current program counter position.
k	k	<b>kill</b> Terminates the current child process, if any.
lz	lz	<b>list signals</b> See the section "Signal Control Commands" in this appendix.

H

Table H-7. Job Control Commands (continued)

Cmd	Syntax	Description
r	r [arguments]	<p><b>run</b>  Runs a new child process with the <i>argument</i> list (if any). The existing child process, if any, is terminated first (after confirmation is given). If no arguments are given, the ones used with the last <b>r</b> command are used again (none if <b>R</b> was used last).</p> <p>The <i>arguments</i> can contain &lt; and &gt; for redirecting standard input and standard output. (&lt; does an <i>open(2)</i> of file descriptor 0 for read-only; &gt; does a <i>creat(2)</i> of file descriptor 1 with mode 0666). Redirection can also be done with &gt;&gt; and &gt;&amp;. Arguments can contain shell variables and meta characters, quote marks, or other special syntax (that will be expanded by a Bourne Shell (<i>sh(1)</i>)). The remainder of the input line following the <b>r</b> command is used as the argument-list, so it cannot be enclosed in a command list (<b>{}</b>). Thus, the <b>r</b> command cannot be used within a breakpoint, assertion, or <b>if</b> command. The environment for the child process is the same as for the debugger.</p>

H



## Job Control Commands

**Table H-7. Job Control Commands (continued)**

Cmd	Syntax	Description
R	R	<b>Run</b> Lets you run a program as a new child process with no argument list. If a child process already exists, the debugger asks if you want to terminate the child process first. The environment for the child process is the same as that for the debugger.
s	s [ <i>number</i> ]	<b>step</b> Single step, executing one source statement or machine instruction before pausing and prompting for another command. In source mode, one source statement is executed; in disassembly mode, one machine instruction is executed. If a procedure call is encountered, the procedure is single stepped in the same manner ("stepped into"). To execute more than one statement or instruction, enter that number as the <i>number</i> parameter.
S	S [ <i>number</i> ]	<b>Step</b> Single steps. In source mode, one source statement is executed; in disassembly mode, one machine instruction is executed (several machine instructions might be equivalent to one source statement). If a procedure call is encountered, it is not "stepped into". Instead, execution steps to the statement following the call ("stepped over"). To execute more than one statement or instruction, enter that number as the <i>number</i> parameter.
z	z [ <i>signal</i> ] [ i ] [ r ] [ s ] [ Q ]	<b>signal</b> See the section "Signal Control Commands" in this appendix.

H

## Breakpoint Commands

### Overall Breakpoint Commands

Table H-8. Overall Breakpoint Commands

Cmd	Syntax	Description
lb	lb [ <i>@shared-library</i> ]	<b>list breakpoints</b> Displays all breakpoints in the program, both active and suspended, and the overall breakpoint state. <i>@shared-library</i> lists only those breakpoints in the named shared library.
tb	tb	<b>toggle breakpoints</b> Toggles the overall breakpoint state from active to suspended or vice versa. The state of the individual breakpoints remains unchanged.

H

## Breakpoint Commands

### Breakpoint Creation Commands

Table H-9. Breakpoint Creation Commands

Cmd	Syntax	Description
b	b [ <i>location</i> ] [ \ <i>count</i> ] [ <i>command-list</i> ]	<b>breakpoint</b> Sets a breakpoint at the location that you specify. If you do not enter a location, the current line in the source or disassembly window is used. The breakpoint is executed on each occurrence ( <i>count</i> ) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the <i>command-list</i> .
ba	ba <i>address</i> [ \ <i>count</i> ] [ <i>command-list</i> ]	<b>breakpoint address</b> Sets a breakpoint at the specified address. Note that the address can be specified by giving the name of a procedure or an expression containing a name. The breakpoint is executed on each occurrence ( <i>count</i> ) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the <i>command-list</i> .

H

## Breakpoint Commands

**Table H-9. Breakpoint Creation Commands (continued)**

Cmd	Syntax	Description
bb	bb [ <i>depth</i> ] [ <i>\count</i> ] [ <i>command-list</i> ]	<p><b>breakpoint beginning</b>            Sets a breakpoint at the first executable statement of the procedure at the specified <i>depth</i> on the program stack. If you do not enter a <i>depth</i>, the procedure shown in the source window is used. The breakpoint is executed on each occurrence (<i>count</i>) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the <i>command-list</i>.</p>

H

## Breakpoint Commands

Table H-9. Breakpoint Creation Commands (continued)

Cmd	Syntax	Description
bi	<pre>bi <i>expr.proc</i> [\ <i>count</i>] [<i>command-list</i>]</pre> <pre>bi <math>\begin{bmatrix} -c \\ -C \end{bmatrix}</math> <i>expr</i> [<i>command-list</i>]</pre>	<p><b>breakpoint instance</b>  Sets an <i>instance</i> breakpoint at the first executable line of member function <i>proc</i> of the class instance to which the expression <i>expr</i> evaluates. If <i>proc</i> is not specified, an instance breakpoint will be set on all member functions of the instance's class. This breakpoint is only recognized when the specified or implied functions are called for this instance. If <i>count</i> is given, the breakpoint will not be recognized until it is hit the designated number of times. If a <i>command-list</i> is specified, it will be executed when the breakpoint is hit. If there is no <i>command-list</i>, the debugger pauses for command input. The <i>-c</i> option forces a breakpoint to be set only on member functions of the instance's immediate class; <i>-C</i> also sets breakpoints on member functions of base classes.</p>

H

## Breakpoint Commands

**Table H-9. Breakpoint Creation Commands (continued)**

Cmd	Syntax	Description
bpc	$\text{bpc} \begin{bmatrix} -c \\ -C \end{bmatrix} \text{class} [\text{command-list}]$	<p><b>class breakpoint</b> Sets <i>class</i> breakpoints at the first executable line of all member functions of <i>class</i>.</p> <p>If <i>-c</i> is given, breakpoints will be set only on member functions of the designated <i>class</i> and not of its base classes. If <i>-C</i> is given, breakpoints are also set on member functions of base classes.</p>
bpo	$\text{bpo} [ [ [ \text{class} ] :: ] \text{proc} [\text{command-list}] ]$	<p><b>breakpoint overload</b> Set <i>overload</i> breakpoints at the first executable line of all overloaded functions with name <i>proc</i> (which may be qualified by a class.)</p> <p>When one of these breakpoints is hit, <i>command-list</i> is executed. If <i>command-list</i> is omitted, the debugger pauses for command input.</p>

H

## Breakpoint Commands

Table H-9. Breakpoint Creation Commands (continued)

Cmd	Syntax	Description
bt	$\text{bt} \left[ \begin{array}{l} \textit{proc} \\ \textit{depth} \end{array} \right] [\backslash\textit{count}] [\textit{command-list}]$	<p><b>breakpoint trace</b>  Sets a <i>trace</i> breakpoint at the current or named procedure or at the procedure that is at the specified depth on the program stack. A breakpoint is set at the entry and exit point of the procedure. If you include a <i>command-list</i>, it is executed at the beginning of the procedure or subprogram. On Series 600/700/800 computers, the following <i>command-list</i> will be executed at the end of the procedure or subprogram.</p> $\{ \text{Q;p \$ret0\d;c} \}$ <p>For Series 300/400 computers, the <i>command-list</i> is:</p> $\{ \text{Q;L;c} \}$ <p>If you omit a <i>command-list</i>, the following is executed at the beginning of the procedure or subprogram:</p> $\{ \text{Q; t 2; c} \}$

H

## Breakpoint Commands

Table H-9. Breakpoint Creation Commands (continued)

Cmd	Syntax	Description
bu	bu [ <i>depth</i> ] [ \ <i>count</i> ] [ <i>command-list</i> ]	<p><b>breakpoint uplevel</b>            Sets an <i>uplevel</i> breakpoint to occur immediately on return from the procedure at the specified <i>depth</i> on the program stack. If you do not enter a <i>depth</i>, the procedure at <i>depth</i> 0 is used. The breakpoint is executed on each occurrence (<i>count</i>) that you specify. You can enter a list of commands to be executed at the breakpoint by specifying the <i>command-list</i>.</p>
bx	bx [ <i>depth</i> ] [ \ <i>count</i> ] [ <i>command-list</i> ]	<p><b>breakpoint exit</b>            Sets an <i>exit</i> breakpoint at the epilogue code of the procedure at the specified <i>depth</i> on the program stack. If you do not enter a <i>depth</i>, the procedure shown in the source window is used. The breakpoint is executed on each occurrence (<i>count</i>) that you specify. You can enter a list of commands to be executed at the breakpoint by specifying the <i>command-list</i>.</p>

H



## Breakpoint Commands

## Breakpoint Status Commands

**Table H-10. Breakpoint Status Commands**

Cmd	Syntax	Description
ab	ab $\left[ \begin{array}{l} \textit{number} \\ * \\ \textit{@shared-library} \end{array} \right]$	<b>activate breakpoint</b> Activates the breakpoint having the <i>number</i> that you specify. If you do not enter a <i>number</i> , the breakpoint at the current line is activated. Use the asterisk (*) to activate all breakpoints, including all-procedure breakpoints. Note that to activate an instance, class, or overload breakpoint, <i>number</i> must be specified. <i>@shared-library</i> activates only those breakpoints in the named shared library.
bc	bc <i>number expr</i>	<b>breakpoint count</b> Sets the count of the specified breakpoint <i>number</i> to the integer value of the evaluated expression <i>expr</i> that you enter.
db	db $\left[ \begin{array}{l} \textit{number} \\ * \\ \textit{@shared-library} \end{array} \right]$	<b>delete breakpoint</b> Deletes the breakpoint having the <i>number</i> that you specify. If you do not enter a <i>number</i> , the breakpoint at the current line is deleted. Use the asterisk (*) to delete all breakpoints including all-procedure breakpoints. <i>@shared-library</i> deletes only those breakpoints in the named shared library.
sb	sb $\left[ \begin{array}{l} \textit{number} \\ * \\ \textit{@shared-library} \end{array} \right]$	<b>suspend breakpoint</b> Suspends (deactivates) the breakpoint having the <i>number</i> that you specify. If you do not enter a <i>number</i> , the breakpoint at the current line is suspended. Use the asterisk (*) to suspend all breakpoints, including all-procedure breakpoints. To suspend an instance, class, or overload breakpoint, <i>number</i> must be specified. <i>@shared-library</i> suspends only those breakpoints in the named shared library.

All-Procedures Breakpoint Commands

Table H-11. All-Procedures Breakpoint Commands

Cmd	Syntax	Description
bp	bp [ <i>@shared-library</i> ] [ <i>command-list</i> ]	<p>breakpoint procedure</p> <p>Sets permanent <i>procedure</i> breakpoints at the first executable statement of every procedure for which debugger information is available. The breakpoint is encountered each time the procedure is entered. When any procedure breakpoint is encountered, the <i>command-list</i> is executed. If <i>command-list</i> is omitted, the debugger pauses for command input. <i>@shared-library</i> sets procedure breakpoints only in the named shared library.</p>

H

## Breakpoint Commands

**Table H-11. All-Procedures Breakpoint Commands (continued)**

Cmd	Syntax	Description
bpt	bpt [ <i>@shared-library</i> ] [ <i>command-list</i> ]	<p>Sets permanent <i>procedure trace</i> breakpoints at the first and last executable statement of every procedure for which debugger information is available. The breakpoints are encountered each time the procedure is entered or exited. The commands, if any, are associated with the entry breakpoint. If no <i>command-list</i> is specified, the entry <i>command-list</i> defaults to:</p> <p style="text-align: center;">{Q;t 2;c}</p> <p>The exit <i>command-list</i> on a Series 600/700/800 computer is:</p> <p style="text-align: center;">{Q;p \$ret\d;c}</p> <p>On a Series 300/400 computer, the exit <i>command-list</i> is:</p> <p style="text-align: center;">{Q;L;c}</p> <p><i>@shared-library</i> sets procedure trace breakpoints only in the named shared library.</p>

H

## Breakpoint Commands

**Table H-11. All-Procedures Breakpoint Commands (continued)**

Cmd	Syntax	Description
bpx	bpx [ @shared-library ] [ command-list ]	Sets permanent <i>procedure exit</i> breakpoints after the last executable statement of every procedure for which debugger information is available. The breakpoint is encountered each time the procedure is exited. When any procedure exit breakpoint is encountered, the <i>command-list</i> is executed. If <i>command-list</i> is omitted, the debugger pauses for command input. @shared-library sets procedure exit breakpoints only in the named shared library.
dp	dp [ @shared-library ]	<b>delete procedure</b> Deletes all <i>procedure</i> breakpoints set with the bp (breakpoint procedure) command. All breakpoints set by commands other than the bp command will remain in effect. @shared-library deletes procedure breakpoints only in the named shared library.

H

## Breakpoint Commands

**Table H-11. All-Procedures Breakpoint Commands (continued)**

Cmd	Syntax	Description
Dpt	Dpt [ <i>@shared-library</i> ]	Deletes all <i>procedure trace</i> breakpoints at the first and last executable statement of every procedure. All breakpoints set by commands other than the <b>bpt</b> command will remain in effect. <i>@shared-library</i> deletes procedure trace breakpoints only in the named shared library.
Dpx	Dpx [ <i>@shared-library</i> ]	Deletes all <i>procedure exit</i> breakpoints at the last executable statement of every procedure. All breakpoints set by commands other than the <b>bpx</b> command will remain in effect. <i>@shared-library</i> deletes procedure exit breakpoints only in the named shared library.

H

## Global Breakpoint Commands

Table H-12. Global Breakpoint Commands

Cmd	Syntax	Description
abc	abc <i>command-list</i>	Defines a global breakpoint <i>command-list</i> which will be executed whenever any user defined breakpoint is encountered. These include normal, procedure, procedure trace, procedure exit, class, instance, and overload breakpoints.
dbc	dbc	Deletes the global breakpoint command list.

## Breakpoint Commands

### Auxiliary Breakpoint Commands

Table H-13. Auxiliary Breakpoint Commands

Cmd	Syntax	Description
" <i>any string</i> "	" <i>any string</i> "	The <i>string</i> command echoes any string that is enclosed in quotation marks.
i	i <i>expr</i> { <i>command-list</i> } [ { <i>command-list</i> } ]	<b>if</b> The i ( <b>if</b> ) command lets you conditionally execute commands in a <i>command-list</i> . If the expression evaluates to a non-zero value, the first group of commands is executed. If the expression evaluates to zero, the second <i>command-list</i> , if provided, is executed.
Q	Q	<b>Quiet</b> The Q( <b>Quiet</b> ) command suppresses the "breakpoint at <i>address</i> ..." debugger messages that are normally displayed when a breakpoint is encountered. The Q ( <b>Quiet</b> ) command must be the first command in a command list; otherwise, it is ignored.

---

## Exception Handling Commands

Table H-14. Exception Handling Commands

Cmd	Syntax	Description
txt	txt	<b>toggle exception throw</b> Turns off and on the stopping of the debugger immediately prior to an exception throw. By default, the debugger stops immediately prior to an exception throw.
xtc	xtc [ <i>command-list</i> ]	<b>exception throw command</b> Defines a debugger <i>command-list</i> to be executed when a stop on throw occurs.
txc	txc	<b>toggle exception catch</b> Turns off and on the stopping of the debugger at the first statement of any <b>catch</b> clause. By default, the debugger stops at the first statement of any <b>catch</b> clause.
xcc	xcc [ <i>command-list</i> ]	<b>exception catch command</b> Defines a debugger <i>command-list</i> to be executed when stop on catch occurs.

H



## Assertion Control Commands

**Table H-15. Assertion Control Commands**

Cmd	Syntax	Description
a	a <i>command-list</i>	<b>assert</b> Creates an assertion consisting of the <i>command-list</i> that you enter. You can enclose the <i>command-list</i> in braces to separate it from other commands on the same line.
aa	aa [ <i>number</i> ] *	<b>activate assertion</b> Activates the assertion having the <i>number</i> that you enter. Using the * option causes all assertions to be activated. Overall assertion mode is activated if any individual suspended assertion is activated.
da	da [ <i>number</i> ] *	<b>delete assertion</b> Deletes the assertion having the <i>number</i> that you enter. Using the * option causes all assertions to be deleted.
la	la	<b>list assertions</b> Lists the number, the state (active or suspended) and the command list for each assertion, as well as the overall assertion state (active or suspended).
sa	sa [ <i>number</i> ] *	<b>suspend assertion</b> Suspends the assertion having the <i>number</i> that you enter. Using the * option causes all assertions to be suspended. Overall assertion mode is suspended if the last active assertion is suspended.

H

## Assertion Control Commands

**Table H-15. Assertion Control Commands (continued)**

Cmd	Syntax	Description
ta	ta	<p><b>toggle assertions</b></p> <p>Toggles the overall assertion state between active and suspended.</p>
x	x [ <i>expr</i> ]	<p><b>exit</b></p> <p>Causes program execution to stop as if a breakpoint has been reached. This can be used only in an assertion command list. If the expression (<i>expr</i>) is not given or it evaluates to zero, the debugger returns to command mode, ignoring any remaining commands in the assertion command list. If <i>expr</i> evaluates to non-zero, any remaining commands in the command list are executed.</p>

H

## Record and Playback Commands

**Table H-16. Record and Playback Commands**

Cmd	Description												
> <i>file</i>	Sets or changes the record file to <i>file</i> , turns recording on, rewrites the file from the beginning, and only records commands. If <i>file</i> exists, you are asked if you want to overwrite it.												
>> <i>file</i>	Sets or changes the record file to <i>file</i> , turns recording on, and only records commands. All recording is appended to the existing <i>file</i> ; otherwise, a new file is created.												
>	Displays the recording state and the current recording file. Can also use ">>".												
< <i>file</i>	Starts playback from the file.												
<< <i>file</i>	Starts playback from the file using the "line-at-a-time" feature. Each command line from the playback file is shown before it is executed, and the debugger provides a list of the following commands for you to take some action:  <i>command</i> (<cr>, S, <num>, C, Q, or ?):  You can use any of the above options as described:  <table style="margin-left: 40px;"> <tr> <td>&lt;cr&gt;</td> <td>execute one command line</td> </tr> <tr> <td>S</td> <td>skip one command line</td> </tr> <tr> <td>&lt;num&gt;</td> <td>execute number of command lines</td> </tr> <tr> <td>C</td> <td>continue through all playback</td> </tr> <tr> <td>Q</td> <td>quit playback mode</td> </tr> <tr> <td>?</td> <td>gives this explanation of the above commands</td> </tr> </table>	<cr>	execute one command line	S	skip one command line	<num>	execute number of command lines	C	continue through all playback	Q	quit playback mode	?	gives this explanation of the above commands
<cr>	execute one command line												
S	skip one command line												
<num>	execute number of command lines												
C	continue through all playback												
Q	quit playback mode												
?	gives this explanation of the above commands												
tr	<b>toggle record</b> Toggles recording; toggles the state of the record mechanism between active and suspended.												
>t	Turns recording on. (active)												
>f	Turns recording off. (suspended)												

H

## Record and Playback Commands

**Table H-16. Record and Playback Commands (continued)**

Cmd	Description
>c	Closes the record file.
>@file	Sets or changes the <i>record-all</i> file to <i>file</i> , rewrites from the beginning, and turns recording on. If <i>file</i> exists, you are asked if you want to overwrite it. Captures all input to and output from the debugger command window, except user program output.
>>@file	Sets or changes the <i>record-all</i> file to <i>file</i> , and turns recording on. Appends <i>record-all</i> output to the existing <i>file</i> . Captures all input to and output from the debugger command window, except user program output.
>@	Displays the current <i>record-all</i> state and file. Can also use ">>@".
tr @	<b>toggle record @</b> Toggles the state of the <i>record-all</i> mechanism between active and suspended.
>@t	Turns <i>record-all</i> on (active).
>@f	Turns <i>record-all</i> off (suspended).
>@c	Closes the <i>record-all</i> file.

H |

---

## Macro Facility Commands

**Table H-17. Macro Facility Commands**

Cmd	Syntax	Description
def	<code>def name replacement-text</code>	Defines a macro substitution (user-defined command) for HP Symbolic Debugger commands. <i>Name</i> can be any string of letters or digits, beginning with a letter. <i>Replacement-text</i> can be any string of letters, blanks, tabs or other printing characters. The string must be contained on one line.
tm	<code>tm</code>	<b>toggle macros</b> Toggles the state of the macro mechanism between active and suspended.
undef	<code>undef { name }           *</code>	Removes macro defined as <i>name</i> . Using the * option causes all macros to become undefined.

H

## Miscellaneous Commands

Table H-18. Miscellaneous Commands

Cmd	Syntax	Description
!	! [ <i>command_line</i> ]	<p>Invokes a shell program. The environment variable <code>SHELL</code> gives the name of the shell program to invoke. If <code>SHELL</code> is not found, the debugger executes <code>/bin/sh</code>. If <i>command_line</i> is present, it is given to <code>SHELL</code> via the <code>-c</code> option. Otherwise, <code>SHELL</code> is given a <code>-i</code> option. In any case, the debugger then waits for the shell or <i>command_line</i> to complete. Upon returning to the debugger, <code>\$result</code> contains the <i>exit status</i> of the shell.</p> <p>As with breakpoints, <i>command_line</i> may be enclosed in “{ }” to delimit it from other (debugger) commands on the same line. For example,</p> <pre>b 14 {!{date};c}; t; la</pre> <p>sets a breakpoint at line 14 that calls <i>date(1)</i>, then continues; then (after setting the breakpoint), the debugger does a stack trace, then lists assertions.</p>

H

## Miscellaneous Commands

**Table H-18. Miscellaneous Commands (continued)**

Cmd	Syntax	Description
#	# [ <i>text</i> ]	Causes the <i>text</i> to be interpreted as a comment. The number symbol (#) must be the first non-blank character on the line.
Return	Return	<p>Repeats the previous command. You can use this command after the following commands:</p> <ul style="list-style-type: none"> <li>■ +</li> <li>■ -</li> <li>■ p (print)</li> <li>■ v (view)</li> <li>■ +r</li> <li>■ -r</li> <li>■ s (step)</li> <li>■ S (Step)</li> <li>■ up</li> <li>■ down</li> </ul> <p>The number of lines to move is repeated if the previous command was + or -. Otherwise, any <i>count</i> associated with the previous command is discarded.</p>

H

## Miscellaneous Commands

**Table H-18. Miscellaneous Commands (continued)**

Cmd	Syntax	Description
~	~	<p>Repeats the previous command. You must use the <b>Return</b> key after typing the ~. You can use this command after the following commands:</p> <ul style="list-style-type: none"> <li>■ +</li> <li>■ -</li> <li>■ p (print)</li> <li>■ v (view)</li> <li>■ +r</li> <li>■ -r</li> <li>■ s (step)</li> <li>■ S (Step)</li> <li>■ up</li> <li>■ down</li> </ul> <p>The number of lines to move is repeated if the previous command was + or -. Otherwise, any <i>count</i> associated with the previous command is discarded.</p>

H



## Miscellaneous Commands

Table H-18. Miscellaneous Commands (continued)

Cmd	Syntax	Description
am	am	<b>activate more</b> Activates (enables) the <i>more</i> feature.
sm	sm	<b>suspend more</b> Suspends the <i>more</i> feature and lets you view the output in a continuous stream.
f	f [ " <i>printf-style-format</i> " ]	<b>format</b> Sets the printing format used by the debugger to print an address. Only the first 128 literal and formatting characters are used. (See the section on <i>printf(3S)</i> in the <i>HP-UX Reference</i> manual for a discussion of valid formats). Using the <b>f (format)</b> command without an argument will reset the format to the default format: 8 hexadecimal digits preceded by "0x".

H

## Miscellaneous Commands

Table H-18. Miscellaneous Commands (continued)

Cmd	Syntax	Description
h	h [ <i>topics</i> ]	<b>help</b> Prints a command summary which describes the syntax and use of each command. The <i>topics</i> include the command names, plus other topics. This facility references the short form of the command only, not the long form. You can use the <b>h help</b> command to get a list of <i>topics</i> other than command names.
M	M M $\begin{bmatrix} t \\ c \end{bmatrix}$ [ <i>expr</i> [ ; <i>expr</i> [ ... ] ] ]	<b>Map</b> Prints the current text ( <i>objectfile</i> ) and core ( <i>corefile</i> ) address maps.  Sets the text ( <i>objectfile</i> ) or the core ( <i>corefile</i> ) address map. The first zero to six map values are set to the <i>expr</i> given. If less than six expressions are given, the remaining map parameters are left unchanged.
tM	tM	<b>toggle maps</b> Toggles the address mapping of <i>corefile</i> between the initial map and the modifiable mapping pair which the user can set with the <b>Mc</b> command.
q	q	<b>quit</b> Quits the debugger after asking for confirmation: enter <b>y</b> (yes) or <b>n</b> (no).

H

## Miscellaneous Commands

**Table H-18. Miscellaneous Commands (continued)**

Cmd	Syntax	Description
ss	ss <i>file</i>	<b>save state</b> Save the current set of breakpoints, macros, and assertions in <i>file</i> . This file can then be used with the -R option to restore this information on another invocation of the debugger <i>on the same object file</i> .
tc	tc	<b>toggle case</b> Toggles case sensitivity; determines whether or not searches or names are case sensitive.

H

## Signal Control Commands

Table H-19. Signal Control Commands

Cmd	Syntax	Description
lz	lz	<b>list signals</b> Lists the current handling of all signals.
z	z [signal] [i] [r] [s] [Q]	<b>zsignal</b> Modifies the <i>signal</i> handling table. The options (which must be adjacent) toggle the appropriate flag: <b>ignore</b> , <b>report</b> , or <b>stop</b> . If <b>Q</b> is present, the new state of the signal is not printed. Note that <b>z signal</b> with no options tells you the state of the selected signal.

H



# Comparison between the xdb and cdb Symbolic Debuggers

---

This appendix provides a comparison between the xdb and cdb symbolic debuggers. In this appendix, cdb refers to cdb, fdb, and pdb. Some debugger features are present in both xdb and cdb on Series 300/400 computers, but not in xdb on Series 600/700/800 computers (and vice versa). These dependencies are not addressed here (e.g., the special variables \$fpa and \$fpa\_reg). Note that cdb is only available on Series 300/400 computers.

Note that “n.a.” indicates that an equivalent command does not exist.

---

## Startup Command File

Table I-1. Startup Command File

xdb	cdb	Description
.xdbrc	.cdbrc .pdbrc .fdbrc	At startup, xdb and cdb execute commands found in the files listed under their respective commands.

---

## Basic Command Form

### Basic Command Form for xdb

`command` [*location*] [*arguments*] [*command-list*]

### Basic Command Form for cdb

[*modifier*] `command` [*arguments*] [*command-list*]

---

## Variable Name Conventions

Table I-2. Variable Name Conventions

xdb	cdb	Description
<i>proc:var</i>	<i>proc.var</i>	Search the stack for the most recent instance of <i>proc</i> (procedure, function, subroutine). If found, see if it has a parameter or local variable named <i>var</i> , as before.
<i>proc:depth:var</i>	<i>proc.depth.var</i>	Use the instance of <i>proc</i> (procedure, function, subroutine) that is at depth <i>depth</i> (exactly), instead of the most recent instance. This is very useful for debugging recursive procedures where there are multiple instances on the stack.

---

## Special Variables

**Table I-3. Special Variables**

<b>xdb</b>	<b>cdb</b>	<b>Description</b>
<code>\$step</code>	<code>\$cBad</code>	Lets you see and modify the number of machine instructions the debugger will step while in a non-debuggable procedure before setting an up-level breakpoint and free-running to it. Setting it to a small value can improve debugger performance at the risk of taking off free-running after missing the up-level break for some reason.
n.a.	<code>\$pagelines</code>	Lets you set the number of lines per "page" of debugger output. The prompt "--More--" occurs between pages. Values of zero or less turn off paging.

---

## Expression Conventions

In `xdb`, expression values that are not command modifiers are not printed unless that expression is used with an `xdb print (p)` command. In `cdb`, expression values that are not command modifiers (stand-alone expressions) are always printed unless the next token is ";" (a command separator) or "}" (a command block terminator). Therefore, breakpoint and assertion commands are normally silent. To force an expression result to be printed when using `cdb`, follow the expression with `"/n"` (print in normal format).



---

## The Debugger Special Variable \$lang

In `xdb`, the initial value of the debugger special variable `$lang` is automatically set by the language type of the procedure being viewed (for example, `main()`). For `cdb`, `fdb`, or `pdb`, the initial value of `$lang` is determined by the symbolic debugger that is invoked.

---

## Division Operator

**Table I-4. Division Operator**

<code>xdb</code>	<code>cdb</code>	Description
<code>/</code>	<code>//</code>	Division operator

---

## Command-Line Editing Environment Variables

**Table I-5. Command-Line Editing Environment Variables**

<code>xdb</code>	<code>cdb</code>	Description
<code>XDBEDIT</code>	<code>CDBEDIT</code>	Environment variable that determines which of the three available editing modes ( <code>vi</code> , <code>emacs</code> , or <code>gmacs</code> ) is used.
<code>XDBHIST</code>	<code>CDBHIST</code>	Environment variable that specifies the command history file.

---

## Split-Screen Mode

When using the `ts` command in `xdb`, the step size (source line or instruction) is determined by screen mode before the `ts` command was executed. It may be toggled with the `td` command. When using the `ts` command in `cdb`, the step size is determined by the command used (`s` or `j`).

---

## Single-Stepping Commands

**Table I-6. Single-Stepping Commands**

<code>xdb</code>	<code>cdb</code>	Description
<code>s [count]</code>	<code>[count] j</code>	Single step 1 (or <i>count</i> ) disassembly instruction. Successive carriage-returns repeat with a <i>count</i> of 1.
<code>S [count]</code>	<code>[count] J</code>	Single step like <code>s</code> , but treat procedure calls as single instructions (do not follow them down).

Note that the `xdb` commands `s` and `S` are also used to step source level statements.

## File Viewing Commands

Table I-7. File Viewing Commands

xdb	cdb	Description
w [ <i>size</i> ]	ws [ <i>size</i> ]	Set the size of the source viewing window.
v [ <i>location</i> ]	e [ <i>location</i> ]	View the source at the specified <i>location</i> .
V [ <i>depth</i> ]	[ <i>depth</i> ] E	View current procedure at <i>depth</i> on the stack.
va [ <i>address</i> ]	n.a.	View the assembly code at the specified <i>address</i> in the source window.
v <i>line</i>	<i>line</i>	View the source <i>line</i> number in the current file.
n.a.	[ <i>line</i> ] p [ <i>count</i> ]	View one (or <i>count</i> ) line(s) starting at the current line (or <i>line</i> number).
n.a.	[ <i>line</i> ] w [ <i>size</i> ]	For the line mode interface, print a window of text containing <i>size</i> (default 11) lines centered around the current or specified <i>line</i> .
n.a.	[ <i>line</i> ] W [ <i>size</i> ]	Same as w given above, but <i>size</i> defaults to 21 lines.
D " <i>directory</i> "	dir " <i>directory</i> "	Add directory to current search path for source file.

**Table I-7. File Viewing Commands (continued)**

xdb	cdb	Description
n.a.	$\left\{ \begin{array}{l} +w \\ +W \end{array} \right\} [size]$	View a window of text of given or default <i>size</i> , ending at the end of the previous window if the previous command was a window command; otherwise, at the current line.
n.a.	$\left\{ \begin{array}{l} -w \\ -W \end{array} \right\} [size]$	View a window of text of given or default <i>size</i> , ending at the beginning of the previous window if the previous command was a window command; otherwise, at the current line.

## Data Viewing Commands

Table I-8. Data Viewing Commands

xdb	cdb	Description
<i>p expr</i>	<i>expr</i>	If <i>expr</i> does not look like anything else (such as a command), it is handled as if you had typed <i>p expr\n</i> (print expression in normal format). When using <i>cdb</i> , if <i>expr</i> does not resemble anything else (such as a command), it is handled as <i>expr/n</i> (print expression in normal format), unless followed by “;” or “}”, in which case nothing is printed (although it is evaluated).
<i>p expr\format</i>	<i>expr/format</i>	Print the contents (value) of <i>expr</i> using <i>format</i> .
<i>p expr?format</i>	<i>expr?format</i>	Print the address of <i>expr</i> using <i>format</i> .
<i>pq expr</i>	<i>expr;</i>	Print quiet.

**Table I-8. Data Viewing Commands (continued)**

xdb	cdb	Description
p [-[\] <i>format</i> ]	^ [ [/\] <i>format</i> ]	Back up to the preceding memory location (based on the size of the last thing displayed). Use <i>format</i> if supplied, or the previous <i>format</i> if not.
p [+[\] <i>format</i> ]	n.a.	Go forward to the following memory location (based on the size of the last thing displayed). Use <i>format</i> if supplied, or the previous <i>format</i> if not.
p <i>class</i> ::	<i>class</i> ::	Print all of the static members of <i>class</i> .
l [ <i>proc</i> [: <i>depth</i> ]]	l [ [ <i>proc</i> [. <i>depth</i> ] ]	Lists all parameters and local variables for the current procedure (or <i>proc</i> , if given, at the specified <i>depth</i> , if any).

## Stack Viewing Commands

**Table I-9. Stack Viewing Commands**

<b>xdb</b>	<b>cdb</b>	<b>Description</b>
<code>t [depth]</code>	<code>[depth] t</code>	Trace the stack for the first <i>depth</i> (default 20) levels.
<code>T [depth]</code>	<code>[depth] T</code>	The same as <code>t</code> , but local variables are also displayed, using <code>\n</code> (for <code>cdb</code> , <code>/n</code> ) format (except that all arrays and pointers are shown simply as addresses, and structures as first words only).

## Job Control Commands

**Table I-10. Job Control Commands**

xdb	cdb	Description
c [ <i>location</i> ]	[ <i>count</i> ] c [ <i>line</i> ]	Continue from a breakpoint ignoring the signal. Set a temporary breakpoint at the specified <i>location</i> . When using <i>cdb</i> , if <i>count</i> is given, the current breakpoint, if any, has its <i>count</i> set to that value and if <i>line</i> is given, a temporary breakpoint is set at that line number with a <i>count</i> of -1.
C [ <i>location</i> ]	[ <i>count</i> ] C [ <i>line</i> ]	Same as “c” above, but allow the signal (if any) to be received.
s [ <i>count</i> ]	[ <i>count</i> ] j	Single step 1 (or <i>count</i> ) assembly statement(s).
S [ <i>count</i> ]	[ <i>count</i> ] J	Single step like <i>s</i> , but treat procedure calls as single statements (do not follow them down).
s [ <i>count</i> ]	[ <i>count</i> ] s	Single step 1 (or <i>count</i> ) source statement(s).
S [ <i>count</i> ]	[ <i>count</i> ] S	Single step like “s”, but treat procedure calls as single statements (do not follow them down).



## Breakpoint Counts

Table I-11. Breakpoint Counts

xdb	cdb	Description
<i>breakpoint-command</i> \num [t   p]	n.a.	Number of times the breakpoint is encountered prior to recognition.
<i>numberp</i>	<i>number</i> > 0	Permanent breakpoint count.
<i>numbert</i>	<i>number</i> < 0	Temporary breakpoint count.
<i>bc number count</i>	n.a.	Explicitly modifies the count for an existing breakpoint.

Breakpoint counts are handled in different ways on **xdb** and **cdb**. For **xdb**, the *count* can be explicitly given with the breakpoint command itself. For example:

```
b \10 add_file
```

sets a permanent breakpoint at a function called `add_file` with a *count* of 10.

For **cdb**, a breakpoint *count* may be specified only from a continue command (`c` or `C`). For example:

```
10 c
```

sets the *count* to 10 on the breakpoint at the current location.

Both **xdb** and **cdb** handle positive breakpoint counts as designating breakpoints which are permanent, that is, not automatically removed when recognized.

Negative breakpoint counts signify temporary breakpoints which are deleted upon recognition. For **xdb**, a *count* of 1 may be specified by `p` and a *count* of -1 by `t`. The `bc` command is also only available on **xdb** to modify the *count* for an existing breakpoint.

## Breakpoint Commands

**Table I-12. Breakpoint Commands**

xdb	cdb	Description
lb	B or lb	List all breakpoints
b [ <i>location</i> ] [\count] [ <i>commands</i> ]	[ <i>line</i> ] b [ <i>commands</i> ] or b [ <i>location</i> ] [ <i>commands</i> ]	Set a breakpoint at the <i>location</i> or <i>line</i> .
db [ <i>number</i> ]	[ <i>number</i> ] d	Delete breakpoint with given <i>number</i> .
db *	D [b]	Delete all breakpoints (including “all-procedure” breakpoints).
dp	D p	Delete all “procedure entry” breakpoints.
bb [ <i>depth</i> ] [\count] [ <i>commands</i> ]	[ <i>depth</i> ] bb [ <i>commands</i> ] or [ <i>depth</i> ] bB [ <i>commands</i> ]	Set a breakpoint at the beginning (first executable line) of the current procedure (or procedure at the given stack <i>depth</i> ).
bx [ <i>depth</i> ] [\count] [ <i>commands</i> ]	[ <i>depth</i> ] bx [ <i>commands</i> ] or [ <i>depth</i> ] bX [ <i>commands</i> ]	Set a breakpoint at the exit (last executable line) of the current procedure (or procedure at the given stack <i>depth</i> ).

**Table I-12. Breakpoint Commands (continued)**

xdb	cdb	Description
bu [ <i>depth</i> ] [\count] [ <i>commands</i> ]	[ <i>depth</i> ] bu [ <i>commands</i> ] or [ <i>depth</i> ] bU [ <i>commands</i> ]	Set an up-level breakpoint. The breakpoint is set immediately after the return to the procedure at the specified stack <i>depth</i> (default one, not zero).
bt [ <i>depth</i>   <i>proc</i> ] [\count] [ <i>commands</i> ]	[ <i>depth</i> ] bt [ <i>proc</i> ] [ <i>commands</i> ] or [ <i>depth</i> ] bT [ <i>proc</i> ] [ <i>commands</i> ]	Trace the current procedure (or procedure at <i>depth</i> , or <i>proc</i> ).
ba [ <i>address</i> ] [\count] [ <i>commands</i> ]	[ <i>address</i> ] ba [ <i>commands</i> ] or [ <i>address</i> ] bA [ <i>commands</i> ]	Set a breakpoint at the given code address.
bc <i>number count</i>	n.a.	Set the count of the existing breakpoint identified by <i>number</i> to <i>count</i> .

---

## Assertion Evaluation

In **xdb**, assertions are lists of commands that are executed before every instruction. In **cdb**, assertions are lists of commands that are executed before every statement.

---

## Assertion Commands

**Table I-13. Assertion Commands**

<b>xdb</b>	<b>cdb</b>	<b>Description</b>
<i>aa number</i>	<i>number aa</i>	Activate assertion <i>number</i> .
<i>aa *</i>	n.a.	Activate all assertions.
<i>da number</i>	<i>number da</i>	Delete assertion <i>number</i> .
<i>da *</i>	D a	Delete all assertions.
<i>sa number</i>	<i>number sa</i>	Suspend assertion <i>number</i> .
<i>sa *</i>	n.a.	Suspend all assertions.
<i>ta</i>	A	Toggle the overall assertions mode between <i>active</i> and <i>suspended</i> .
<i>x [expr]</i>	<i>[expr] x</i>	Force an exit from assertion mode.

---

## Signal Command

**Table I-14. Signal Command**

<b>xdb</b>	<b>cdb</b>	<b>Description</b>
<i>z [signal] [i] [r] [s] [Q]</i>	<i>[signal] z [i] [r] [s] [Q]</i>	Modifies the signal handling table for the given signal.

---

## Toggle Recording

**Table I-15. Toggle Recording**

<b>xdb</b>	<b>cdb</b>	<b>Description</b>
<b>tr [0]</b>	n.a.	Toggle recording (i.e., if it is <b>ON</b> turn it <b>OFF</b> and if it is <b>OFF</b> turn it <b>ON</b> ).

---

## Toggle Case Sensitivity

**Table I-16. Toggle Case Sensitivity**

<b>xdb</b>	<b>cdb</b>	<b>Description</b>
<b>tc</b>	<b>Z</b>	Toggles case sensitivity (i.e., if it is <b>ON</b> turn it <b>OFF</b> and if it is <b>OFF</b> turn it <b>ON</b> ).

---

## Save-State

**Table I-17. Save-State**

<b>xdb</b>	<b>cdb</b>	<b>Description</b>
<b>ss</b>	<b>ss</b>	In <b>xdb</b> , <b>ss</b> saves the current value of <i>count</i> for breakpoints. In <b>cdb</b> , <b>ss</b> does not save the current value of <i>count</i> for breakpoints.

## Registers Displayed by the HP Symbolic Debugger in Disassembly Mode

---

This appendix lists the registers displayed by the HP Symbolic Debugger in disassembly mode for Series 300/400 and Series 600/700/800 computers.

### Register Names for Series 600/700/800 Computers

#### Special Variables Names Used for Registers

Register(s)	Description
\$r0 .. \$r31	General registers
\$f0 .. \$f31	Floating-point (64 bit) registers <sup>1</sup>
\$f0r .. \$f31r	Floating-point registers, right half (the 32 least-significant bits of the registers f0 through f31) <sup>1</sup>
\$f0l .. \$f31l	Floating-point registers, left half (the 32 most-significant bits of the registers f0 through f31) <sup>1</sup>
\$fpstat	Pseudonym for the register \$f0l
\$pc	Program counter (IAOQ-head)

<sup>1</sup> Floating-point registers \$f16 through \$f31 are available only on PA-RISC version 1.1 machines.

## Special Variables Names Used for Registers (Continued)

Register(s)	Description
\$sp	Stack pointer; pseudonym for \$r30
\$dp	Global data pointer; pseudonym for \$r27
\$arg0 .. \$arg3	Pseudonyms for \$r26 .. \$r23
\$ret0 .. \$ret1	Pseudonyms for \$r28 .. \$r29

## Registers Displayed in the General or Floating-Point Register Windows

Register(s)	Description
r0 .. r31	General registers
f0 .. f31	Floating-point (64 bit) registers <sup>1</sup>
pc	Program counter; IASQ-head.IAOQ-head
priv	Privilege level, IAOQ[30 .. 31]
psw	Process status word (lowercase means a 0 bit; uppercase means a 1 bit)
sar	Shift amount register, CR11[27 .. 31]
fpsr	Floating-point coprocessor status flags (lowercase means a 0 bit; uppercase means a 1 bit)
RM	Rounding mode (from coprocessor status word)
enable	Enable flags for the coprocessor (from coprocessor status word)

<sup>1</sup> Floating-point registers f16 through f31 are available only on PA-RISC version 1.1 machines.

## Registers Displayed in the Special Register Window

Register(s)	Description
tr0 .. tr7	Temporary registers, CR24 .. CR31
sr0 .. sr7	Space registers
pid1 .. pid4	Protection id's, CR8, CR9, CR12, CR13
ccr	Coprocessor configuration register, CR10
sar	Shift amount register, CR11
eiem	External interrupt enable mask, CR15
itmr	Internal timer, CR16
isr	Interruption space register, CR20
iva	Interruption vector address, CR14
rctr	Recovery counter, CR0
eirr	External interrupt request register, CR23
ior	Interruption instruction register, CR21
iir	Interruption instruction register, CR19
pch	IASQ-head.IAOQ-head
pct	IASQ-tail.IAOQ-tail
priv	Privilege level, IAOQ[30 .. 31]
psw	Process status word

J



---

## Register Names for Series 300/400 Computers

### Special Variable Names Used for Registers

Register(s)	Description
\$a0 .. \$a7	Address registers
\$d0 .. \$d7	Data registers
\$ps	Status register
\$pc	Program counter
\$fp	Frame pointer; pseudonym for \$a6
\$sp	Stack pointer; pseudonym for \$a7

### Registers Displayed in the General and Floating-Point Register Window

Register(s)	Description
d0 .. d7	Data registers
a0 .. a7	Address registers
pc	Program counter
ps	Status register
fp0 .. fp7	MC68881/MC68882 floating-point registers
fpsr	MC68881/MC68882 status registers
fpcr	MC68881/MC68882 control registers
fpa0 .. fpa7	HP 98248 floating-point registers (Series 300 only)
fpasr	HP 98248 status registers (Series 300 only)
fpacr	HP 98248 control registers (Series 300 only)

# Glossary

---

## address

Virtual memory address used to reference program code or data. When used to designate an address with the `ba` (**breakpoint address**) command, it can be either one of the following:

- Strictly a numeric value (such as `0x00001358`)
- A symbolic address with or without an offset (such as `main+0x1c`).

## archive library

An archive library 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 `a.out` file. Note that archive library names end with `.a`.

## assertion

A list of commands performed before the debugger executes each program statement. Useful for tracking unexpected changes in program data (undesired side effects).

**breakpoint**

A software “trigger” inserted into the user program, that, when encountered during execution, pauses the program 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 line number, procedure entry or exit point, etc.

In general, breakpoints can have the following associated with them:

- *command list*- list of commands executed when the breakpoint is triggered
- *count*- how many times the breakpoint must be encountered before it is triggered.
- *lifespan*- “temporary” or “permanent” status (this information is actually determined by whether count is less than or greater than zero, respectively). A temporary breakpoint is removed when it is triggered; a permanent breakpoint is not.

**child process**

A subordinate process that is initiated and closely controlled by the debugger (parent). This process is a running instance of the program being debugged.

**command**

Commands tell the HP Symbolic Debugger which functions to perform, and can be spelled out or abbreviated. The abbreviation for most commands is the first character of each word in the command name. Commands are separated with a semicolon within a command list. For more information, see Chapter 4 “HP Symbolic Debugger Commands.”

**command list****Glossary**

A sequence of one or more debugger commands separated by a semicolon (;). Some commands expect command-lists as arguments. Braces ({} ) must sometimes be used to enclose command-lists. For more information, see the individual command listings in Chapter 4 “HP Symbolic Debugger Commands.”

**coprocessor**

This is an additional processor used in conjunction with the main processor for speeding up and reducing the workload of the main processor. For example, the floating-point coprocessor speeds up the mathematical computations of the system.

**corefile**

This is the core image of an executable file resulting from an aborted execution of that file.

**current location**

The “point-of-interest” in the source as displayed in the source window. Many commands take this as a default location. The current location is not necessarily the current point of program suspension (where the program is currently paused.)

**debugger 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.

**depth**

Number of levels back in the current procedure call chain (stack). Depth 0 is where execution is suspended. If procedure *A* calls *B*, procedure *B* calls *C*, and *C* is where the program is suspended, then *B* is at depth 1 and *A* is at depth 2. The **t** (**trace**) or **T** (**Trace**) commands display the procedures and their depths on the stack. (See **stack** in this appendix.)

**exception**

Either a hardware or software generated condition that causes the program to be asynchronously suspended or halted. Examples of these might be:

- user-generated (keyboard) interrupt
- floating-point overflow
- segmentation violation (invalid addressing operation)
- bus error (invalid memory access)
- other signals (see *signal(4)* in the *HP-UX Reference*)

**expression**

A valid combination of data object names, language operators, and constant numeric values. Every expression is evaluated and reduced to a single value.

**format**

Used with the debugger command **p (print)** to describe how data will be accessed and displayed. A format consists of:

- an optional repetition count
- a formatting character
- an optional object size

The access and display operation is performed once for each repetition (default 1). The number of bytes in each object is determined by the given object size (default depends on the formatting character). The formatting character determines how each object is interpreted and printed. For example, to print four sequential 16-bit integers in octal, use the format `4o2` or `4os`.

**Glossary**

**line mode**

Debugger user interface that does not use any special terminal functions. This must be used for terminals that do not support window mode.

**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 (see above) can also be used to specify a location for certain commands.

**machine instruction**

Presented to the user when debugging in disassembly mode. Actual instruction mnemonics and syntax are described in the *HP Precision Architecture and Instruction Reference Manual* or *HP-UX Assembler and Tools*.

**macro**

Simple form of command aliasing using text substitution. A macro can be used as a shorthand for one or more commands.

**memory lock**

A terminal feature that allows some upper portion of the terminal screen to remain constant while the remainder of the screen is scrolled. This feature is required by the debugger for its window-oriented interface. If memory lock is unavailable, the line-oriented interface (line mode) is used.

**procedure**

A procedure, function, subroutine, or module name. Also a user program name.

### registers

Precision Architecture (Series 600/700/800 computers) or MC680x0 (Series 300/400 computers) hardware registers. Most of these are directly accessible by the debugger through symbolic names (e.g. `$pc`). Many registers have special meaning; some cannot be modified by the debugger user. See the *HP Precision Architecture and Instruction Reference Manual* or *HP-UX Assembler and Tools* for a discussion on the use of each register. Actual modification of hardware registers should not normally be necessary while debugging. Correct program execution depends highly on registers and their contents.

### shared library

Like an archive library, a shared library contains relocatable object code. However, `ld` treats shared libraries quite differently than archive libraries. When linking an object file with a shared library, `ld` does not copy object code from the library into the `a.out` file; instead, the linker simply notes in the `a.out` file that the code calls a routine in the shared library. The actual linkage does not occur until the program is run. Note that shared library names end with `.sl`.

### source

Source text (files) used to compile the user program. These can be in any of the programming languages supported by the debugger.

### source line

A single line of text in a source file, denoted by a line number. A source line might or might not contain actual executable statements. Conversely, more than one statement can occur on a single line.

### special variables

Named variable (prefixed by `$`) local to the debugger. Many special variables are predefined by the debugger to have a unique meaning. For example, `$line` is always the current line number, and `$dp` is the data-pointer register (Series 600/700/800 general register 27).

User-defined special variables are also available. They are created when first referenced, and allow you to store and reference numeric variables independent of the program being debugged.

## Glossary

**stack**

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 **T (Trace)** command) will display information contained in each stack frame; in particular, the values of all local variables. (See **depth** in this appendix.)

**string**

Quoted sequence of arbitrary characters. Quotes can be single (') or double (") depending on the current language (**\$lang**). Character escapes allow inclusion of control or other non-printing characters.

**stub**

(Series 600/700/800 only) 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, read the *PA-RISC Procedure Calling Conventions Reference Manual* (09740-90015).

**window**

Region of the terminal screen limited to displaying specific information. The debugger has at least three: the source, location, and command windows.

**window mode**

A display mode where the debugger divides the terminal screen up into regions dedicated to the display of specific information (see the section "Terminal Support" in the chapter "Introducing the HP Symbolic Debugger"). Note that this *does not* refer to the X Window System.





# Index

---

## Special characters

!, 4-112, B-2, H-45-50  
 !=, B-3  
 #, 4-112, 4-113, H-45-50  
 %, B-2  
 &, B-2, B-3  
 &&, B-3  
 ( ), B-2, C-2, D-2  
 \*, B-2, C-2, D-2  
 +, 3-18, 4-28, 4-113, 4-114, B-2, C-2, D-2, H-7-9  
 -, 3-18, 4-28, 4-113, 4-114, H-7-9  
 ->, B-2  
 ., 3-32, B-2, C-2, D-2  
 /, 3-27, 4-28, B-2, C-2, D-2, H-7-9, I-4  
 //, I-4  
 ::, B-2  
 :=, D-3  
 <, B-3, D-3  
 <<, B-3  
 <=, B-3, D-3  
 < >, D-3  
 =, C-3, D-3  
 = , B-3  
 ==, B-3  
 >, 4-102, B-3, D-3, H-42-43  
 >=, B-3, D-3  
 >>, B-3  
 >@, 4-104, H-42-43  
 ?, 3-27, 4-28, H-7-9  
 [ ], B-2, D-2  
 ~, B-3, D-2

|, B-3  
 ||, B-3  
 ~, 4-112, 4-114, H-45-50  
 ~, B-2

## A

a, 3-51  
 \$a0 .. \$a7, J-4  
 a0 .. a7, J-4  
 aa (**activate assertion**), 4-97, 4-99, H-40-41, I-15  
 a (**assert**), 4-97, H-40-41  
 ab (**activate breakpoint**), 4-70, H-32-33  
 abbreviating commands, 4-6  
 abc, 4-71, 4-92, H-37  
**activate breakpoint**, 4-86  
**activate more**, 3-20, 4-114  
 \$addr, B-2, C-2, D-2  
 addr, D-2  
 address, Glossary-1  
*address*, 4-4-5  
 address format restriction, F-12  
 adopted process, debugging shared libraries, 6-12  
 adopting  
   a running process, 3-5, 3-55  
   shared libraries, 6-12  
 all-procedures breakpoint commands  
   bp, H-33-36  
   bpt, 4-89, H-33-36  
   bpx, 4-90, H-33-36

dp, 4-90, H-33-36  
 Dpt, 4-91, H-33-36  
 Dpx, 4-91, H-33-36  
 altering execution sequence, 3-53  
 am (activate more), 3-20, 4-112, 4-114,  
   H-45-50  
 and, D-3  
 .AND., C-2  
 anonymous unions, 5-40  
 "any string", 4-71, 4-93, H-38  
 apm (add path map), 3-62, 4-35, **4-35**,  
   H-18  
 archive library, Glossary-1  
 \$arg0 .. \$arg3, J-2  
 arrays, 4-20  
 assembly code, displaying, 3-22, 3-24  
 assert, **4-98**  
 assertion, Glossary-1  
 assertion, 3-51  
 assertion commands, I-15  
 assertion control commands  
   a, 4-98, H-40-41  
   aa, 4-99, H-40-41  
   da, 4-100, H-40-41  
   la, 4-100, H-40-41  
   sa, 4-100, H-40-41  
   ta, 4-101, H-40-41  
   x, 4-101, H-40-41  
 assertion evaluation, I-14  
 assertions, hints for using, F-12  
 assignment, 4-19  
 auxiliary breakpoint commands  
   "any string", 4-93, H-38  
   i, 4-93, H-38  
   Q, 4-94, H-38

## B

-b, 3-4  
 b, 3-28  
 B, I-13-14  
 bA, I-13-14

ba (breakpoint address), 4-70,  
   H-26-31, I-13-14  
 basic command form for  
   cdb, I-2  
   xdb, I-2  
 bB, I-13-14  
 bb (breakpoint beginning), 4-70,  
   H-26-31, I-13-14  
 b (breakpoint), 4-70, 5-69, H-26-31,  
   I-13-14  
 bc (breakpoint count), 4-70, H-32-33,  
   I-12, I-13-14  
 bi (breakpoint instance), 3-29, 4-70,  
   5-54, H-26-31  
 bp (breakpoint procedure), 3-28,  
   4-70, H-33-36  
 bpc (breakpoint class), 4-70, 5-53,  
   5-67, 5-68, H-26-31  
 bpo (breakpoint overload), 3-29,  
   4-70, 5-52, 5-68, H-26-31  
 bpt, 3-29, 4-70, 4-89, H-33-36  
 bpx, 3-29, 4-70, 4-90, H-33-36  
 breakpoint, Glossary-2  
 breakpoint, 3-28, **4-79**, 5-69  
 breakpoint address, **4-80**  
 breakpoint at all member functions of  
   a class, setting, 5-53  
 breakpoint beginning, **4-81**  
 breakpoint class, **4-83**, 5-67, 5-68  
 breakpoint commands, 5-50, I-13  
 breakpoint count, **4-86**  
 breakpoint counts, I-12  
 breakpoint creation commands  
   b, H-26-31  
   ba, 4-80, H-26-31  
   bb, 4-81, H-26-31  
   bi, 4-81, H-26-31  
   bpc, 4-83, H-26-31  
   bpo, 4-84, H-26-31  
   bt, 4-82, H-26-31  
   bu, 4-84, H-26-31

- bx, 4-85, H-26-31
  - breakpoint exit, 4-85
  - breakpoint instance, 3-29, 4-81, 4-81
  - breakpoint on function, setting, 5-50
  - breakpoint on overloaded function, setting, 5-52
  - breakpoint overload, 4-84
  - breakpoint procedure, 3-28, 4-88
  - breakpoints, 3-28
    - command list, 4-72
    - count, 4-72
    - location, 4-72
    - setting, 3-28
  - breakpoint status commands
    - ab, H-32-33
    - bc, 4-86, H-32-33
    - db, 4-86, H-32-33
    - sb, 4-87, H-32-33
  - breakpoint trace, 4-82
  - breakpoint types, 4-74
  - breakpoint uplevel, 4-84, 5-83
  - bT, I-13-14
  - bt (breakpoint trace), 4-70, H-26-31, I-13-14
  - bU, I-13-14
  - bu (breakpoint uplevel), 4-70, H-26-31, I-13-14
  - bX, I-13-14
  - bx (breakpoint exit), 4-70, H-26-31, I-13-14
- C**
- >@c, 4-104, H-42-43
  - >c, 4-104, H-42-43
  - c, 3-29, 3-30, I-11
  - C, 1-4
  - C, I-11
  - C++, 1-4
  - C and C++
    - Language operators, B-1
  - C++ and the symbolic debugger, 5-1
  - capture and rerun a debugger session, 1-2
  - capturing a debugger session, 3-41
  - case insensitivity, 3-9, 3-27, 4-6, 4-118
  - case sensitivity, 3-9, 3-27, 4-5, 4-118, 6-8
  - casts, 4-20, 5-36
  - catch, 5-57, 5-57, 5-62
  - catch, stopping on, 5-61
  - \$cBad, I-3
  - c (continue), 4-64, H-22-24
  - C (Continue), 4-64, H-22-24
  - ccr, J-3
  - C++ data types, 5-2
  - cdb, 1-1
  - cdb and xdb comparison, I-1
  - cdb, basic command form, I-2
  - CDBEDIT, I-4
  - C++\_demo, 2-4
  - C\_demo, 2-2
  - C++ expressions, 5-10
  - changing execution sequence, 3-53
  - character and string expressions, 4-16
  - characters, display non-ASCII, 3-44
  - child process, 3-5, Glossary-2
  - class
    - commands, 5-4
    - members, 5-28
    - objects, 5-18
    - scope, 5-5
    - template, all member functions, 5-67
    - template member function, a single, 5-69
    - template member function instance, a single, 5-69
    - templates, listing, 5-72
  - class, 4-2
  - classes
    - and objects, 5-4
    - listing, 5-72
    - nested, 5-74

- class names of an enclosed class, 5-75
- C++ main source file, 2-12
- C main source file, 2-10-11
- COLUMNS**, 3-10, E-4
- command, Glossary-2
- command history, 1-10
- command-line
  - editing environment variables, I-4
  - procedure call limitations, F-7
- command-list*, 4-2
- command list, Glossary-2
- commands
  - xdb, 3-41
- commands, entering, 4-1
- command window, 3-14, 3-20
- compiler, symbolic debugger options, 3-3
- compiling a program, symbolic debugger information, 3-3
- COMPLEX** variables, F-6
- composite types, 4-20
- continue**, 3-29, **3-30**, 4-65
- Continue**, **4-65**
- conventions, expression, I-3
- coprocessor, Glossary-3
- coredumped, debugging a program that, 3-57
- corefile, Glossary-3
- count*, 4-2
- \$cplusplus**, 4-12, 5-21, 5-76, E-1-3
- creating an executable program file, 1-7
- C++ scope rules, 5-2
- current location, Glossary-3
- customizing default debugger behavior, 5-76
- customizing the debugger environment, 3-9
- D**
- d, I-13-14
- D, I-6-7, I-13-14
- \$d0 .. \$d7**, J-4
- d0 .. d7**, J-4
- da (delete assertion)**, 4-97, 4-100, H-40-41, I-15
- data members, 5-28
- data modification commands, 4-36
- data viewing commands
  - comparisons between **xdb** and **cdb**, I-8
  - l**, 4-36, H-10-17
  - lc**, 4-38, H-10-17
  - lcl**, 4-39, H-10-17
  - lct**, 4-39, H-10-17
  - lft**, 4-39, H-10-17
  - lg**, 4-40, H-10-17
  - ll**, 4-40, H-10-17
  - lm**, 4-41, H-10-17
  - lo**, 4-42, H-10-17
  - lp**, 4-42, H-10-17
  - lr**, 4-43, H-10-17
  - ls**, 4-44, H-10-17
  - lsl**, 4-45, H-10-17
  - ltf**, 4-45, H-10-17
  - lx**, 4-46, H-10-17
  - p**, 4-47, H-10-17
  - pq**, 4-55, H-10-17
- data viewing formats, 4-53
- db**, **3-31**
- dbc**, 4-71, 4-92, H-37
- db (delete breakpoint)**, 4-70, H-32-33, I-13-14
- D command, 3-64
- d** command-line option, 3-64, 4-35
- D (**Directory**), 4-28, H-7-9
- debugger
  - command syntax, 4-2
  - environment, customizing, 3-9
  - information, Glossary-3
  - installation, G-1
  - invocation options, H-1
  - session scenario one, 2-2

- session scenario two, 2-4
- > debugger prompt, 3-14
- debugging
  - shared libraries, 6-1
  - shared libraries in an adopted process, 6-12
- debugging sessions for C++, 5-77
- declaration statement scope, 5-7
- def**, 3-52, 4-105, 4-106, H-44
- delete assertion**, 4-100
- delete breakpoint**, 3-31, 4-86
- delete procedure**, 4-90
- deleting all-procedure breakpoints, 3-31
- deleting breakpoints, 3-31
- \$depth**, 3-38
- depth, Glossary-3
- depth*, 4-2
- dir**, I-6-7
- Directory**, 4-32
- directory mapping, source, 3-62, 4-35
- disassembly mode, 3-3, 3-22, 3-24
  - shared libraries, 6-10
- disassembly mode limitations, F-10
- display
  - non-ASCII characters, 3-44
- display and modify variables, 1-2
- displaying
  - assembly code, 3-22
  - character data and using NLS, 3-43
  - data, 3-32
  - lines, 3-18
  - source and assembly code, 3-24
  - static data members, 5-42
  - the contents of an object, 5-23
  - type information for an object, 5-19
- div**, D-2
- division operator, I-4
- DMEM**, 6-8, 6-10
- down**, 4-56, 4-60, H-19-20
- down command**, 3-38
- \$dp**, J-2
- dp**, 3-31, 3-31, 4-90
- D p**, I-13-14
- dp (delete procedure)**, 4-70, H-33-36, I-13-14
- dpm (delete path map)**, 4-35, 4-35, H-18
- Dpt**, 3-31, 4-70, 4-91, H-33-36
- Dpx**, 3-31, 4-70, 4-91, H-33-36
- E**
- e**, I-6-7
- EDITOR**, 1-10
- eiem**, J-3
- eirr**, J-3
- enable**, J-2
- enclosed class
  - class names, 5-75
  - static members, 5-74
- ending a program, 3-17
- ending the symbolic debugger, 3-17
- environment variables, 3-9, E-1, E-4
- environment variables, command-line
  - editing, I-4
- .EQ.**, C-2
- .EQV.**, C-3
- error messages
  - debugger, A-1
  - user, A-1, A-3
- escape sequences, 4-16
- examine core files, 1-3
- exception, Glossary-4
- exception catch command**, 5-62
- exception handling, 5-4, 5-57
- exception handling commands
  - txc**, 4-96, H-39
  - txt**, 4-95, H-39
  - xcc**, 4-96, H-39
  - xtc**, 4-96, H-39
- exceptions, listing, 5-63
- exception throw command**, 5-60
- executable program file, 1-7

executing  
  commands before machine instructions,  
  1-2  
executing a program, 3-5  
executing commands at each source line,  
  3-51  
execution stack, navigating, 3-37  
**exit**, 4-101  
export stub, 4-41, 4-61  
*expr*, 4-2  
expression, Glossary-4  
  conventions, I-3  
  entering, 4-15

## F

>**@f**, 4-104, H-42-43  
>**f**, 4-104, H-42-43  
**f**, 4-116  
**\$f0** .. **\$f31**, J-1  
**f0** .. **f31**, J-2  
**\$f0l** .. **\$f31l**, J-1  
**\$f0r** .. **\$f31r**, J-1  
**fdb**, 1-1  
  **.fdbrc**, I-1  
**f (format)**, 4-112, H-45-50  
<<*file*, 4-102, H-42-43  
<*file*, 4-102, H-42-43  
>>**@file**, 4-104, H-42-43  
>>*file*, 4-102, H-42-43  
>**@file**, 4-104, H-42-43  
>*file*, 4-102, H-42-43  
*file*, 4-2-4  
files restrictions, F-6  
file viewing commands, I-6  
  +, 4-29, H-7-9  
  -, 4-29, H-7-9  
  /, 4-31, H-7-9  
  ?, 4-31, H-7-9  
  D, 4-32, H-7-9  
  L, 4-34, H-7-9  
  ld, 4-33, H-7-9

**lf**, 4-33, H-7-9  
  **n**, 4-31, H-7-9  
  **N**, 4-32, H-7-9  
  **v**, 4-30, H-7-9  
  **va**, 4-34, H-7-9  
floating-point constants, 4-18  
**floating point registers**, 4-24  
**format**, 4-116  
format, Glossary-4  
*format*, 4-2-4  
formats, data viewing, 4-53  
FORTRAN 77, 1-4  
  Language operators, C-1  
FORTRAN 77 main source file, 2-7  
Fortran\_demo, 2-2  
FORTRAN structures  
  printing the type, C-6  
  printing the value, C-6  
**\$fp**, J-4  
**fp0** .. **fp7**, J-4  
**\$fpa**, E-1-3  
**fpa0** .. **fpa7**, J-4  
**fpacr**, J-4  
**\$fpa\_reg**, E-1-3  
**fpasr**, J-4  
**fpacr**, J-4  
**fpsr**, J-2, J-4  
**\$fpstat**, J-1  
**fr (floating point registers)**, 4-23,  
  H-5-6  
function  
  calls, 5-12  
  listing, 5-45  
  template instance, calling a, 5-70  
  templates, 5-70  
  templates, listing, 5-72

## G

-g, 3-3  
  **.GE.**, C-2  
**general registers**, 4-25

**g** (**goto**), 3-53, 4-64, 4-66, H-22-24  
 global breakpoint commands  
   **abc**, 4-92, H-37  
   **dbc**, 4-92, H-37  
 global variables, 5-11  
**gr** (**general registers**), 4-23, H-5-6  
**-g**, symbolic debug option, 5-65  
**.GT.**, C-2

## H

handling exceptions, 5-57  
**h** (**help**), 3-54, 4-112, **4-116**, H-45-50  
 hints for using assertions, F-12  
**HISTSIZ**E, 1-10, E-4

## I

**i**, H-38  
**if**, **4-93**  
**i** (**if**), 4-71  
**I** (**Inquire**), H-21  
**iir**, J-3  
 import stub, 4-41  
**\$in**, B-2, C-2, D-2  
**Inquire**, **4-63**  
 installed files, G-1  
 instance breakpoint, 5-4  
 instance breakpoint, setting, 5-54  
 interfaces, separate, 3-47  
 invocation options, debugger, H-1  
**ior**, J-3  
**isr**, J-3  
**itmr**, J-3  
**iva**, J-3

## J

**j**, I-11  
**J**, I-11  
 job control commands  
   **c**, 4-65, H-22-24  
   **C**, 4-65, H-22-24

comparison between **xdb** and **cdb**,  
   I-11  
**g**, H-22-24  
**k**, 4-66, H-22-24  
**r**, 4-64, H-22-24  
**R**, 4-65, H-22-24  
**s**, 4-67, H-22-24  
**S**, 4-68, H-22-24

## K

**k** format specifier, 5-23  
**K** format specifier, 5-23  
**kill**, 3-17, 4-66  
**k** (**kill**), 3-17, 4-64, H-22-24

## L

**l**, I-8-9  
**label**, 4-2-4  
**la** (**list assertions**), 4-97, 4-100,  
   H-40-41  
**\$lang**, E-1-3, I-4  
**LANG**, 3-11, 3-43, E-4  
 language operators  
   C and C++, B-2  
   explanation, D-1  
   Explanation, B-1, C-1  
   FORTRAN 77, C-2  
   Pascal, D-2  
   restrictions for C, B-1  
   restrictions for Pascal, D-1  
**lb** (**list breakpoints**), **3-30**, 4-70,  
   H-25, I-13-14  
**LC\_CTYPE**, 3-43  
**lc** (**list common**), 4-36, H-10-17  
**lcl** (**list classes**), 4-36, 5-72, H-10-17  
**-l** command-line option, shared libraries,  
   6-4  
**lct** (**list class templates**), 4-36,  
   5-72, H-10-17  
**LC\_TYPE**, E-4  
**ld**, 3-4



- ld (list directories), 4-28, H-7-9
  - .LE., C-2
  - lf (list files), 4-28, H-7-9
  - lft (list function templates), 4-36, 5-72, H-10-17
  - lg (list globals), 4-36, H-10-17
  - limitations and hints, F-1
  - \$line, E-1-3
  - line, 4-2-4
  - line mode, Glossary-5
  - LINES, 3-10, E-4
  - linking a program, symbolic debugger information, 3-3
  - list, 4-36
  - list assertions, 4-100
  - list breakpoints, 3-30, 4-77
  - list classes, 4-39, 5-72
  - list class templates, 4-39, 5-72
  - list common, 4-38
  - list directories, 4-33
  - list exceptions, 4-46, 5-63
  - list files, 4-33
  - list function templates, 4-39, 5-72
  - list globals, 4-40
  - listing
    - classes, 5-72
    - class templates, 5-72
    - exceptions, 5-63
    - functions, 5-45
    - function templates, 5-72
    - local variables, 5-44
    - overloaded functions, 5-46
    - template functions, 5-73
    - templates, 5-71
  - list labels, 4-40
  - list macros, 4-41
  - list overload, 4-42
  - list procedures, 4-42
  - list registers, 4-43
  - list shared libraries), 4-45
  - list specials, 4-44
  - list template functions, 4-45, 5-73
  - l (list), 4-36, H-10-17
  - ll (list labels), 4-36, H-10-17
  - L (Location), 4-28, 4-34, H-7-9
  - lm (list macros), 4-36, H-10-17
  - locale, setting up, 3-11
  - local variables, listing, 5-44
  - location, Glossary-5
  - location, 4-2-4
  - location window (line), 3-14
  - lo (list overload), 4-36, H-10-17
  - longjmp(), F-2
  - lowercase, 4-5
  - lp (list procedures), 4-36, H-10-17
  - lpm (list path map), 4-35, 4-35, H-18
  - L prefix (wide-character prefix), 3-46, 4-16
  - lr (list registers), 4-36, H-10-17
  - ls (list specials), 4-36, H-10-17
  - lsl (list shared libraries), 4-36, 6-4, 6-6, H-10-17
  - .LT., C-2
  - ltf (list template functions), 4-36, 5-73, H-10-17
  - lx, 5-63
  - lx (list exceptions), 4-36, H-10-17
  - lz (list signals), H-51
  - lz (list signals), 4-108
- ## M
- M, H-45-50
  - machine instruction, Glossary-5
  - macro, Glossary-5
  - macro facility commands
    - def, 4-106, H-44
    - tm, 4-106, H-44
    - undef, 4-107, H-44
  - macros, 3-52
  - \$malloc, E-1-3
  - Mc, 4-112, 4-121, H-45-50
  - member

functions, 5-2, 5-30  
 pointers, 5-35  
 memory lock, Glossary-5  
 memory locking, 3-13  
 —, B-2, C-2, D-2  
 miscellaneous commands  
   !, 4-112, H-45-50  
   #, 4-113, H-45-50  
   ~, 4-114, H-45-50  
   am, 4-114, H-45-50  
   f, 4-116, H-45-50  
   h, 4-116, H-45-50  
   M, 4-118, H-45-50  
   Mc, 4-121, H-45-50  
   mm, 4-46, H-45-50  
   Mt, 4-122, H-45-50  
   q, 4-117, H-45-50  
   Return, 4-113, H-45-50  
   sm, 4-115, H-45-50  
   ss, 4-117, H-45-50  
   tc, 4-118, H-45-50  
   tM, 4-122, H-45-50  
 M (Map), 4-112, 4-118  
 mm (memory map), 4-36, 4-46, 6-7,  
   H-45-50  
 mod, D-2  
 modifying data, 3-35  
 more, 3-20  
 Mt, 4-112, 4-122, H-45-50  
  
**N**  
 naming restrictions, F-6  
 navigating the execution stack, 3-37  
   .NE., C-2  
   .NEQV., C-3  
 nested classes, 5-4, 5-74  
 next, 3-27, 4-31  
 Next, 3-27, 4-32  
 n (next), 3-27, 4-28, H-7-9  
 N (Next), 3-27, 4-28, H-7-9  
 non-ASCII characters, 3-44

not, D-2  
   .NOT., C-2  
   number, 4-4-5  
   numberp, I-12  
   numbert, I-12  
 numeric constants, 4-18  
  
**O**  
 object  
   identification, 5-4, 5-26, 5-94  
   pointers, 5-33  
   type limitations, F-4  
 object, displaying type information,  
   5-19  
 op= , B-3  
 operand, promotion, 4-19  
 operators, 5-16  
 operators limitations, F-3  
 optimized programs, debugging, 1-7  
 options  
   -d, 3-5, H-1  
   -e, 3-5, 3-8, H-1  
   -l, 3-5, 3-8, H-1  
   -L, 3-5, H-1  
   -o, 3-5, 3-8, H-1  
   -p, 3-5, H-1  
   -P, 3-5, H-1  
   -r, 3-5, H-1  
   -R, 3-5, H-1  
   -s, 3-5, 3-8, H-1  
   -S, 3-5, 3-8, H-1  
 or, D-3  
   .OR., C-2  
 overall breakpoint commands  
   lb, H-25  
   tb, 4-78, H-25  
 overloaded  
   breakpoints, 3-29  
   functions and operators, 5-2  
   functions, listing, 5-46

**P**

p, 3-32, 3-35, I-6-7, I-8-9, I-12  
**\$pagelines**, I-3  
parameterized  
  types, 5-4  
  types, debugging, 5-65  
  types, using, 5-66  
parent process, 3-5  
Pascal, 1-4  
  language operators, D-1  
Pascal\_demo, 2-2  
Pascal main source file, 2-8-9  
path-map, 4-35  
path map (apm), using the, 3-62  
path, prefixing, 3-66  
pausing during execution, 3-28  
**\$pc**, E-1-3, J-1, J-4  
**pc**, J-2, J-4  
**pch**, J-3  
**pct**, J-3  
**.pdbrc**, I-1  
**pid1 .. pid4**, J-3  
playback and record commands, 4-102  
pointer types, 4-20  
**-p** option, 3-41  
**-P** option, 3-55  
position independent code, 3-4  
**p** (**print**), 4-36, 4-113, 4-114, H-10-17  
**pq**, 3-35, I-8-9  
**pq** (**print quiet**), 4-36, **4-55**, H-10-17  
**\$print**, E-1-3  
**print**, 3-32, 3-35, **4-47**  
**priv**, J-2, J-3  
**proc**, 4-2-4  
procedure, Glossary-5  
procedure calls in an expression, entering,  
  4-21  
process, adopting an existing, 3-55  
process limitations, F-2  
promotion of operands, 4-19  
**\$ps**, J-4

**ps**, J-4  
**psw**, J-2, J-3  
**ptrace(2)**, F-2  
**pxdb**, 6-12  
**pxdb++**, 1-4

**Q**

**Q**, H-38  
**Q** (**Quiet**), 3-29, 4-71  
**q** (**quit**), 3-17, 3-29, 4-112, 4-117,  
  H-45-50  
**@-qualified**, 6-8  
**Quiet**, **4-94**  
**quit**, **4-117**

**R**

**+r**, 4-23, 4-113, 4-114, H-5-6  
**-r**, 4-23, 4-113, 4-114, H-5-6  
**\$r0 .. \$r31**, J-1  
**r0 .. r31**, J-2  
**\$r7**, E-1-3  
**rctr**, J-3  
record and playback commands  
  **>**, 4-102, H-42-43  
  **>@**, 4-104, H-42-43  
  **>@c**, 4-104, H-42-43  
  **>c**, 4-104, H-42-43  
  **>@f**, 4-104, H-42-43  
  **>f**, 4-104, H-42-43  
  **<<file**, 4-102, H-42-43  
  **<file**, 4-102, H-42-43  
  **>>@file**, 4-104, H-42-43  
  **>>file**, 4-102, H-42-43  
  **>@file**, 4-104, H-42-43  
  **>file**, 4-102, H-42-43  
  **>@t**, 4-104, H-42-43  
  **>t**, 4-104, H-42-43  
  **tr**, 4-102, H-42-43  
  **tr @**, 4-104, H-42-43  
recording, toggle, I-16  
reference types, 5-12

- registers, Glossary-6
    - displayed by the symbolic debugger, J-1
    - displayed in floating-point register windows, J-2
    - displayed in general register windows, J-2
    - displayed in the special register window, J-3
    - Series 300/400 computers, J-4
  - Release
    - 7.0, 1-5
    - 7.40, 1-4
    - 8.0, 1-5
    - 8.00, 1-4
  - rerunning a debugger session, 3-41
  - restoring and saving the debugger state, 3-42
  - \$result**, E-1-3
  - resuming execution after a breakpoint, 3-30
  - \$ret0 .. \$ret1**, J-2
  - (Return)**, 4-112, 4-113, H-45-50
  - RM, J-2
  - r option, 3-41
  - R option, 3-42
  - r (run)**, 3-16, 4-64, H-22-24
  - R (Run)**, 3-16, 4-64, 4-65, H-22-24
  - running process, adopting a, 3-5
  - running the HP symbolic debugger, 3-5
  - run-time stack, 3-18, 3-36
- S**
- s, I-11
  - S, I-11
  - sa, 4-100
  - sample
    - debugger session, 2-1
    - program listings, 2-6
    - session for scenario two, 2-4
    - sessions for scenario one, 2-2
  - sample program
    - C, 2-10
    - C++, 2-12
    - FORTRAN 77, 2-7
    - Pascal, 2-8
  - sar**, J-2, J-3
  - sa (suspend assertion)**, 4-97, H-40-41, I-15
  - save-state, I-16
  - save state**, 4-117
  - save state limitations, F-11
  - saving and restoring the debugger state, 3-42
  - sb (suspend breakpoint)**, 4-70, H-32-33
  - s command-line option, shared libraries, 6-4
  - scope, 5-5
  - screen, setting up, 3-10
  - searching a program, 3-27
  - separate interfaces, 3-47
  - setjmp(3C)**, F-2
  - setting breakpoints, 3-28
  - setuid(2)**, F-2
  - set-user-ID**, F-2
  - shared libraries, Glossary-6
    - abbreviated library names with -l, 6-3
    - adoption, 6-12
    - basename limitations, 6-2
    - core files, 6-14
    - creating, 3-4
    - creating the library, 6-2
    - debugging in an adopted process, 6-12
    - debugging program that use, 6-1
    - disassembly mode, 6-10
    - enable debugging, 6-2
    - explicit loading (**shl\_load(3x)**), 6-4
    - explicit references to, 6-9
    - force loading of symbols in, 6-9

- how symbols are found by the debugger, 6-6
- implicit loading (*ld(1)* -l), 6-4
- invoking the debugger, 6-4
- l command-line option, 6-4
- ld(1)* +b option, 6-3
- ld(1)* +s option, 6-3, 6-15
- /lib/crt0.o, 6-14
- /lib/frt0.o, 6-14
- loading with BIND\_FIRST, 6-15
- lsl (list shared libraries), 6-4, 6-6
- mm (memory map) command, 6-7
- modification while debugging, 6-15
- @ operator, 6-8
- overriding normal symbol binding rules, 6-8
- prepare for debugging, 3-4
- s command-line option, 6-4
- searching for, 6-3
- SHLIB\_PATH environment variable, 6-3, 6-15
- special considerations, 6-14
- special symbol DMEM, 6-10
- special symbol TMEM, 6-10
- summary of extended commands, 6-11
- swap requirements while debugging, 6-14
- symbol binding, 6-6
- symbols in, 6-8
- /usr/bin/pxdb, 6-12
- /usr/lib/endl.o, 6-2, 6-13, 6-14
- versioning, 6-14
- +z compiler options (PIC) with -g, 6-2
- +Z compiler options (PIC) with -g, 6-2
- shared library, 4-2-4, 4-4-5, Glossary-6
- shared library limitations, F-8
- shl\_definesym(3X)*, 6-10
- shl\_load(3X)*, 6-6, 6-7
- shorthand notation for size, 4-53
- SIGINT signal, F-3
- \$signal, E-1-3
- signal command, I-15
- signal commands
  - lz, H-51
  - z, 4-110, H-51
- signals restrictions, F-3
- SIGTRAP signal, F-3
- sigwinch, 3-10
- single-stepping commands, I-5
- size, changing the source window, 3-21
- \$sizeof, B-2, C-2, D-2
- sizeof, B-2, D-2
- sm, 3-20
- sm (suspend more), 4-112, 4-115, H-45-50
- source, Glossary-6
  - code display and assembly code display, 3-24
  - mode, 3-13, 3-24
  - window, 3-14, 3-18
- source code, displaying, 3-24
- source directory mapping, 3-62
- source directory mapping commands
  - apm, H-18
  - dpm, H-18
  - lpm, H-18
- source file mapping commands
  - apm, 4-35
  - dpm, 4-35
  - lpm, 4-35
- source limitations, F-1
- source line, Glossary-6
- source window
  - size, 3-21
- \$sp, E-1-3, J-2, J-4
- special variables
  - \$cplusplus, 4-12
  - \$depth, 4-10

- description, 4-9, Glossary-6
  - `$fpa`, 4-14
  - `$fpa_reg`, 4-14
  - `$lang`, 4-11
  - `$line`, 4-11
  - `$malloc`, 4-11
  - names used for registers, J-1
  - `$print`, 4-11
  - `$result`, 4-10
  - `$signal`, 4-12
  - `$step`, 4-14
  - table of, E-1
  - `$var`, 4-10
  - xdb and cdb comparison, I-3
  - split-screen mode, I-5
  - `sr0 .. sr7`, J-3
  - `sr` (**special registers**), 4-23, H-5-6
  - `ss` (**save state**), 3-42, 4-112, 4-117, H-45-50, I-16
  - `s` (**step**), 3-16, 3-26, 4-64, 4-113, 4-114, H-22-24, I-5
  - `S` (**Step**), 3-16, 3-26, 4-64, 4-113, 4-114, H-22-24, I-5
  - stack, Glossary-7
    - run-time, 3-18, 3-36
  - stack viewing commands, I-10
    - `down`, 4-60, H-19-20
    - `t`, 4-57, H-19-20
    - `T`, 4-59
    - `top`, 4-61, H-19-20
    - `tst`, 4-61, H-19-20
    - `up`, 4-60, H-19-20
    - `V`, 4-60, H-19-20
  - startup command file, I-1
  - static data members, displaying, 5-42
  - static members of an enclosed class, 5-74
  - status viewing commands
    - `I`, 4-63, H-21
  - `$step`, E-1-3, I-3
  - `Step`, 4-68
  - `Step-into` (`s`), 5-64
  - `Step-over` (`S`), 5-64
  - stepping through a program, 3-26
  - `step` (`step`), 4-67
  - stopping execution (temporarily), 3-28
  - string, Glossary-7
  - `stub`, 4-13, 4-58, 4-62, Glossary-7
  - `stub`, export, 4-41, 4-61
  - `stub`, import, 4-41
  - `suspend assertion`, 4-100
  - `suspend breakpoint`, 4-87
  - `suspend more`, 3-20, 4-115
  - symbolic constants, 4-17
  - symbolic debugger, I-1
    - commands summary, H-1
    - terminal support, I-8
    - use of, 3-1
    - user requirements, I-4
  - symbolic debug option (`-g`), 5-65
- ## T
- `>@t`, 4-104, H-42-43
  - `>t`, 4-104, H-42-43
  - `t`, 3-36, I-10
  - `T`, 3-36, 4-59, I-10
  - `ta` (**toggle assertions**), 4-97, 4-101, H-40-41, I-15
  - `tb` (**toggle breakpoints**), 4-70, H-25
  - `tc` (**toggle case**), 3-9, 3-27, 4-6, 4-112, 4-118, 4-118, H-45-50, I-16
  - `td` (**toggle disassembly**), 3-22, 4-23, 4-23, H-5-6
  - `template`, 5-66
  - template
    - data, displaying, 5-70
    - function, 5-70
    - functions, listing, 5-73
    - listing, 5-71
    - type of an object, 5-71
  - template class
    - all member functions, 5-68

data member values, 5-70  
 type of an object declared as a, 5-71  
**template**, setting breakpoints, 5-67  
**TERM**, 3-10, E-4  
 terminals  
   supported, 1-8  
   that do not support window mode,  
     1-8  
   that do support window mode, 1-8  
     without memory locking, 3-13  
**terminfo**, 3-10  
**tf (toggle float)**, 4-23, H-5-6  
**throw**, 5-57, 5-57  
 throw, stopping on, 5-59  
**tM**, H-45-50  
**TMEM**, 6-8, 6-10  
**tm (toggle macros)**, 4-105, 4-106, H-44  
**tM (toggle maps)**, 4-122  
**tM(toggle maps)**, 4-112  
**toggle assertions**, 4-101  
**toggle breakpoints**, 4-78  
**toggle case**, 3-9, 3-27, 4-118  
 toggle case sensitivity, I-16  
**toggle case (tc)**, 4-6, 4-118  
**toggle disassembly**, 3-22  
**toggle exception catch**, 5-61  
**toggle exception throw**, 5-59  
**toggle float**, 4-24  
**toggle macros**, 4-106  
 toggle recording, I-16  
**toggle screen**, 3-24  
 toggle screen, 4-26  
 toggling the case sensitivity, 3-9  
**top**, 3-39, 4-56, 4-61, H-19-20  
**tr**, 4-102, H-42-43, I-16  
**tr @**, 4-104, H-42-43  
**tr0 .. tr7**, J-3  
**trace**, 3-36, 4-57  
 trace breakpoint, 3-29  
 trace program flow, 1-2

tracing function and procedure calls,  
   3-36  
 transparent name demangling, 5-2  
**try**, 5-57  
**ts (toggle screen)**, 3-24, 4-23, H-5-6  
**tst (toggle stubs)**, 4-56, 4-61, H-19-20  
**t (trace)**, 4-56, H-19-20  
**T (Trace)**, 4-56, H-19-20  
**txc (toggle exception catch)**, 4-95,  
   4-96, 5-61, H-39  
**txt (toggle exception throw)**, 4-95,  
   5-59, H-39

## U

**undef**, 4-105, 4-107, H-44  
**up**, 3-39, 4-56, 4-60, H-19-20  
**update**, 4-26  
**Update**, 4-26  
 uppercase, 4-5  
 user errors (UE42 - UE2031), A-3  
 /usr/bin/pxdb, shared libraries, 6-12  
 /usr/lib/xdb\_demos, 2-2  
**u (update)**, 4-23, H-5-6  
**U (Update)**, 4-23, H-5-6

## V

**v**, I-6-7  
**V**, I-6-7  
**\$var**, E-1-3  
**var**, 4-4-5  
 variable, 5-10  
   name conventions, I-2  
   names, entering, 4-6  
   special, 4-9  
   specifying, 4-6  
**va (view address)**, 4-28, H-7-9, I-6-7  
**view**, 3-18, 4-30  
**View**, 3-18, 4-60  
**view address**, 4-34  
 viewing functions with the debugger,  
   5-47

view machine instructions, 1-2  
 view source code, 1-2  
 virtual table pointer, 5-91  
**VISUAL**, 1-10  
 VMS FORTRAN record support, C-1,  
**C-4**

VMS FORTRAN record types

Maps, C-4  
 Records, C-4  
 Structures, C-4  
 Unions, C-4  
**v (view)**, 3-18, 4-28, 4-113, 4-114, H-7-9  
**V (View)**, 3-18, 3-40, 4-56, H-19-20

## W

**+w**, I-6-7  
**+W**, I-6-7  
**-w**, I-6-7  
**-W**, I-6-7  
 wide characters, 3-45  
**window**, 3-21, **4-27**  
 window, Glossary-7  
 window command  
**fr**, 4-24  
**td**, 4-23  
 window mode, Glossary-7  
 window mode commands  
**gr**, 4-25, H-5-6  
**+r**, 4-25, H-5-6  
**-r**, 4-25, H-5-6  
**sr**, 4-25, H-5-6  
**tf**, 4-24, H-5-6  
**ts**, 4-26, H-5-6  
**u**, 4-26, H-5-6  
**U**, 4-26, H-5-6  
**w**, 4-27, H-5-6  
 window mode requirements, F-13  
 windows

command, 3-14, 3-20  
 location, 3-14  
 source, 3-13, 3-14, 3-18, 3-21  
**ws**, I-6-7  
**w (window)**, 3-21, 4-23, H-5-6, I-6-7

## X

**x**, 4-101  
**xcc (exception catch command)**, 4-95,  
**4-96**, 5-62, H-39  
**xdb**, 1-1  
 command, 3-5, 3-41  
**xdb** and **cdb** comparison, I-1  
**xdb**, basic command form, I-2  
**xdb.cat**, 3-11  
**XDBEDIT**, 1-10, E-4, I-4  
**xdb.help**, G-2  
**XDBHIST**, 1-10, E-4  
**xdb** options, H-2  
 directory option, 3-5  
 line mode option, 3-5  
 object file, 3-5  
 playback file, 3-5  
 record file, 3-5  
 string cache size option, 3-5  
 version number option, 3-5  
**.xdbrc**, 3-9, I-1  
**x (exit)**, 4-97, H-40-41, I-15  
**xtc (exception throw command)**, 4-95,  
**4-96**, 5-60, H-39  
 X windows, 3-48

## Z

**+z**, 3-4  
**z**, I-15  
**Z**, I-16  
**signal**, **4-110**, H-51







Reorder No. or  
Manual Part No.  
B2355-90044

Copyright © 1992  
Hewlett-Packard Company  
Printed in USA E0892

**Manufacturing  
Part No.  
B2355-90044**



B2355-90044