# Digital UNIX

# Ladebug Debugger Manual

Order Number: AA–PZ7EE–TE

**March 1996**

This manual describes how to use the Ladebug debugger (both the graphical user interface and the command-line interface) to debug applications written in the programming languages Ada, C, C++, COBOL, Fortran 77, and Fortran 90, on the Digital UNIX operating system.

The Ladebug debugger was formerly called DECladebug.

**Revision/Update Information:** This is a revised manual.

**Product Version:** Digital UNIX Version 4.0 and higher
Ladebug Version 4.0

**Digital Equipment Corporation**
**Maynard, Massachusetts**

**March 1996**

ZK6450

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

# Contents

## Examples

## Tables

# Preface

This manual contains information for debugging programs with the Digital Ladebug debugger. Ladebug is a debugger on the Digital UNIX® operating system. Digital has changed the name of its UNIX operating system from DEC OSF/1 to Digital UNIX. The new name reflects Digital's commitment to UNIX and its conformance to UNIX standards.

The terms *graphical user interface* and *window interface*, as used in this manual, are synonymous. They refer to the window environment available in Digital UNIX.

## Intended Audience

This manual is intended for programmers with a basic understanding of one of the programming languages that Ladebug supports (C, C++, Ada, COBOL, Fortran, and machine code), and the Digital UNIX operating system.

## Structure of this Document

This manual is organized as follows:

– Chapter 1 briefly introduces debugging concepts and Ladebug features, and tells how to prepare for debugging.

- Part I describes the debugger's window interface. Part I includes the following chapters:
  - – Chapter 2 introduces the debugger's window interface.
  - – Chapter 3 explains how to start and end a debugging session.
  - – Chapter 4, which is organized by task, explains how to use the debugger.
  - – Chapter 5 discusses how (from the window interface) to debug programs containing multiple processes and threads.

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

- Chapter 6 describes how to use the Ladebug debugger from within the DEC FUSE environment.

- Part II describes the debugger's command interface. Part II includes the following chapters:
  - Chapter 7 describes basic debugging techniques with the command interface. This chapter and Chapter 9 also have information on debugging C programs.
  - Chapter 8 explains how to examine information in your programs.
  - Chapter 9 describes how to control program execution.
- Part III presents language-specific topics and includes the following chapters:
  - Chapter 10 describes the debugger support for C++.
  - Chapter 11 describes the debugger support for Fortran.
  - Chapter 12 describes the debugger support for Ada.
  - Chapter 13 describes the debugger support for COBOL.
- Part IV describes advanced topics and includes the following chapters:
  - Chapter 14 describes some techniques for debugging core files.
  - Chapter 15 explains how to use debugger scripts.
  - Chapter 16 explains how to debug programs with shared libraries.
  - Chapter 17 explains how to debug programs with limited symbolic information.
  - Chapter 18 describes machine-level debugging.
  - Chapter 19 explains how to debug multithreaded applications.
  - Chapter 20 explains how to debug multiprocess applications.
  - Chapter 21 describes client/server remote debugging.
  - Chapter 22 describes kernel debugging.
- Part V, the Command Reference, presents the debugger commands alphabetically and fully describes them, along with the commands and options for invoking the debugger. Part V is similar to the `ladebug(1)` reference page.
- The appendixes are as follows:
  - Appendix A describes debugging within the `emacs` editing environment.

– Appendix B explains how to write a remote debugger server.

– Appendix C describes Ladebug's support for international users.

## Programming Languages

This manual emphasizes debugger usage that is common to all or most supported languages. For more information specific to a particular language, see:

- The debugger's online help system
- The documentation supplied with that language, particularly regarding compiling and linking the program for debugging

## Related Documents

The following documents contain related information:

- *Digital UNIX Programmer's Guide*
- *Digital UNIX Motif User's Guide*
- *CDE User's Guide*
- *Developing Ada Programs on Digital UNIX Systems*
- *The Annotated C++ Reference Manual* (Ellis and Stroustrup, 1990, Addison-Wesley)
- *DEC COBOL User Manual*
- *DEC Fortran 90 User Manual for DEC OSF/1 Systems*
- Release Notes for Ladebug Version 4.0
- *DEC Fuse Handbook*
- Reference pages:

  `ladebug(1)` , similar to Part V in this manual
  `cxx(1)` for C++
  `cc(1)` for C
  `ada(1)` for Ada
  `cobol(1)` for DEC COBOL
  `f77(1)` for DEC Fortran 77
  `f90(1)` for DEC Fortran 90

The printed version of the Digital UNIX documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Digital.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

| Audience | Icon | Color Code |
|---|---|---|
| General users | G | Blue |
| System and network administrators | S | Red |
| Programmers | P | Purple |
| Device driver writers | D | Orange |
| Reference page users | R | Green |

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the Digital UNIX documentation set.

## Reader's Comments

Digital welcomes any comments and suggestions you have on this and other Digital UNIX manuals. You can send your comments in the following ways:

- Fax: 603-881-0120 Attn: UEG Publications, ZK03-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on line in the following location:

  `/usr/doc/readers_comment.txt`
- Mail:  
  Digital Equipment Corporation  
  UEG Publications Manager  
  ZK03-3/Y32  
  110 Spit Brook Road  
  Nashua, NH 03062-2698

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)

- The section numbers and page numbers of the information on which you are commenting.
- The version of Digital UNIX that you are using, for example, Digital UNIX Version 4.0.
- If known, the type of processor that is running the Digital UNIX software, for example, AlphaServer 2000.

The Digital UNIX Publications group cannot respond to system problems or technical support inquiries.

## Reporting Software Problems

Information provided with the software media explains how to send problem reports to Digital. If you need to report a software problem with the Ladebug debugger, contact your local Digital Customer Support Center. Please include as much information as possible to help diagnosis and resolution of the problem. Useful information includes the following:

- The Ladebug version number as reported in the "Welcome" banner
- The source code of a sample program or an executable binary
- The system configuration
- The exact steps used to reproduce the problem
- The `.i` or `.ixx` file (if the executable program is compiled with a compiler that supports the "-P" option)
- The core file if available

## New and Changed Features

This manual has been revised to document all of the Ladebug changes that are part of the current release (Version 4.0). They are described in the following sections.

### Multiprocess Application Debugging

You can now have multiple processes (zero or more processes) under debugger control. You can keep track of the processes using the `show process` command and switch between them using the `process` command, and debug nonrelated processes simultaneously.

## Multithreaded Debugging

Ladebug supports the debugging of DECthreads and native threads (for example, Digital UNIX kernel (machine) level threads). You specify whether you are working with DECthreads or native threads with the debugger command `set $threadlevel`, as follows:

```
(ladebug) set $threadlevel="decthreads"
(ladebug) set $threadlevel="native"
```

The `thread` command lets you identify or set the current thread context. The `show thread` command lists all threads known to the debugger.

Other commands with enhanced syntax for multithread debugging include:

- `stop`
- `trace`
- `when`
- `step` (and `stepi`)
- `next` (and `nexti`)
- `cont`

See the information on each of these commands. See also information about the `where`, `show condition`, and `show mutex` commands.

## Load/Unload a Process

The `load` command lets you load an image file or core file for debugging. The `unload` command removes the symbol table information that the debugger had associated with the process being debugged.

## Kernel Debugging

Ladebug supports kernel debugging. The functionality is equivalent to kernel debugging using dbx.

When you have a problem with a process, you can debug the running kernel or examine the values assigned to system parameters. (It is generally recommended that you avoid modifying the value of the parameters, which can cause problems with the kernel.) Kernel debugging requires superuser privilege.

To debug a kernel locally, invoke the debugger with the following command:

```
#  ladebug -k
```

The `-k` flag maps virtual to physical addresses to enable local kernel debugging. The `/vmunix` and `/dev/mem` parameters cause the debugger to operate on the running kernel. Use Ladebug commands to display the current process identification numbers (pid) and trace the execution of processes.

To debug a remote kernel, invoke the debugger with the following command:

```
#  ladebug -remote /testdir/vmunix
```

Refer to Chapter 22 for more information about kernel debugging.

### Multilanguage Support

Ladebug Version 4.0 enhances language support as follows:

- C/C++ — Ladebug fully supports debugging ACC and DEC C programs, and DEC C++ programs; DEC C++ now supports cfront compatibility and the GEM compiler backend. Ladebug's support for C++ includes C++ names and expressions, including template instantiations and exception handling; and setting breakpoints in member functions, overloaded functions, constructors and destructors, template instantiations, and exception handlers. You can change the current class scope to set breakpoints and examine members of a class that are not currently in scope. You can debug mixed-language programs.
- Fortran — Ladebug lets you debug DEC Fortran 77 and DEC Fortran 90 programs. You can specify identifiers, program names, subroutine names, and array sections to Ladebug with Fortran language syntax, including case insensitivity. You can display values of variables in a Fortran common block; access Fortran derived-type, record, array, and complex variables; examine Fortran 77 and Fortran 90 data types (with some limitations); and debug mixed-language programs.

### Window Interface

Ladebug's window interface supports the major Ladebug command-line functionality. Other features can be accessed within the window interface from a command window. The window interface includes a main window covering the basic debugging and convenience features; optional views windows, various pop-up menus and dialog boxes, and a command-entry prompt.

Ladebug's window interface is documented in this revised manual. Earlier versions of this manual documented only the command interface.

### Support for International Users

User programs may set different locales in order to interpret text according to different language/culture-related criteria. In addition, locales may be switched inside a user program.

Ladebug can follow the debugged program so that its interpretation of program data is identical to that of the debugged user program.

Your Digital UNIX system needs to have the worldwide (WW) subsets installed. This is standard procedure and the WW subsets are available on the base system CD-ROM. You can then set your locale to a special locale (such as a Japanese locale) and input characters which are multibytes. These characters can come from a script or entered from a VT terminal using compose sequences.

Wide characters (type wchar_t) and wide strings (type wchar_t *) are used in international applications as run-time representation of multibyte character data. Ladebug supports input of multibyte characters that are regarded as components of symbol literals, string literals or wide character literals. Ladebug's basic support for wide characters (wchars) and wide strings (wstrings) is as follows:

- Accept input of wide character literals and wide string literals (for C and C++ programs).
- Print out wide character/string data as the real characters they represent.

### Environment-Manipulation Commands

Ladebug provides commands for manipulating the environment of subsequent debuggees with environment variables. With the `setenv`, `export`, `printenv`, and `unsetenv` commands, used within the debugger, you can set the value of an environment variable, display the values of environment variables, and remove environment variables. See Chapter 9 for more information.

### The pop Command

The `pop` command removes execution frames from the call stack. It is useful when execution has passed an error that needs to be corrected. See Chapter 9 for more information.

## Conventions

Table 1 lists the conventions used in this manual.

**Table 1 Conventions Used in This Manual**

| Convention | Meaning |
|---|---|
| # | A pound sign (#) is the default superuser prompt. |
| $\boxed{\text{Return}}$ | In examples, a boxed symbol indicates that you must press the named key on the keyboard. |

(continued on next page)

**Table 1 (Cont.) Conventions Used in This Manual**

| Convention | Meaning |
|---|---|
| Ctrl/C | This symbol indicates that you must press the Ctrl key while you simultaneously press another key (in this case, C). |
| **`user input`** | In interactive examples, this typeface indicates input entered by the user. |
| `filesystem` | In text, this typeface indicates the exact name of a command, routine, partition, pathname, directory, or file. This typeface is also used in interactive examples and other screen displays. |
| UPPERCASE<br>lowercase | The Digital UNIX operating system differentiates between lowercase and uppercase characters. On the operating system level, examples, syntax descriptions, function definitions, and literal strings that appear in text must be typed exactly as shown. |
| `setld(8)` | Cross-references to online reference pages include the appropriate section number in parentheses. For example, `setld(8)` indicates that you can find the material on the `setld` command in Section 8 of the reference pages. |
| `[y]` | In a prompt, square brackets indicate that the enclosed item is the default response. For example, `[y]` means the default response is Yes. |
| *file* | Italic type indicates variable values, placeholders, function argument names, and names in examples. |
| Actions:Create Group... | Indicates an item on a menu. In this example, you would choose the Create Group... item on the Actions menu. |
| [ \| ]<br>{ \| } | In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed. |
| . . . | In syntax definitions, an ellipsis indicates that the preceding item can be repeated one or more times. |
| Meta+x | This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the plus sign. |

# 1 Introduction to Debugging

This chapter introduces some fundamental debugging concepts, briefly describes key Ladebug features, explains how to compile and link a program for debugging, and introduces the two Ladebug interfaces (window and command).

## 1.1 Overview of Debugging Concepts

The debugger helps you locate run-time programming or logic errors, also known as bugs. You use the debugger with a program that has been compiled and linked successfully, but does not run correctly. For example, the program might give incorrect output, go into an infinite loop, or terminate prematurely.

You locate errors with the debugger by observing and manipulating your program interactively as it executes. The debugger lets you:

- Display your program's source code and browse through the code to locate points of interest where you might test for certain conditions
- Set breakpoints to suspend program execution or enter command sequences at such points and set tracepoints that notify you of certain conditions
- Execute your program, including stepping through one source line or machine instruction at a time and restarting from the beginning of the program
- Display, monitor, and change the values of program variables and data structures and examine their types
- Examine and manipulate the currently active routines on the call stack
- Examine and manipulate processes and threads in the program
- Disassemble and examine machine code; examine and modify machine-register values
- Customize the debugging environment
- Intercept software signals sent to the program by the operating system

As you use the debugger and its documentation, you will discover variations on the basic techniques. You can also customize the debugger to meet your own needs.

Because the debugger is a symbolic debugger, you can specify variable names, routine names, and so on, precisely as they appear in your source code. You do not need to specify memory addresses or registers when referring to program locations, but you can if you want.

You can use the debugger with programs written in any of the supported languages:

C
C++
Fortran 77
Fortran 90
Ada
COBOL
Machine code

See your compiler documentation for information about the current extent of support for your language.

## 1.2 Key Features of the Ladebug Debugger

Key features of the debugger allow you to:

- Perform source-level debugging
- Debug programs written in the supported languages
- Attach to a running process
- Debug programs with shared libraries
- Debug multithreaded applications
- Debug more than one program at a time (multiprocess applications)
- Perform remote client/server debugging
- Perform kernel debugging
- Use local language characters (for international users)

You can customize the debugger environment or a debugger session by using:

- Scripts containing debugger commands
- Aliases for frequently used debugger command sequences

- Variables to store information

## 1.3 Basic Debugging Technique

Programmers use debuggers most often to extract important pieces of information during program execution. A simple debugging procedure might include these steps:

1. Compile and link the program. (See Section 1.4.) Your program must compile and link without any errors; the debugger only operates on programs that successfully compile into executable files.
2. Invoke the debugger on your program.
3. Set a breakpoint at the line number or function suspected to be at fault.
4. Run the program from the debugger prompt.
5. Examine the values of program variables.

   When program execution suspends at the specified breakpoint, examine the values of program variables. If the variables do not provide you with any clues, try setting other breakpoints, executing program statements line by line, or tracing a variable's value during program execution.

## 1.4 Preparation for Debugging: Compiling and Linking a Program

Direct the compiler (normally with the `-g` flag; use the appropriate flag for your compiler) to produce an executable file that includes full symbolic debugging information.

Some systems provide variants of the flag (`-g1`, `-g2`, and so on). These give different levels of symbol information and optimization.

Some compilers optimize the object code to reduce the size of the program or to make it run faster. For example, some optimization techniques eliminate certain variables. In addition to building symbolic information, the `-g` flag disables optimization of your program so that you an debug it more easily. For detailed information on compiling and linking, see the chapter on your language or your compiler documentation.

Ladebug can debug programs with less than complete symbolic information. (See Chapter 17 for details.)

The following example shows how to compile and link a C program named `eightqueens` before using the debugger. The compiler is invoked from the C shell (%). In the following example, the file `eightqueens.c` contains the program's source code.

```
% cc -g eightqueens.c -o eightqueens
```

The `cc` command invokes both the compiler and the linker. (For more information about compiling that is specific to a particular language, see the documentation furnished with that language.)

The `-g` flag directs the compiler to write the symbolic information associated with `eightqueens.c` into the program file `eightqueens` (in addition to the code and data for the program). This symbol information allows you to use the names of variables and other symbols declared in `eightqueens.c` when using the debugger. If your program's source code is in several files, you must compile each file whose symbols you want to reference with the `-g` flag.

## 1.5 Overview of the Two Debugger Interfaces

The debugger has the following user interface options to accommodate different needs and debugging styles:

- The debugger has a window interface for workstations. This window interface can be accessed from CDE (Common Desktop Environment) or from Motif. See the *CDE User's Guide* for more information about CDE. When using this interface, you interact with the debugger by using a mouse and pointer to choose items from menus, click on buttons, select names in windows, and so on. The window interface provides the debugging and convenience features that you will need most of the time.

  The window interface has a command-entry prompt (in the Command Message View of the main window) that enables you to enter debugger commands for the following purposes:

  - As an alternative to using the window interface for certain operations
  - To do debugging tasks not available through the window interface

  You can customize the window interface with many of the special features of the debugger by modifying the push buttons and their associated debugger commands or by adding new push buttons.

  Choose Help:On Commands for online help on debugger commands or type `help` at the command line.

- The debugger also has a command interface. With the command interface, you accomplish your debugging tasks by typing commands at the debugger prompt.

# Part I

## Graphical User Interface

This part describes the debugger's window interface.

# 2 Introduction to the Ladebug Debugger: Graphical User Interface

This chapter introduces the Ladebug debugger graphical user interface and provides the following information:

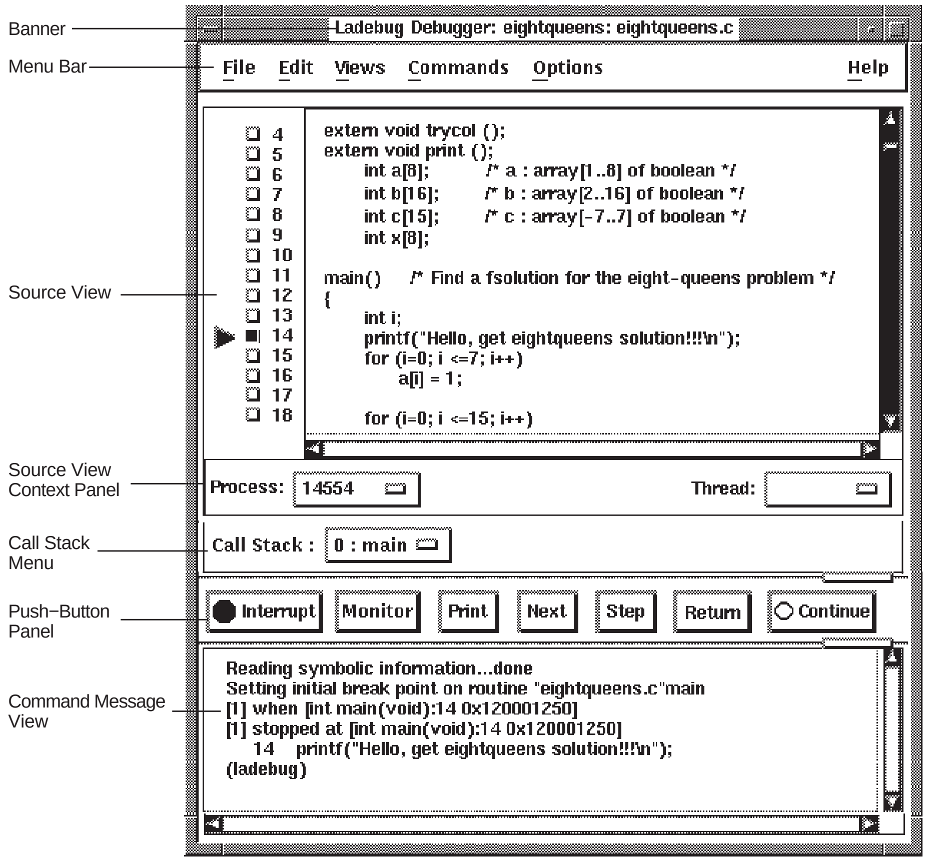
- An orientation to the debugger's screen features, such as windows, menus, and so on (Section 2.2)
- Instructions for using context-sensitive pop-up menus (Section 2.3)
- Instructions for entering debugger commands at the command-entry prompt (Section 2.4)
- Instructions for accessing online help (Section 2.5)

For information about starting a debugger session, see Chapter 3. For information about using the debugger, see Chapter 4. For information about advanced debugging techniques, see Chapter 5.

## 2.1 Convenience Features

This section highlights some of the convenience features of the debugger's window interface. Figure 2–1 gives visual details of the Main Window.

**Figure 2–1 Default Main Window Configuration**


Banner
Menu Bar
Source View
Source View Context Panel
Call Stack Menu
Push−Button Panel
Command Message View
Ladebug Debugger: eightqueens: eightqueens.c
File Edit Views Commands Options Help
Process: 14554
Thread:
Call Stack : 0 : main
Interrupt Monitor Print Next Step Return Continue

```
extern void trycol ();
extern void print ();
      int a[8];          /* a : array[1..8] of boolean */
      int b[16];         /* b : array[2..16] of boolean */
      int c[15];         /* c : array[-7..7] of boolean */
      int x[8];

main()    /* Find a fsolution for the eight-queens problem */
{
      int i;
      printf("Hello, get eightqueens solution!!!\n");
      for (i=0; i <=7; i++)
           a[i] = 1;

      for (i=0; i <=15; i++)
```

```
Reading symbolic information...done
Setting initial break point on routine "eightqueens.c"main
[1] when [int main(void):14 0x120001250]
[1] stopped at [int main(void):14 0x120001250]
    14    printf("Hello, get eightqueens solution!!!\n");
(ladebug)
```

## Source View

The Source View is used for examining the source code of an object. The source-code display in the Source View is automatically updated to show where program execution is currently paused. The Source View has a line number area for setting breakpoints. You can enable and disable the display of line numbers.

A source browser feature lists the images, modules, and routines of your program and lets you display source code in arbitrary modules or routines and set breakpoints on routines. By double clicking on image and module names, you can list the underlying hierarchy of modules and routines. See Section 4.1.1 for more information about the source browser.

### Source View Context Panel

The Source View Context Panel has pull-down menu buttons that allow the user to select a process or thread to be displayed in the Source View or to replace the currently displayed process or thread with a different process or thread.

The Source View Context Panel has three option menu buttons:

- The Process menu lists the available processes (processes the debugger knows about)
- The Thread menu lists the threads within the current process. If the current process has only one thread, the thread menu button is blank.
- The Call Stack menu lists the sequence of routine calls currently on the Call Stack.

See Chapter 5 for more information about using the current process and thread.

### Command Message View

The Command Message View, located directly under the push-button panel in the Main Window, displays user commands and debugger output.

### Breakpoints

A breakpoint is a location in your program at which you want execution to stop so that you can perform actions such as checking the current value of a variable or stepping into a routine. You set, deactivate, and activate breakpoints by clicking on buttons next to the source lines in the main window or the instruction window.

Optionally you can set, deactivate, or activate breakpoints by selecting items in window pull-down menus, context-sensitive menus, or dialog boxes. You can set conditional breakpoints or action breakpoints. See Section 4.4 for more information about breakpoints.

### Push-Button Panel

Push buttons in the push-button panel control common operations: by clicking on a button, you can start execution, step to the next source line, display the value of a variable selected in a window, interrupt execution, and so on.

You can modify, add, remove, and resequence buttons and the associated debugger commands. See Section 4.9.3 for more information about customizing the push-button panel.

### Context-Sensitive Pop-Up Menus

Context-sensitive pop-up menus in debugger windows and views list common operations associated with your location: when you click on MB3, the menu lists actions for the text you have selected, the source line at which you are pointing, or the window pane in which you are working. For more information about using context-sensitive pop-up menus (see Section 2.3).

### Displaying and Manipulating Data

To display the value of a variable or expression, you select text from the main window and click on the Print button. If you choose, you can display values in different type or radix formats.

### Run/Rerun Program

You can rerun the same program or run another program from the same debugging session without exiting the debugger. When rerunning a program, you can choose to save the current state (activated or deactivated) of breakpoints.

### Optional Views

Optional Views include several context-related views. The Optional View Window also contains process and thread visual cues (indicators), which show the currently active process or thread (see Chapter 5).

Optional Views include:

- Breakpoint
- Instruction
- Register
- Monitor
- Local Variables

These views are described in more detail in Section 2.2.2.

### Debugger I/O Window

The Debugger I/O Window displays user input and program output for interactive programs. This isolates program I/O from debugger I/O, which appears in the Command Message View.

A Debugger I/O Window displays by default if you invoke the debugger in any of the following ways:

- If you invoke the debugger within DEC FUSE.
- If you invoke the debugger from the command line using the `-iow` option.

\

If you invoke the debugger within CDE, by default a terminal window opens to display I/O.

### Integration with Command Interface

The debugger's window interface is layered on, and closely integrated with, the command-driven debugger as follows:

- When you use the window interface, the resulting commands are echoed in the Command Message View so that you can correlate input to and output from the debugger.
- When you enter commands at the prompt, they update the window views accordingly.

### Integration with the Source-Level Editor

After you locate an error in your source code, you do not need to exit the debugger to edit your source code. You can display the source code in an editor window, search and replace text, or add additional text.

### Customization

You can modify the following and other aspects of the debugger's window interface and save the current settings in a resource file to customize your debugger startup environment:

- Configuration of windows and views (for example, size, screen location, order)
- Push-button labels and associated debugger commands, including adding and removing buttons and commands

### Online Help

You can get context-sensitive online help for the debugger's window interface. You can also display reference pages for debugger commands. For more information about getting online help, see Section 2.5.

## 2.2 Debugger Windows and Menus

The following sections describe the debugger windows, menus, views, and other screen features.

### 2.2.1 Default Main Window Configuration

By default at startup, the debugger displays a Main Window which consists of:

- Banner
- Menu Bar (see Section 2.2.1.1)
- Source View (see Section 2.2.1.2)
- Source View Context Panel (see Section 2.2.1.3)
  - Process menu
  - Thread menu
  - Call Stack menu
- Push-button panel (see Section 2.2.1.4)
- Command Message View (see Section 2.2.1.5)

The Main Window is shown in Figure 2–1.

When you start the debugger as explained in Section 3.1, the Source View is initially empty. Figure 2–1 shows the Source View after a program has been brought under debugger control (by directing the debugger to run a specific image, in this example, `eightqueens`).

You can customize the startup configuration to your preference as described in Section 4.9.1.

#### 2.2.1.1 Menus on the Main Window

Figure 2–2 and Table 2–1 show and describe the menus on the Main Window.

**Figure 2–2 Menus on the Main Window**

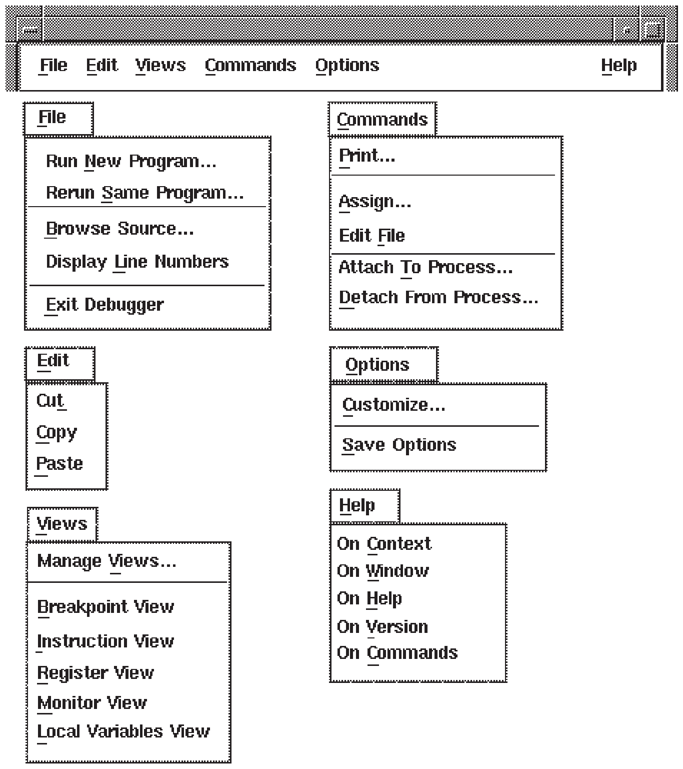


**Table 2–1 Menus on the Main Window**

| Menu | Item | Description |
|---|---|---|
| File | Run New Program... | Brings a program under debugger control by specifying an executable image. |
| | Rerun Same Program... | Reruns the previous program under debugger control. |
| | Browse Source... | Displays the source code in any module of your program. You may set breakpoints on routines. |
| | Display Line Numbers | Displays or hides line numbers in the source display. |
| | Exit Debugger | Ends the debugging session, terminating the debugger. |
| Edit | Cut | Cuts selected text from the window and copies it to the clipboard. |
| | Copy | Copies selected text from the window to the clipboard without removing it from the window. |
| | Paste | Pastes text from the clipboard to a text-entry field or region. |
| Views | Manage Views... | Displays a dialog box which you can use to manage all the views. |
| | Breakpoint View | Toggles the Breakpoint View. |
| | Instruction View | Toggles the Instruction View. |
| | Register View | Toggles the Register View. |
| | Monitor View | Toggles the Monitor View. |
| | Local Variables View | Toggles the Local Variables View. |
| Commands | Print... | Prints the current value of a variable or expression. |
| | Assign... | Changes the value of a variable. |
| | Edit File | Opens the editable source file with the current file displayed. |
| | Attach to Process... | Replace the currently active process with a selected process, leaving the previous process under debugger control. |
| | Detach from Process... | Allows you to detach from a previously attached process. |

(continued on next page)

**Table 2–1 (Cont.) Menus on the Main Window**

| Menu | Item | Description |
|---|---|---|
| Options | Customize... | Modifies, adds, removes, or resequences a button in the push-button pane and the associated debugger command. |
| | Save Options | Saves the customizations that you have made for use in subsequent debugger sessions. |
| Help | On Context | Provides context-sensitive help. |
| | On Window | Provides information about the main window. |
| | On Help | Provides information about the online help system. |
| | On Version | Provides the current version of the debugger. |
| | On Commands | Provides information about debugger commands. |

#### 2.2.1.2 Source View

The Source View in the Main Window shows the following:

- Source display of the code you are debugging and, optionally, the line numbers to the left of the code. (To choose not to display line numbers, select File:Display Line Numbers.
- Breakpoint toggles allow you to activate or deactivate breakpoints on specific source lines or routines in your program.
- Current location pointer to the left of the breakpoint toggles, which points to the line of source code that will be executed when program execution resumes.

The portion of the source display pane containing the breakpoint toggles and current location pointer is referrred to as the annotation area.

For more information about displaying source code, see Section 2.2.1.1 and Section 4.1.

#### 2.2.1.3 Source View Context Panel

Figure 2–3 and Table 2–2 show and describe the menu buttons on the Source View Context Panel of the Main Window.

**Figure 2–3 Menu Buttons on Source View Context Panel**

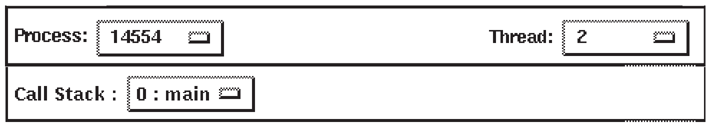


**Table 2–2 Source View Context Panel**

| Button | Function |
|---|---|
| Process | Allows the user to detach the Source View from the current process and to attach it to a different process (see Chapter 5). |
| Thread | Allows the user to detach the Source View from the current thread and to attach it to a different thread (see Chapter 5). |
| Call Stack | Identifies the routine where execution is stopped. This menu lists the sequence of routine calls currently on the stack and lets you set the scope for source display and symbol searches to any routine on the stack (see Section 4.6.1). |

#### 2.2.1.4 Push-Button Panel

Figure 2–4 and Table 2–3 show and describe the default push buttons in the push-button panel. You can modify, add, remove, and resequence buttons and their associated commands as explained in Section 4.9.3.

**Figure 2–4 Default Buttons in the Push-Button Panel**

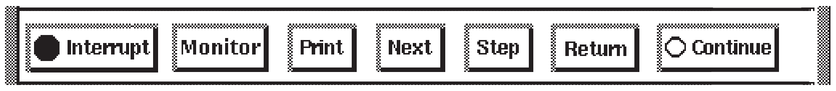


**Table 2–3 Default Buttons in the Push-Button Panel**

| Button | Description |
|---|---|
| Interrupt | Interrupts program execution or a debugger operation without ending the debugging session. |

(continued on next page)

**Table 2–3 (Cont.) Default Buttons in the Push-Button Panel**

| Button | Description |
|---|---|
| Monitor | In the Monitor View, displays the name and current value of a variable that you have selected in a window. Whenever the debugger regains control from your program, it automatically checks the variable and updates the displayed value accordingly. |
| Print | In the Command Message View, displays the current value of a variable whose name you have selected in a window. |
| Next | Executes the program one step unit of execution. By default, this is one executable line of source code. |
| Step | When execution is suspended at a routine call statement, moves execution into the called routine just past the start of the routine. If not at a routine call statement, this push button has the same behavior as the Next push button. |
| Return | Executes the program directly to the end of the current routine. |
| Continue | Starts or resumes execution from the current program location. |

#### 2.2.1.5 Command Message View

The Command Message View, located directly under the push-button panel in the main window, displays any debugger output. Examples of such output are:

- The result of a debugger operation.
- Diagnostic messages.
- Command echo. The debugger translates your window input into commands. The command echo enables you to correlate your input with the corresponding command line that the debugger processes.

The Command Message View has a command-entry prompt `(ladebug)` that enables you to enter commands as explained in Section 2.4.

### 2.2.2 Optional Views Window

Table 2–4 lists the optional views. They are accessible from either the Views menu on the Main or the Optional Views Window.

**Table 2–4 Optional Views**

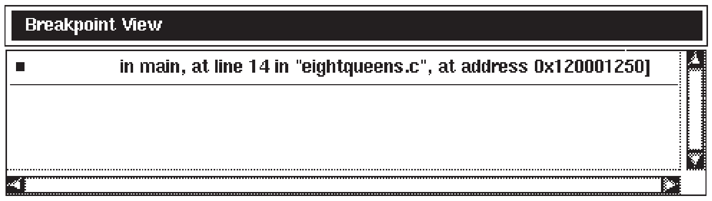
| View | Description |
|---|---|
| Breakpoint | Lists all breakpoints that are currently set and identifies those which are activated, deactivated, or qualified as conditional breakpoints. The Breakpoint View also allows you to modify the state of each breakpoint (see Section 4.4). |
| Monitor | Lists variables whose values you want to monitor as your program executes. The debugger updates the values whenever it regains control from your program (for example, after a step or at a breakpoint). If you choose, you can also change the values of variables (see Section 4.5.3.3). |
| Local Variables | Lists local variables and parameters passed to a routine. If you change the Call Stack level, this view is updated to show the variables at the routine in the given stack. When a new routine is detected at the top of the stack, all variables for the previous routine are removed and variables for the new routine are displayed, (see Section 4.5.4.1). |
| Register | Displays the current contents of all machine registers. The debugger updates the values whenever it regains control from your program. The Register View also enables you to change the values in registers (see Section 4.7). |
| Instruction | Displays the decoded instruction stream of your program and allows you to set breakpoints on instructions. By default, the debugger displays the corresponding source-code line numbers to the left of the instructions. You can choose to suppress these, if you wish (see Section 4.8). |

All views by default are not displayed at startup.

You can move and resize all windows. You can also save a particular configuration of the windows and views so that it is set up automatically when you restart the debugger (see Section 4.9.1).

Figure 2–5 shows the Breakpoint View.

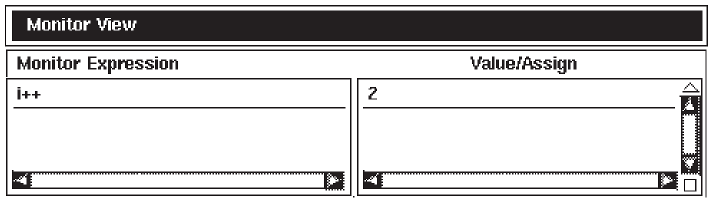
**Figure 2–5 Breakpoint View**



■ in main, at line 14 in "eightqueens.c", at address 0x120001250]

Figure 2–6 shows the Monitor View.

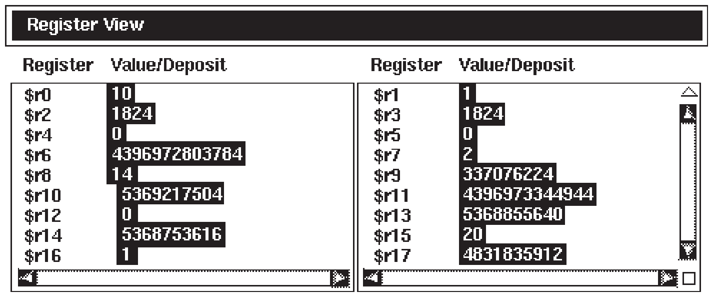
**Figure 2–6 Monitor View**



```
Monitor View
Monitor Expression              Value/Assign
i++                             2
```

Figure 2–7 shows the Register View.

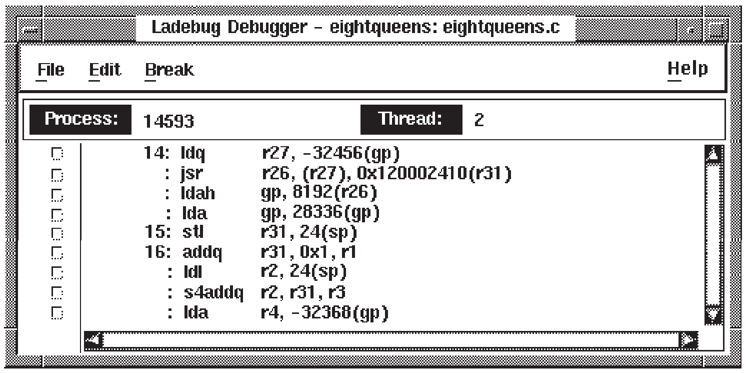
**Figure 2–7 Register View**



```
Register View
Register  Value/Deposit            Register  Value/Deposit
$r0       10                       $r1       1
$r2       1824                     $r3       1824
$r4       0                        $r5       0
$r6       4396972803784            $r7       2
$r8       14                       $r9       337076224
$r10       5369217504              $r11      4396973344944
$r12       0                       $r13      5368855640
$r14       5368753616              $r15      20
$r16       1                       $r17      4831835912
```

Figure 2–8 shows the Instruction View.

**Figure 2–8 Instruction View**

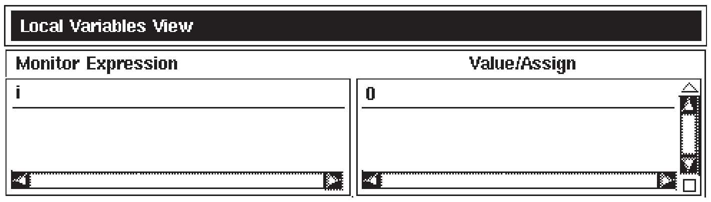


Process: 14593 Thread: 2

```
14: ldq      r27, -32456(gp)
  : jsr      r26, (r27), 0x120002410(r31)
  : ldah     gp, 8192(r26)
  : lda      gp, 28336(gp)
15: stl      r31, 24(sp)
16: addq     r31, 0x1, r1
  : ldl      r2, 24(sp)
  : s4addq   r2, r31, r3
  : lda      r4, -32368(gp)
```

Figure 2–9 shows the Local Variables View.

**Figure 2–9 Local Variables View**



| Monitor Expression | Value/Assign |
|---|---|
| i | 0 |

### 2.2.2.1 Menus on Optional Views Window

Figure 2–10 shows the menus in the Optional Views Window.

**Figure 2–10 Menus on Optional Views Window**

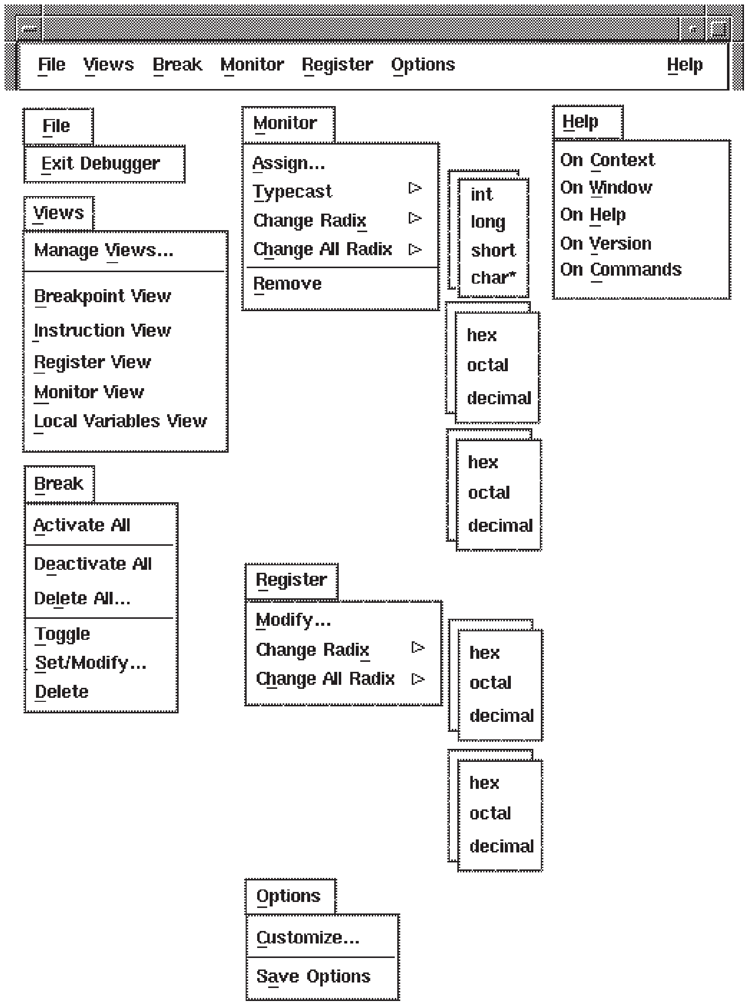


Table 2–5 describes the menus in the Optional Views Window.

**Table 2–5 Menus in Optional Views Window**

| Menu | Item | Description |
|---|---|---|
| File | Exit Debugger | Ends the debugging session, terminating the debugger. |
| Views | Manage Views... | Allows you to manage all views. |
| | Breakpoint View | Toggles the Breakpoint View. |
| | Instruction View | Toggles the Instruction View. |
| | Register View | Toggles the Register View. |
| | Monitor View | Toggles the Monitor View. |
| | Local Variables View | Toggles the Local Variables View. |
| Break | Activate All | Activates any previously deactivated breakpoints (see Section 4.4.4). |
| | Deactivate All | Deactivates any previously activated breakpoints (see Section 4.4.4). |
| | Delete All... | Removes all breakpoints from the debugger's breakpoint list and from the Breakpoint View (see Section 4.4.4). |
| | Toggle | Toggles a breakpoint (see Section 4.4.4). |
| | Set/Modify... | Sets a new breakpoint, optionally associated with a particular condition or action, at a specified location. (See Section 4.4.5 and Section 4.4.6). |
| | Delete | Deletes an individual breakpoint (see Section 4.4.4). |
| Monitor | Assign... | Changes the value of a monitored element. |
| | Typecast | Uses the submenu to typecast output for a selected variable to int, long, short, or char*. |
| | Change Radix | Uses the submenu to change the output radix for a selected variable to hex, octal, or decimal. |
| | Change All Radix | Uses the submenu to change the output radix for all subsequent monitored elements to hex, octal, or decimal. |
| | Remove | Removes an element from the Monitor View. |
| Register | Modify... | Changes the value of a selected register (see Section 4.7). |

(continued on next page)

**Table 2–5 (Cont.) Menus in Optional Views Window**

| Menu | Item | Description |
|---|---|---|
| | Change Radix | Uses the submenu to change current and previous output for selected register to hex, octal, or decimal. |
| | Change All Radix | Uses the submenu to change current and previous output for all registers to hex, octal, or decimal. |
| Options | Customize... | Modifies, adds, removes, or resequences a button in the push-button panel and the associated debugger command (see Section 4.9.1). |
| | Save Options... | Saves the current settings of all window features of the debugger that you can customize interactively, such as the configuration of the windows and views, button definitions, and so on. These settings are observed when you subsequently start up the debugger (see Section 4.9.1). |
| Help | On Context | Provides context-sensitive help (see Section 2.5). |
| | On Window | Provides information about the Optional Views Window. |
| | On Help | Provides information about the online help system. |
| | On Version | Provides the current version of the debugger. |
| | On Commands | Provides information about debugger commands. |

#### 2.2.2.2 Menus on Instruction View

Figure 2–11 shows the menus in the Instruction View.

**Figure 2–11 Menus on Instruction View**

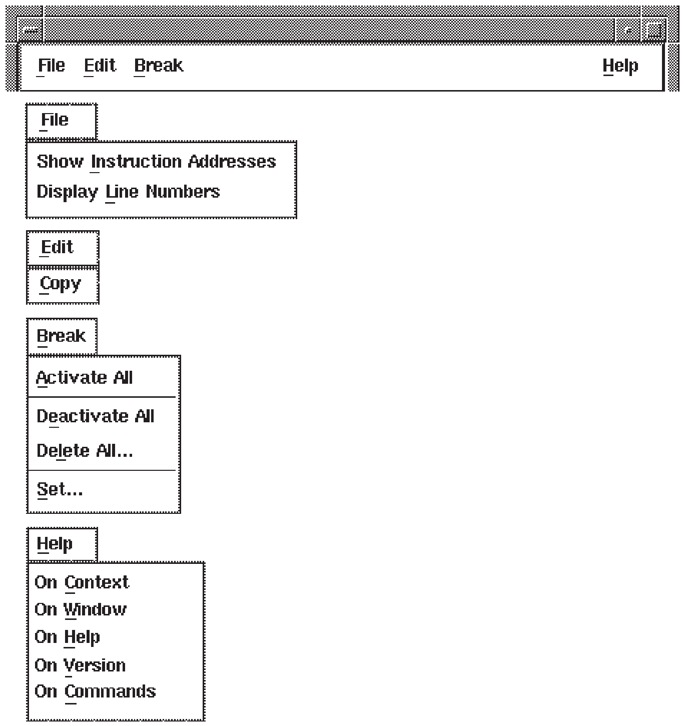


Table 2–6 describes the menus on the Instruction View.

**Table 2–6 Menus on Instruction View**

| Menu | Item | Description |
|---|---|---|
| File | Show Instruction Addresses | Displays the address associated with each instruction listed in the Instruction View. |
| | Display Line Numbers | Displays the line number of your source-code program associated with each instruction or set of instructions listed in the Instruction View. |
| Edit | Copy | Copies text you have selected in the window to the clipboard without removing it from the window. To paste your text from the clipboard to a text-entry field or region, choose Edit:Paste item on the main window. |

(continued on next page)

**Table 2–6 (Cont.) Menus on Instruction View**

| Menu | Item | Description |
|---|---|---|
| Break | Activate All | Activates any previously set breakpoints. |
| | Deactivate All | Deactivates any previously set breakpoints. |
| | Delete All... | Removes all breakpoints from the debugger's breakpoint list and from the Instruction View. |
| | Set... | Sets an individual breakpoint (see Section 4.4.5 and Section 4.4.6.) |
| Help | On Context | Provides context-sensitive help (see Section 2.5). |
| | On Window | Provides information about the Instruction View. |
| | On Help | Provides information about the online help system. |
| | On Version | Provides the current version of the debugger. |
| | On Commands | Provides information about debugger commands. |

## 2.3 Using Context-Sensitive Pop-Up Menus

Context-sensitive pop-up menus in debugger windows and views list common operations associated with your location: when you click MB3, the menu lists actions for the text you have selected, the source line at which you are pointing, or the window pane in which you are working.

### 2.3.1 Source View Pop-Up Menu

To use pop-up menus in the source view, select text or position your mouse pointer in the source display, and press MB3. A pop-up menu appears with the items in Table 2–7. The debugger inserts the selected text or line number in the menu items of the pop-up menu.

**Table 2–7 Source View Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Print [selection] | Evaluates the selected expression and prints its value in the Command Message View. |
| Monitor [selection] | Inserts the selected expression in the monitor list of the Monitor View. |
| Assign [selection] | Provides the Assign dialog box. |

(continued on next page)

**Table 2–7 (Cont.) Source View Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Stop in Routine [selection] | Sets a breakpoint on a selected routine name. |
| Toggle Breakpoint at Line [line number] | Activates/deactivates a breakpoint at the mouse pointer. |
| Temporary Breakpoint at Line [line number] | Sets a temporary breakpoint at the mouse pointer. |
| Go until Line [line number] | Sets a breakpoint and executes until [line number] is reached. |
| Go to Line [line number] | Branches to a specified line without executing source code between the line at which execution is suspended and the specified line. |
| Next | Steps to the next line by stepping over routine calls. |
| Continue | Resumes execution of the program. |

## 2.3.2 Annotation Area Pop-Up Menu

The annotation area is at the left side of the source pane and contains the breakpoint toggles and current location pointer. If you press MB3 while your mouse pointer rests in the annotation area of the source pane, a pop-up menu with the items in Table 2–8 appears.

**Table 2–8 Annotation Area Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Set/Modify Breakpoints... | Displays a dialog box you can use to activate or deactivate a breakpoint and modify the breakpoint attributes (location, condition, or action). |
| Toggle Breakpoint at Line [line number] | Activates/deactivates a breakpoint at the mouse pointer. |
| Temporary Breakpoint at Line [line number] | Sets a temporary breakpoint at the mouse pointer. |
| Go until Line [line number] | Sets a temporary breakpoint and executes until [line number] is reached. |
| Go to Line [line number] | Branches to a specified line without executing source code between the line at which execution is suspended and the specified line. |

(continued on next page)

**Table 2–8 (Cont.) Annotation Area Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Next | Steps to the next line by stepping over routine calls. |
| Continue | Resumes execution of the program. |

## 2.3.3 Command Message View Pop-Up Menu

If you press MB3 while your mouse pointer rests in the Command Message View, a pop-up menu with the items in Table 2–9 appears.

**Table 2–9 Message Region Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Repeat Command [last command] | Reenters your last command. |
| Clear Command Line | Clears the command line. |
| Next | Steps to the next line by stepping over routine calls. |
| Continue | Resumes execution of the program. |

## 2.3.4 Browse Source Pop-Up Menu

If you press MB3 while your mouse pointer rests in the Browse Source dialog box, a pop-up menu with the items in Table 2–10 appears.

**Table 2–10 Source Browser Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Expand | Expands the selected image or module to include its component modules or functions in the Source Browser display. |
| Collapse | Collapses an expanded image, module, or function display. |
| Set Breakpoint | Sets a breakpoint on the selected function. |
| Next | Steps to the next line by stepping over routine calls. |
| Continue | Resumes execution of the program. |

### 2.3.5 Breakpoint View Pop-Up Menu

If you position your mouse pointer in the breakpoint list or annotation area of the Breakpoint View and press MB3, a pop-up menu with the items in Table 2–11 appears.

**Table 2–11 Breakpoint View Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Toggle | Toggles a selected breakpoint. |
| Set/Modify... | Provides the Set/Modify Breakpoint dialog box, which contains information about a selected breakpoint. |
| Delete | Deletes a selected breakpoint. |
| Next | Steps to the next line by stepping over routine calls. |
| Continue | Resumes execution of the program. |

### 2.3.6 Monitor View and Local Variables View Pop-Up Menu

If you position your mouse pointer in the Monitor View or Local Variables View and press MB3, a pop-up menu with the items in Table 2–12 appears.

**Table 2–12 Monitor View and Local Variables View Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Monitor | When selected from the Monitor View, this menu item has no effect. When selected from the Local Variables View, inserts the selected expression in the monitor list of the Monitor View. |
| Expand | Expands a monitored aggregate to show its members. |
| Collapse | Collapses an expanded aggregate. |
| Typecast–> | Provides the list of type choices for modifying values. |
| Change Radix–> | Provides the list of radix choices for modifying values. |
| Next | Steps to the next line by stepping over routine calls. |

### 2.3.7 Register View Pop-Up Menu

If you position your mouse pointer in the Register View and press MB3, a pop-up menu with the items in Table 2–13 appears.

**Table 2–13 Register View Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Change Radix–> | Provides a list of radix choices for modifying values. |
| Next | Steps to the next line by stepping over routine calls. |
| Continue | Resumes execution of the program. |

### 2.3.8 Instruction View Pop-Up Menu

If you position your mouse pointer in the Instruction View and press MB3, a pop-up menu with the items in Table 2–14 appears.

**Table 2–14 Instruction View Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Print [selection] | Evaluates the selected text and prints its value in the Command Message View. |
| Toggle Breakpoint [line number] | Toggles the breakpoint at your mouse pointer location. |
| Next | Steps to the next line by stepping over routine calls. |
| Continue | Resumes execution of the program. |

The annotation area of the Instruction View contains the breakpoint toggles. If you press MB3 while your mouse pointer rests in the annotation area of the Instruction View, a pop-up menu with the items in Table 2–15 appears.

**Table 2–15 Instruction View Annotation Area Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Toggle Breakpoint [address number] | Toggles the breakpoint at your mouse pointer location. |
| Next | Steps to the next line by stepping over routine calls. |

(continued on next page)

**Table 2–15 (Cont.) Instruction View Annotation Area Pop-Up Menu**

| Menu Item | Description |
|---|---|
| Continue | Resumes execution of the program. |

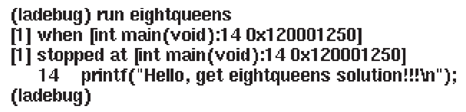
## 2.4 Entering Commands at the Prompt

The debugger command-entry prompt, `(ladebug)` is located in the Command Message View. The command-entry prompt:

- Echoes the command equivalent for each mouse action you make in the window interface.
- Accepts debugger command or internal commands that you choose to enter for the following purposes:
  - As an alternative to using the window interface
  - Because command functionality has no window interface equivalent

Figure 2–12 shows an example of the `run` command entered at the prompt.

**Figure 2–12 Command Prompt**



```
(ladebug) run eightqueens
[1] when [int main(void):14 0x120001250]
[1] stopped at [int main(void):14 0x120001250]
     14    printf("Hello, get eightqueens solution!!!\n");
(ladebug)
```

For more information about bringing a program under debugger control from the prompt, see Section 7.1.2.

When you use the window interface, it translates your input into debugger commands. These commands are echoed in the Command Message View, at the prompt, so that you can correlate your input with the corresponding command line that the debugger processes. Echoed commands are visually indistinguishable from commands that you enter explicitly.

Choose Help:On Commands for online help on debugger commands or type `help` at the command line. For more information about displaying online help, see Section 2.5.

In addition to entering debugger commands interactively at the prompt, you can:

- Recall previously entered commands, using the Up and Down arrow keys.

- Execute commands in debugger initialization files and command files for noninteractive execution.
- Assign commands to new push buttons you define for the debugger's push-button panel. This option allows you to add debugger features of your choice to the window interface.

See Part V, Command Reference for information about Ladebug engine commands with no exact mouse action equivalent.

## 2.5 Displaying Online Help About the Debugger

The following types of online help about the debugger and debugging are available during a debugging session:

- Context-sensitive help—information about an area or object in a window or dialog box (Section 2.5.1)
- Command-oriented help–information on debugger commands (Section 2.5.2)

### 2.5.1 Displaying Context-Sensitive Help

Context-sensitive help is information about an area or object in a window or a dialog box.

To display context-sensitive help:

1. Choose Help:On Context in a debugger window. The pointer shape changes to a question mark (?).
2. Place the question mark on an object or area in a debugger window or dialog box.
3. Click on MB1. Help for that area or object is displayed in a Help window.

To display context-sensitive help for a dialog box, you can also click on the Help button in the dialog box.

### 2.5.2 Displaying Command-Oriented Help

You can display help about debugger commands using the following methods:

- Choose Help:On Commands on the main or optional views window.
- Enter `help` at the command-entry prompt to display common Ladebug commands.

# 3 Starting and Ending a Debugging Session: Graphical User Interface

The sections in this chapter are:

- Starting the debugger from within CDE (Section 3.1)
- Starting the debugger from a command-line prompt. (Section 3.2)
- When your program completes execution (Section 3.3)
- Rerunning the same program from the current debugging session (Section 3.4)
- Running another program from the current debugging session (Section 3.5)
- Interrupting program execution and aborting debugger operations (Section 3.6)
- Ending a debugging session (Section 3.7)

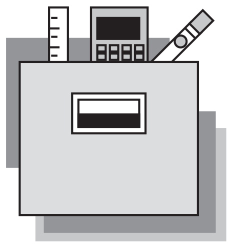
## 3.1 Starting the Debugger from Within CDE

The debugger is integrated with the common UNIX desktop user environment, CDE (Common Desktop Environment). For more information about using CDE, see the *CDE User's Guide*. This section explains the most common ways to start the debugger from within this environment. Section 3.2 explains optional ways to start the debugger.

To start the debugger and bring your program under debugger control:

1. From the CDE Front Panel, click on the Application Manager icon (Figure 3–1).

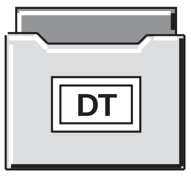
**Figure 3–1 CDE Application Manager Icon**



2. From the CDE Application Manager group window, double click on the Developer's Toolkit icon (Figure 3–2) to open the Developer's Toolkit group.

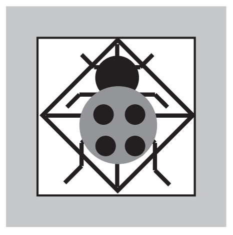
   You can move icons from the Developer's Toolkit group into another group or to the front of the Control Panel depending on your preference. See the *CDE User's Guide* for more information.

**Figure 3–2 CDE Developer's Toolkit Icon**



3. From the Developer's Toolkit group, double click on the Ladebug icon (Figure 3–3).

**Figure 3–3 CDE Ladebug Icon**



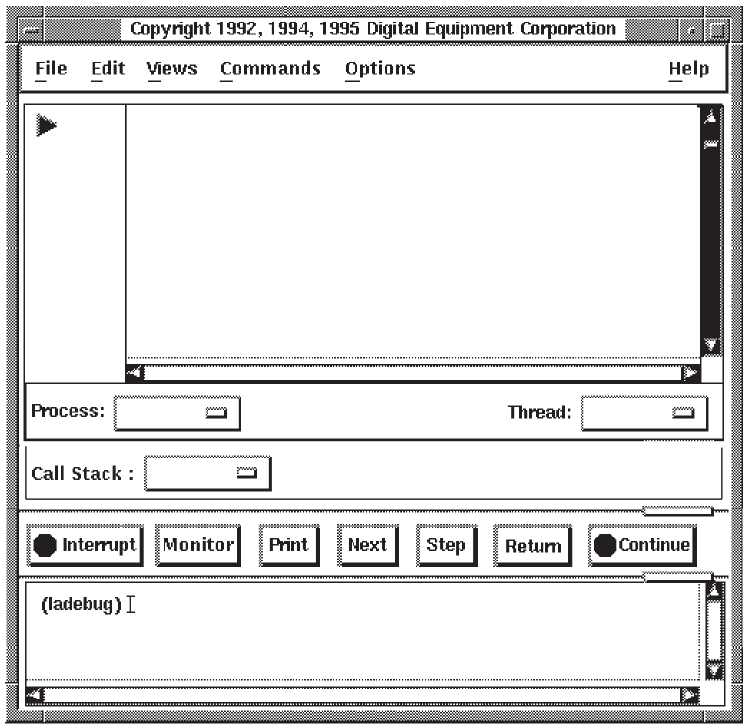
   To access the Ladebug release notes, double click on the README.Ladebug icon.

4. When you double click on the Ladebug icon, a dialog box appears in which you can enter the name of the executable file you want to debug. You can drag an executable file from the CDE File Manager into the dialog box space for file name.

   Alternatively, you can drag an executable file from the CDE File Manager onto the Ladebug icon in the Developer's Toolkit group to start the debugger using that executable.

   If you do not specify a file to debug, click on OK and the debugger starts up as shown in Figure 3–4. The Main Window remains empty until you bring a program under debugger control (see step 6). Upon startup, the debugger executes any user-defined initialization file.

**Figure 3–4 Debugger Window at Startup**

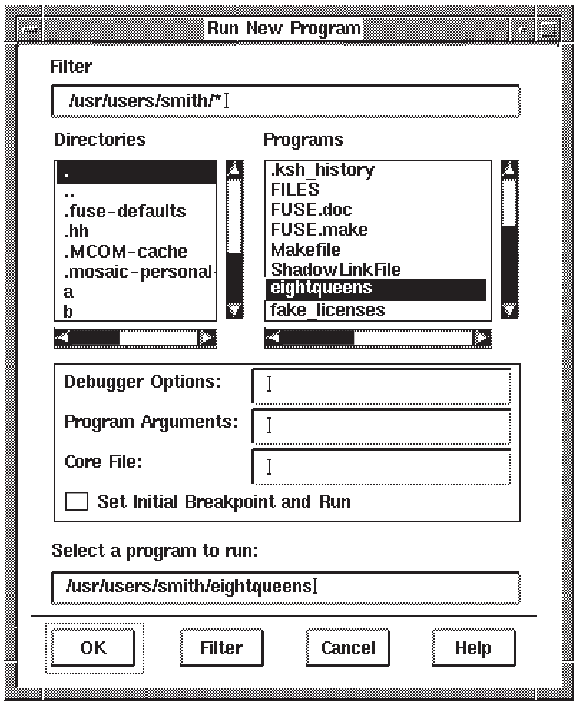


```
(ladebug)
```

5. Bring a specified program under debugger control by using the following steps:

   a. Choose File:Run New Program... on the Main Window. The Run New Program dialog box appears, listing the files in your working directory (see Figure 3–5). Figure 3–5 shows the Run New Program dialog box as it appears if the debugger is not invoked from within CDE. If you invoke the debugger from within CDE, this dialog box displays "Enter path or folder name" in place of "Filter", and an additional window section entitled "Folders".

**Figure 3–5 Running a Program by Specifying a Program File**



Filter: /usr/users/smith/*

Directories: . .. .fuse-defaults .hh .MCOM-cache .mosaic-personal- a b

Programs: .ksh_history FILES FUSE.doc FUSE.make Makefile ShadowLinkFile eightqueens fake_licenses

Select a program to run: /usr/users/smith/eightqueens

b. Click on Filter to display the programs available in the subdirectory. In Programs, click on the name of the program file to be debugged. The program field now shows the program file name.

c. Enter any debugger options in Debugger Options: field. For more information, see the online help about Command Options or the `ladebug` (1) reference page.

d. Enter any program arguments in the Program Arguments: field.

e. Enter the name of the program core dump file in the Core File: field. If such a core file exists in the current directory, you can use the debugger to look at the state of the program when it failed.

f. Toggle the Set Initial Breakpoint and Run button, if necessary.

   The default position for the Set Initial Breakpoint and Run toggle is on. If the toggle is on, the debugger:

   - Loads the program.
   - Sets the initial breakpoint on the program's main entry point.
   - Issues the `run` command to execute the program to that breakpoint.

   If the toggle is off:

   - The debugger loads the program. A breakpoint at the program's entry point is not set and execution of the program does not commence.
   - If you linked your program with the `ald` command or the `amake` command, both of which are specific to Ada programs, the command line will be initially case insensitive.
   - You must enter a `run` command or press the Continue button at the prompt to execute the program. This will cause the program to run to completion, if you have not set any breakpoints.

   The off toggle selection is useful for the following tasks:

| Type of Debugger User | Task |
|---|---|
| All debugger users | Examine a core file. |
| Expert debugger users | Set initial breakpoints in the command-line or window interface. |
| Ada debugger users | Debug Ada package elaboration routines. |

g. Click on OK.

When the program is under debugger control and the Set Initial Breakpoint and Run toggle is on, the debugger:

- Displays a terminal window if the debugger was invoked from within CDE or a Debugger I/O Window if you specified the `-iow` option.
- Displays the program's source code in the Source View, as shown in Figure 4–1.
- Suspends execution at the program's entry point (the start of the main program in most languages). The current-location pointer, to the left of the source code, shows the line whose code will be executed next.

You can now debug your program, (see Chapter 4).

## 3.2 Starting the Debugger from a Command-Line Prompt

You can start the debugger by specifying the program to be debugged with the shell command. For example, you can invoke the debugger from a terminal window and debug a program named `eightqueens`, by entering the following:

```
% dxladebug eightqueens
```

To display a separate Debugger I/O Window, specify the `-iow` option.

The source code is displayed if the module was compiled with `-g`.

By default, the debugger starts up as shown in Figure 4–1, executing any user-defined initialization file. The debugger inserts a breakpoint on the program's entry point and runs to that point, displaying the program's source code in the main window. The current-location pointer (to the left of the source code) shows the line whose code will be executed next.

You can now debug your program.

For more information about debugger startup, see Section 3.1.

## 3.3 When Your Program Completes Execution

When your program completes execution normally during a debugging session, the debugger issues the following message:

```
Process has exited with status...
```

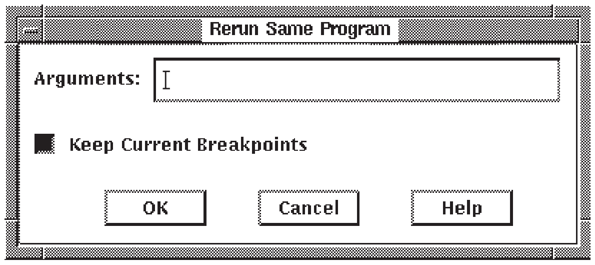
You then have the following options:

- You can rerun the same program from the same debugging session (see Section 3.4).
- You can run another program from the same debugging session (see Section 3.5).
- You can end the debugging session (see Section 3.7).

## 3.4 Rerunning the Same Program from the Current Debugging Session

You can rerun the program currently under debugger control at any time during a debugging session. If you modified the program (created a new binary) during the debugging session, choosing File:Rerun Same Program will bring the new binary under debugger control. The Rerun Same Program dialog box appears (Figure 3–6).

**Figure 3–6 Rerunning the Same Program**



Enter any arguments to be passed to the program in the Arguments: field. Choose whether to keep or not to keep the user-set breakpoints that you previously set, activated, or deactivated (see Section 4.4). Click on OK. An initial breakpoint will be set unless you previously turned the Set Initial Breakpoint and Run button off in the Run New Program dialog box.

When you rerun a program, it is in the same initial state as when you first brought it under debugger control. Any breakpoints that were set in the program remain set. The source display and current location pointer are updated accordingly.

After you revise the program and create a new version of the program file with a new name, choose File:Run New Program... in the main window to bring the new version under debugger control.

## 3.5 Running Another Program from the Current Debugging Session

You can bring another program under debugger control at any time during a debugging session, if you started the debugger as explained in Section 3.1. Follow the procedure in that section to bring another program under debugger control.

## 3.6 Interrupting Program Execution and Aborting Debugger Operations

To interrupt program execution or to abort a debugger operation during a debugging session, click on the Interrupt button on the push-button panel (see Figure 2–4). This is useful if, for example, the program is in an infinite loop.

Clicking on Interrupt does not end the debugging session. Clicking on Interrupt has no effect when the program is not running nor does it have any affect when the debugger is not performing an operation. Click on Continue to resume program execution.

## 3.7 Ending a Debugging Session

To end a debugging session and terminate the debugger, choose File:Exit Debugger on the main window or enter `quit` from the command prompt. This returns control to system level.

To run another program from the current debugging session, see Section 3.5.

# 4 Using the Debugger: Graphical User Interface

This chapter explains how to:

- Display the source code of your program (Section 4.1)
- Edit your program under debugger control (Section 4.2)
- Execute your program under debugger control (Section 4.3)
- Suspend execution with breakpoints (Section 4.4)
- Examine and manipulate program variables (Section 4.5)
- Access program variables (Section 4.6)
- Display and modify values stored in registers (Section 4.7)
- Display the decoded instruction stream of your program (Section 4.8)
- Customize the debugger's window interface (Section 4.9)

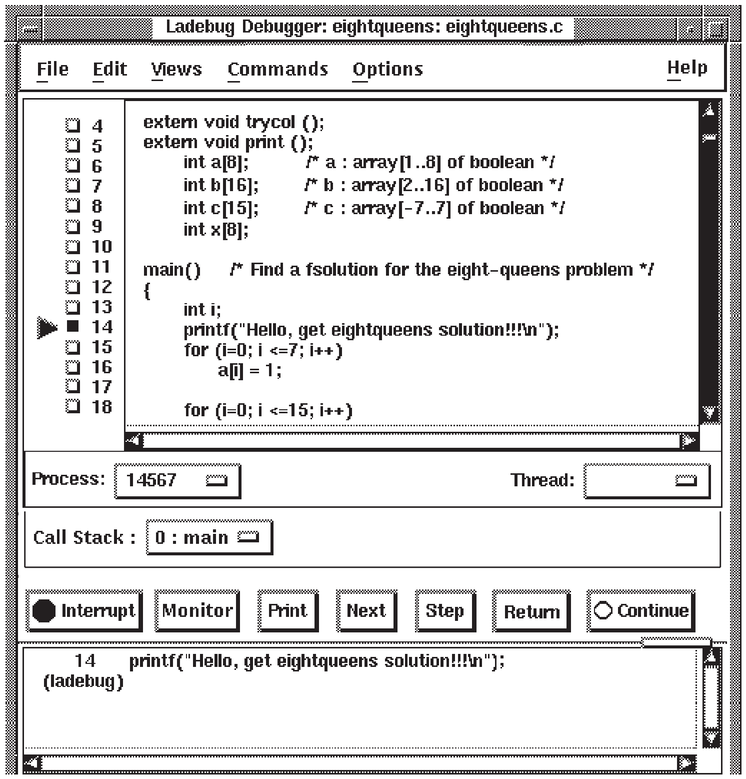
The chapter describes window actions and window menu choices, but you can perform most common debugger operations by choosing items from context-sensitive pop-up menus. To access these menus, click on MB3 while the mouse pointer is in the window area.

You can also enter commands at the command-entry prompt. For information about entering debugger commands, see Section 2.4.

## 4.1 Displaying the Source Code of Your Program

By default, the debugger's Main Window displays the source code of your program (see Figure 4–1).

**Figure 4–1 Source Display**



```
 4   extern void trycol ();
 5   extern void print ();
 6         int a[8];         /* a : array[1..8] of boolean */
 7         int b[16];        /* b : array[2..16] of boolean */
 8         int c[15];        /* c : array[-7..7] of boolean */
 9         int x[8];
10
11   main()    /* Find a fsolution for the eight-queens problem */
12   {
13         int i;
14         printf("Hello, get eightqueens solution!!!\n");
15         for (i=0; i <=7; i++)
16               a[i] = 1;
17
18         for (i=0; i <=15; i++)
```

Process: 14567 Thread:

Call Stack : 0 : main

```
     14     printf("Hello, get eightqueens solution!!!\n");
 (ladebug)
```

Whenever execution is suspended (at a breakpoint for example), the debugger updates the source display by displaying the code that surrounds the point at which execution is paused. The current-location pointer to the left of the source code marks the line whose code will execute next. (A source line corresponds to one or more programming-language statements, depending on the language and coding style.)

By default, the debugger displays line numbers to the left of the source code. These numbers help you identify breakpoints that are listed in the Breakpoint View (see Section 4.4.3). You can choose not to display line numbers. To hide or display line numbers, choose File:Display Line Numbers on the Main Window.

The current-location pointer is normally filled in as shown in Figure 4–1. The current-location pointer is cleared if the displayed code is not that of the routine in which execution is paused.

You can use the scroll bars to show more of the source code. However, you can scroll vertically through only one module of your program at a time.

The following sections explain how to display source code for other parts of your program so that you can set breakpoints in various modules. Section 4.1.2 explains what to do if the debugger cannot find source code for display. Section 4.6.1 explains how to display the source code associated with routines that are currently active on the call stack.

After you navigate the Main Window, you can redisplay the location at which execution is paused by clicking on the top item in the Call Stack menu.

If your program was optimized during compilation, the source code displayed might not reflect the actual contents of some program locations.

### 4.1.1 Displaying Source Code Contained in Another Module

To display source code contained in another module or source file:

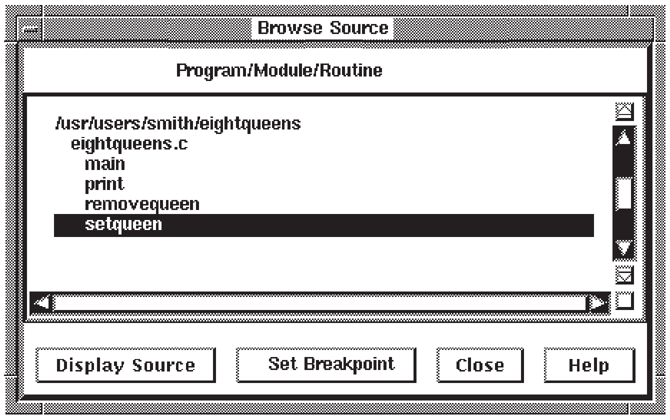
1. Choose File:Browse Source... on the Main Window. The Browse Source dialog box appears listing your program file. See Figure 4–2.
2. Double click on the image to get a list of modules.
3. Double click on the name of the module or source file containing the routine of interest. The names of the program's modules are displayed (indented) under the file name, and the Display Source button is now highlighted.
4. Click on the name of the module whose source code you want to display.
5. Click on Display Source. The source display in the Main Window now displays the module's source code.

Section 4.6.1 describes an alternative way to display routine source code for routines currently active on the call stack using the Call Stack menu.

The Browse Source dialog box displays information in a hierarchical form about all the binary files in your application. For each binary file, it displays information about the modules contained in the binary. For each module, a list of routines in that module can be displayed.

You can manipulate the Browse Source display by clicking on menu items in the Browse Source pop-up menus.

**Figure 4–2 Displaying Source Code in Another Routine**



## 4.1.2 Making Source Code Available for Display

In certain cases, the debugger cannot display source code. Possible causes are:

- The source file has not been compiled with the `-g` compiler option.
- Execution might be paused within a system or library routine for which no symbolic information is intended to be available. In such cases you can quickly return execution to the calling routine by clicking one or more times on the Return push button on the push-button panel (see Section 4.3.5).
- The source file might have been moved to a different directory after it was compiled. Section 4.1.3 explains how to tell the debugger where to look for source files.

If the debugger cannot find source code for a display, it tries to display the source code for the next routine on the call stack for which source code is available. If the debugger can display source code for such a routine, the current-location pointer is cleared. The pointer then marks the source line to which execution returns in the calling routine.

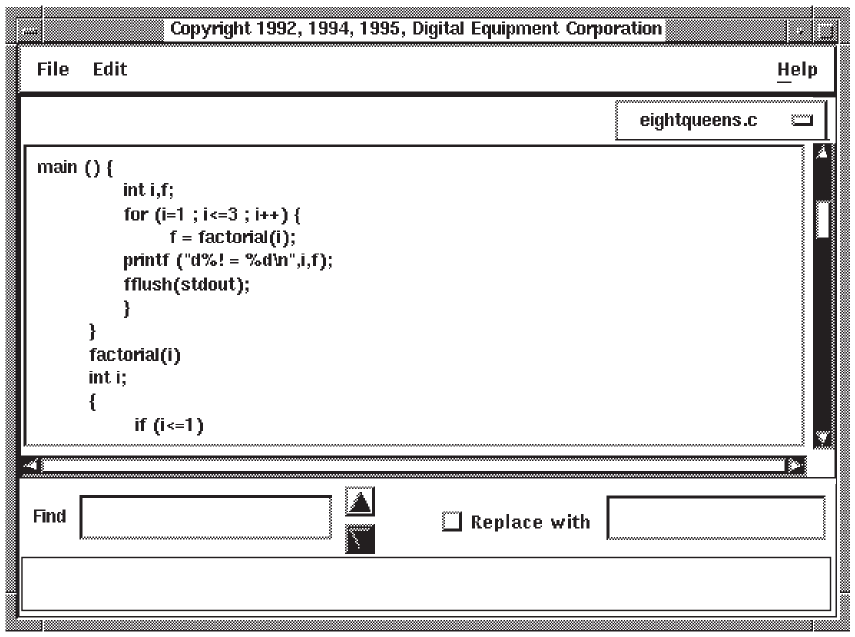
## 4.1.3 Specifying the Location of Source Files

Information about the characteristics and the location of source files is embedded in the debug symbol table of your program. If a source file has been moved to a different directory since compile time, the debugger might not find the file. To direct the debugger to your source files, enter the `use [directory]` command at the command-entry prompt.

## 4.2 Editing Your Program

The debugger provides a simple text editor you can use to edit your source files while debugging your program (Figure 4–3).

**Figure 4–3 Editor Window**



```
main () {
        int i,f;
        for (i=1 ; i<=3 ; i++) {
            f = factorial(i);
        printf ("d%! = %d\n",i,f);
        fflush(stdout);
        }
    }
    factorial(i)
    int i;
    {
         if (i<=1)
```

To invoke the editor, choose Commands:Edit File on the Main Window. By default, the editor window displays an empty text buffer, called `dbg_editor_main`. However, if you are debugging a program when you invoke the editor, the editor window displays this program, names the filled text buffer with its file specification, and places `dbg_editor_main` on the buffer menu at the upper right corner as an alternative text buffer.

The editor allows you to create any number of text buffers by choosing File:New (for empty text buffers) or File:Open (for existing files). The name of each text buffer appears in the buffer menu. You can cut, copy, and paste text across buffers by choosing items from the Edit menu and selecting buffers from the buffer menu.

You can perform forward and backward search and replace operations by entering strings in the Find text and Replace with entry boxes and clicking on a directional arrow.

If you continue to click on a directional arrow, or if you continue to press the Return key, a repeated search for the string occurs in the direction you indicate.

You can also continue a search by choosing the Edit:Find/Replace Next or Edit:Find/Replace Previous.

When you complete your edits, save these to your file by choosing the File:Save or File:Save As. Then recompile and relink your source file so the changes will take effect.

## 4.3 Executing Your Program

This section explains how to:

- Determine where execution is currently paused within your program (Section 4.3.1)
- Start or resume program execution (Section 4.3.2)
- Execute the program one source line at a time, step by step (Section 4.3.3)
- Step into a called routine (Section 4.3.4)
- Return from a called routine (Section 4.3.5

For information about rerunning your program or running another program from the current debugging session, see Section 3.4 and Section 3.5.

### 4.3.1 Determining Where Execution Is Currently Paused

To determine where execution is currently paused within your program:

1. To list the sequence of routine calls that are currently active on the call stack, click on the Call Stack menu. Level 0 denotes the routine in which execution is paused, level 1 denotes the calling routine, and so on.
2. If the current-location pointer is not visible in the Source View, select the top item in the Call Stack menu. This will reset the Source View to the location of the pointer.
3. Look at the current-location pointer:
    - If the pointer is filled in, it marks the source line whose code will execute next (see Section 4.1). The Call Stack menu always shows the routine at level 0 (where execution is paused) when the pointer is filled in.

- If the pointer is cleared, the source code displayed is that of a calling routine, and the pointer marks the source line to which execution returns in that routine as follows:
  - If the Call Stack menu shows level 0 and no routine name is displayed, source code is not available for display for the routine in which execution is paused (see Section 4.1.2).
  - If the Call Stack menu shows a level other than 0, you are displaying the source code for a calling routine (see Section 4.6.1).

### 4.3.2 Starting or Resuming Program Execution

To start program execution or resume execution from the current location, click on the Continue button on the push-button panel (see Figure 2–4) or the Continue item in the pop-up menu on the window or view of your choice.

Once started, program execution continues until one of the following occurs:

- The program completes execution.
- A breakpoint is reached.
- A signal is caught.
- You click on the Interrupt button on the push-button panel on the Main Window.

Whenever the debugger suspends execution of the program, the source display display is updated and the current-location pointer marks the line whose code will execute next.

Letting your program run freely without debugger intervention is useful in situations such as the following:

- To test for an infinite loop.

  If your program does not terminate and you suspect that it is looping, click on the Interrupt button. The source display will show where you interrupted program execution, and the Call Stack menu will identify the sequence of routine calls at that point (see Section 4.3.1).

### 4.3.3 Executing Your Program One Source Line at a Time

To execute one source line of your program, click on the Next button on the push-button panel or the Next pop-up menu item in the window or view of your choice. Note that the Next button executes a routine call, but does not step into it.

After the line executes, the source display is updated and the current-location pointer marks the line whose code will execute next.

Executable lines have a toggle button to their left in the source pane.

Nonexecutable lines do not have a toggle button to their left in the source pane.

Keep in mind that if you optimized your code at compilation time, the source code displayed might not reflect the code that is actually executing (see Section 1.4).

### 4.3.4 Stepping into a Called Routine

When program execution is paused at a routine call statement, clicking on the Next push button executes the called routine in one step. The debugger suspends execution at the next source line in the calling routine (assuming no breakpoint was set within the called routine). "Stepping over" called routines lets you move through the code quickly without having to trace execution. Use the Next button to step over routines.

To "step into" a called routine so that you can execute it one line at a time:

1. Suspend execution at the routine call statement by setting a breakpoint (see Section 4.4) and then clicking on the Continue push button on the push-button panel.
2. When execution is paused at the breakpoint, click on the Step push button on the push-button panel. This moves execution just past the start of the called routine.

Once execution is within the called routine, click on the Next push button to execute the routine line by line.

Clicking on the Step button when execution is not paused at a routine call statement is the same as clicking on the Next push button.

### 4.3.5 Returning from a Called Routine

When execution is suspended within a called routine, you can execute your program directly to the end of that routine by clicking on the Return button on the push-button panel on the Main Window, returning you to the calling routine.

The Return button is particularly useful if you have inadvertently stepped into a routine that is out of the scope of the bug you are tracking.

## 4.4 Suspending Execution by Setting Breakpoints

A breakpoint is a location in your program at which you want execution to stop so that you can perform some action such as checking the current value of a variable, stepping into a routine, etc. When using the window interface, you can set breakpoints on:

- Specific source lines
- Specific routines (for example, functions, subprograms)
- Specific instructions (displayed in the Instruction View–see Section 4.8)

A breakpoint with no condition associated with it can be set by using the Break menu, but it is often simpler to click on the toggle push button in the source display.

You can also qualify breakpoints as follows:

- You can set a conditional breakpoint which triggers only when a specified relational expression is evaluated as true.
- You can set an action breakpoint which executes one or more specified system-specific commands when the breakpoint triggers.

You can set a breakpoint that is both a conditional and an action breakpoint. The following sections explain these breakpoint options.

### 4.4.1 Setting Breakpoints on Source Lines

You can set a breakpoint on any source line that has a toggle button to its left in the source pane. These are the lines for which the compiler has generated executable code (for example, routine declarations, assignment statements).

To set a breakpoint on a source line:

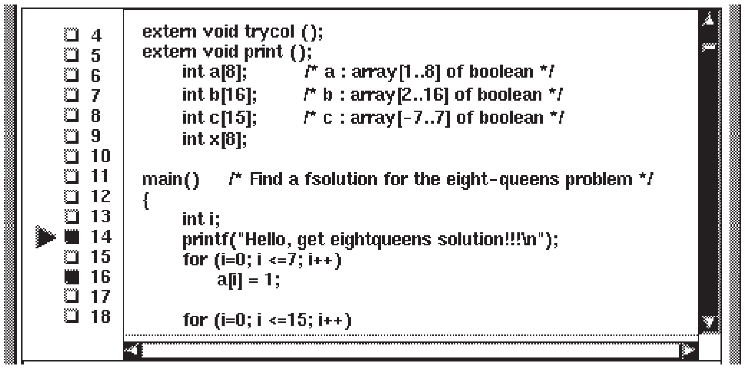
1. Find the source line on which you want to set a breakpoint (see Section 4.1).

2. To set the breakpoint, do one of the following:
   - Click on the toggle button to the left of that line.
   - Select the Toggle Breakpoint at line [line number] item in the pop-up menu in the annotation area or source display.
   - Select the Temporary Breakpoint at line [line number] item in the pop-up menu in the annotation area or source display.
   - Select the Toggle pop-up menu item in the Breakpoint View to activate or deactivate a selected breakpoint.
   - Enter the `stop at` or `stop in` commands at the command prompt.
   - Use the Source Browser.

   The breakpoint is set when the button is filled in. The breakpoint is set at the start of the source line—that is, on the first machine-code instruction associated with that line.

Figure 4–4 shows that a breakpoint has been set on the start of line 14.

**Figure 4–4 Setting a Breakpoint on a Source Line**



```
 4   extern void trycol ();
 5   extern void print ();
 6          int a[8];          /* a : array[1..8] of boolean */
 7          int b[16];         /* b : array[2..16] of boolean */
 8          int c[15];         /* c : array[-7..7] of boolean */
 9          int x[8];
10
11   main()     /* Find a fsolution for the eight-queens problem */
12   {
13          int i;
14          printf("Hello, get eightqueens solution!!!\n");
15          for (i=0; i <=7; i++)
16                a[i] = 1;
17
18          for (i=0; i <=15; i++)
```

To set a breakpoint in the Instruction View:

1. In the Instruction View, find the instruction on which you want to set a breakpoint.
2. Click on the toggle button to the left of that line. (The breakpoint is set when the toggle button is filled in.)

### 4.4.2 Setting Breakpoints on Routines with Source Browser

The Browse Source dialog box displays hierarchical information about all the binary files in your application. For each binary file, it displays information about the modules contained in the binary. For each module, you can display a list of routines in that module.

Setting a breakpoint on a routine enables you to move execution directly to the routine and inspect the local environment.

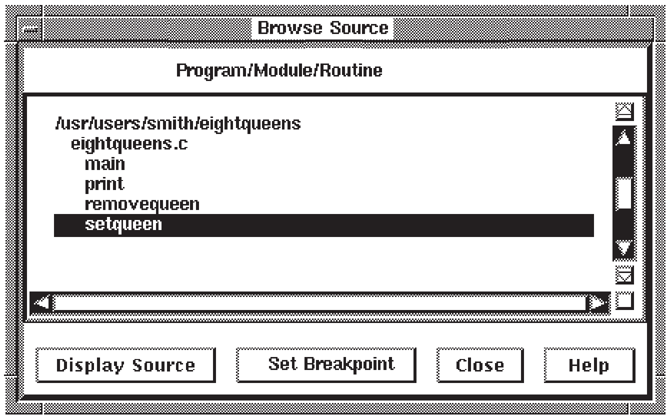
To set a breakpoint on a routine:

1. Choose File:Browse Source... on the Main Window (see Figure 2–2). The Browse Source dialog box appears, listing your program file.
2. Double click on the name of your program file. The names of any additional source files required for compilation are displayed (indented) under the program name.
3. Double click on the name of the file where you want to set a breakpoint. The names of the routines in that file are displayed in the Routine column (see Figure 4–5).
4. Do one of the following:
   - Double click on the name of the routine on which you want to set a breakpoint.
   - Click on the name of the routine and then on the Set Breakpoint button.

     Either of these actions sets a breakpoint at the start of the routine (directly after any prolog code).

In the source pane, the toggle button to the left of the source line that contains the start of the routine is now filled in, confirming that the breakpoint is set. (If the Instruction Window is open, the breakpoint will also display for the corresponding instruction.)

**Figure 4–5 Setting a Breakpoint on a Routine**



## 4.4.3 Identifying the Currently Set Breakpoints

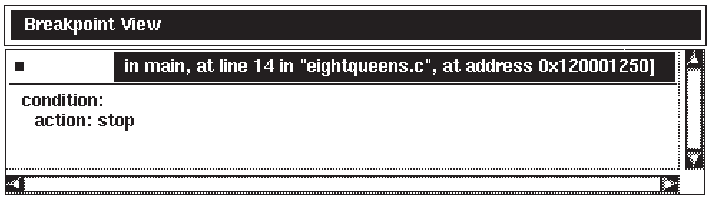
There are two ways to determine which breakpoints are currently set:

- Choose Views:Manage Views... on the Main or Optional View Window. The Manage Views dialog box appears with toggles for all the views. You can either toggle Breakpoint View or you can select Breakpoint View directly from the Views menu. Breakpoints are listed in the Breakpoint View display. A filled-in button in the State column indicates that the breakpoint is set and active. A cleared button indicates that the breakpoint is deactivated.

  Double clicking on an entry in the Breakpoint View expands the one-line entry into a description of the action and condition (if any) that are currently associated with the breakpoint, whether or not the breakpoint is active or deactivated (see Figure 4–6).

- Scroll through your source or instruction code and note the lines whose breakpoint button is filled in. This method can be time consuming and also does not show which breakpoints were set and then deactivated (see Section 4.4.4).

**Figure 4–6 Breakpoint View with Action and Condition**



in main, at line 14 in "eightqueens.c", at address 0x120001250]

condition:
  action: stop

## 4.4.4 Deactivating, Activating, and Deleting Breakpoints

After a breakpoint is set, you can deactivate, activate, or delete it.

Deactivating a breakpoint causes the debugger to ignore the breakpoint during program execution. However, the debugger keeps the breakpoint listed in the Breakpoint View so that you can activate it at a later time, for example, when you rerun the program (see Section 3.4). The following are procedures to deactivate or activate breakpoints:

- To deactivate or activate a specific breakpoint, toggle the button for that breakpoint in the source display, the instruction code display, or in the Breakpoint View.

  In the Breakpoint View, you can also choose Break:Toggle or press MB3, if the breakpoint is currently activated.

- To deactivate or activate all breakpoints, choose Break:Deactivate All or Break: Activate All on the Main, Instruction, or Optional Views Window.

When you delete a breakpoint, it is no longer listed in the Breakpoint View so that later you cannot activate it from that list. You would have to reset the breakpoint as explained in Section 4.4.1 and Section 4.4.2. The following are procedures to delete breakpoints:

- To delete a specific breakpoint, choose Break:Delete on the Optional Views Window or press MB3 from the Breakpoint pop-up menu.

- To delete all breakpoints, choose Break:Delete All... on the Main, Instruction, or Optional Views Window.

### 4.4.5 Setting and Modifying a Conditional Breakpoint

A conditional breakpoint suspends execution only when a specified expression is evaluated as true. For example, you can specify that a breakpoint take effect when the value of a variable in your program is 4. The breakpoint is ignored if the value is other than 4.

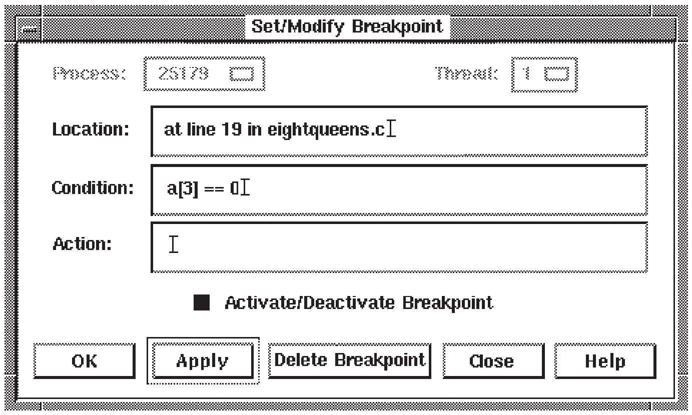
The debugger evaluates the conditional expression when the breakpoint triggers during execution of your program.

To set a conditional breakpoint:

1. Display the source or instruction line on which you want to set the conditional breakpoint (see Section 4.1).
2. Display the Set/Modify Breakpoint dialog box in one of the following ways:
   - Press Ctrl/MB1 on the button to the left of the source line, an instruction line, or a breakpoint entry in the Breakpoint View or press MB3 in the annotation area. This causes the Set/Modify Breakpoint dialog box to appear, which shows the source line you selected in the Location: field (see Figure 4–7).
   - Select a breakpoint entry in the Breakpoint View on the Optional Views window. Press MB3 to access the Breakpoint pop-up menu and choose Set/Modify. The Set/Modify Breakpoint dialog box appears with the location of the source line showing in the Location field.
   - Choose Break:Set... (Instruction Window) or Break:Set/Modify (Optional Views Window). When the Set/Modify dialog box appears, enter the location in the Location field, as shown in the following examples:

     ```
     at line 10 in factorial.c
     in routine seven
     at address 0x01024b
     ```
3. Enter a relational expression in the Condition: field of the dialog box. The expression must be valid in the source language. For example, `a[3] == 0` is a valid relational expression in the C language.
4. Click on OK. The conditional breakpoint is now set. The debugger indicates that a breakpoint is conditional by changing the shape of the breakpoint's button from a square to a diamond.

**Figure 4–7 Setting a Conditional Breakpoint**



Process: 25179 Thread: 1
Location: at line 19 in eightqueens.c
Condition: a[3] == 0
Action:

The following procedure modifies a conditional breakpoint; it can be used to change the location or condition associated with an existing conditional breakpoint, or to change an unqualified breakpoint into a conditional breakpoint:

1. Press Ctrl/MB1 on the button to the left of a source line, an instruction code line, or a breakpoint entry in the Breakpoint View.
2. Click on a breakpoint entry in the Breakpoint View, and choose the Set/Modify item from the Break menu.
3. Enter a relational expression in the Condition: field of the dialog box. The expression must be valid in the source language. For example, `a[3] == 0` is a valid relational expression in the C language.
4. Click on OK. The conditional breakpoint is now set. The debugger indicates that a breakpoint is conditional by changing the shape of the breakpoint's button from a square to a diamond.
5. Press MB3 over the breakpoint item in the Breakpoint View.

### 4.4.6 Setting and Modifying an Action Breakpoint

When an action breakpoint triggers, the debugger suspends execution and then executes a specified list of commands.

To set an action breakpoint:

1. Display the source or instruction line on which you want to set the action breakpoint (see Section 4.1).

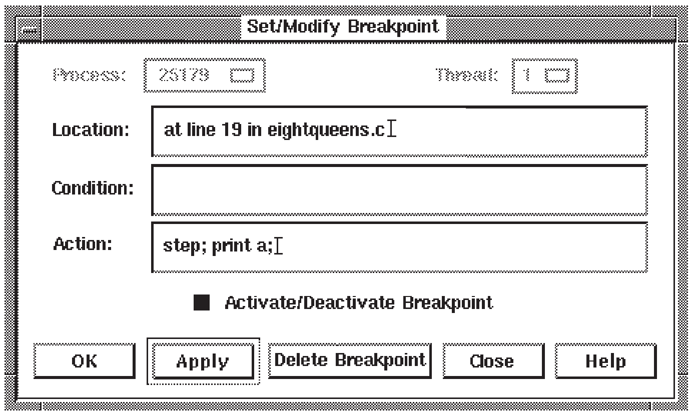
2. Display the Set/Modify Breakpoint dialog box in one of the following ways:

   - Press Ctrl/MB1 on the button to the left of the source line, an instruction line, or a breakpoint entry in the Breakpoint View or press MB3 in the annotation area. This causes the Set/Modify Breakpoint dialog box to appear, showing the source line you selected in the Location: field (see Figure 4–7).
   - Select a breakpoint entry in the Breakpoint View on the Optional Views window. Press MB3 to access the Breakpoint pop-up menu, and choose the Set/Modify menu item. The Set/Modify Breakpoint dialog box appears with the location of the source line showing in the Location: field.
   - Choose the Break:Set/Modify (Optional View Window). When the Set/Modify Breakpoint dialog box appears, enter the location in the Location: field, as shown in the following examples:

     ```
     at line 10 in factorial.c
     in routine seven
     at address 0x01024b
     ```

3. Enter one or more debugger commands in the Action: field of the dialog box. For example: `assign x[j]=3; step; print a;`
4. Click on OK. The action breakpoint is now set (see Figure 4–8).

**Figure 4–8 Setting an Action Breakpoint**


Process: 25179
Thread: 1
Location: at line 19 in eightqueens.c
Condition:
Action: step; print a;
■ Activate/Deactivate Breakpoint

The following procedure modifies an action breakpoint; that is, it can be used to change the command associated with an existing action breakpoint, or to change an unqualified breakpoint into an action breakpoint:

1. Do one of the following:
   - Press Ctrl/MB1 on the button to the left of a source line, an instruction code line, or a breakpoint entry in the Breakpoint View or press MB3 in the annotation area.
   - Click on a breakpoint entry in the Breakpoint View, press MB3 and choose the Set/Modify menu item from the Breakpoint pop-up menu.
   - Click on a breakpoint entry in the Breakpoint View, and choose the Break:Set/Modify.
   - Choose Break:Set/Modify from the debugger Optional View Window. Press MB3 over the breakpoint item in the Breakpoint View.
2. Enter one or more debugger commands in the Action: field of the dialog box. For example: `assign x[j]=3; step; print a;`
3. Click on OK. The action breakpoint is now modified.

## 4.5 Examining and Manipulating Variables

This section explains how to:

- Use available options to examine and manipulate variables (Section 4.5.1)
- Select variable names from windows (Section 4.5.2)
- Display the value of a variable (Section 4.5.3)
- Change the value of a variable (Section 4.5.4)

### 4.5.1 Available Options

Table 4–1 describes options available for selecting and changing variables and values.

**Table 4–1 Available Options for Selecting and Changing Variables and Values**

| Option | Attributes |
|---|---|
| Local Variables View | • Local variables monitored automatically by debugger. You do not have to select names as with the Print button or Monitor View.<br>• Cannot monitor global variables.<br>• Can expand aggregates into their components.<br>• Can assign new values directly in the Local Variables View. |
| Print button | • Must select a variable name.<br>• Can display values of global names or local variables.<br>• Must select name and press the Print button each time you want to see the updated value.<br>• Shows output of the debugger in the Command Message View. |
| Monitor View | • Must select a variable name.<br>• Can monitor global or local variables (when selected local variables are active).<br>• Shows output in the Monitor View.<br>• Can expand aggregates into their components.<br>• Can assign new values directly in the Monitor View. |
| Print Dialog Box | Allows for typecasting. |
| Assign Dialog Box | Allows for typecasting. |

### 4.5.2 Selecting Variable Names from Windows

Use the following techniques to select variable names from windows.

When selecting names, follow the syntax of the source programming language:

- To specify a scalar (nonaggregate) variable, such as an integer, real, Boolean, or enumeration type, select the variable's name.
- To specify an entire aggregate, such as array or structure (record), select the variable's name.
- To specify a single element of an aggregate variable, select the entity using the language syntax. For example:
  - The string `arr2[7]` specifies element `7` of array `arr2` in the C language.
- To specify the object designated by a pointer variable, select the entity following the language syntax. For example, the string `*int_point` specifies the object designated by pointer `int_point` in the C language.

Select character strings from windows in one of the following ways:

- To select a string delimited by blank spaces, use the standard window interface word-selection technique in any window: position the pointer on that string and then double click on MB1.
- To select an arbitrary character string, use the standard window interface text-selection technique in any window: position the pointer on the first character, press and hold MB1 while dragging the pointer over the string, and then release MB1.
- You also have the option of using language-sensitive text selection in the debugger source display. To select a string delimited by language-dependent identifier boundaries, position the pointer on that string and press Ctrl/MB1.

  For example, suppose the source display contains the character string `arr2[m]`, then:
  - To select `arr2`, position the pointer on `arr2` and press Ctrl/MB1.
  - To select `m`, position the pointer on `m` and press Ctrl/MB1.

### 4.5.3 Displaying the Current Value of a Variable

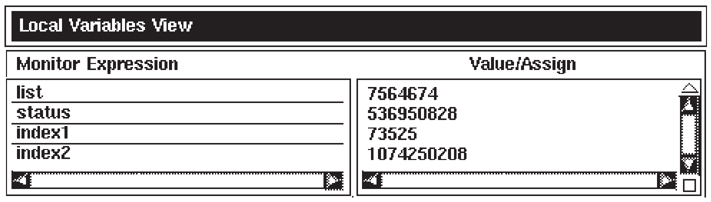
You can display the current value of a variable or expression as described in Table 4–1.

The following sections describe these methods.

#### 4.5.3.1 Using the Local Variables View

In the Local Variables View, you can monitor the values of all local variables and parameters passed to a routine, as shown in Figure 4–9.

**Figure 4–9 Local Variables View**



| Monitor Expression | Value/Assign |
|---|---|
| list | 7564674 |
| status | 536950828 |
| index1 | 73525 |
| index2 | 1074250208 |

The debugger automatically displays these values. It checks and updates them whenever the debugger regains control from your program (for example, after a step or at a breakpoint).

To monitor a local variable or parameter using the Local Variables View, select Views:Local Variables View or select Views:Manage Views and toggle the Local Variables View button on the Main or Optional Views Window.

The debugger automatically lists all local variable and parameter names (in the Monitor Expression column) and current values (in the Value/Assign column).

You cannot add or remove an entry to the local variables and parameters list. The debugger automatically removes previous entries and adds new entries when a new routine appears at the top of the Call Stack.

To monitor a global variable, use the Monitor View (see Section 4.5.3.3). Section 4.5.3.3 also explains how to monitor an aggregate variable and a pointer variable. The technique is the same whether you use the Monitor View or the Local Variables View.

#### 4.5.3.2 Using the Print Button

To display the current value of a variable using the Print button:

1. Find and select the variable name or expression in a window, as explained in Section 4.5.2.
2. Click on the Print button on the push-button panel of the Main Window. The debugger displays the variable or expression and its current value in the Command Message View. (This is the value of the variable or expression in the current scope, which might not be the same as the source location where you selected the variable name or expression.)

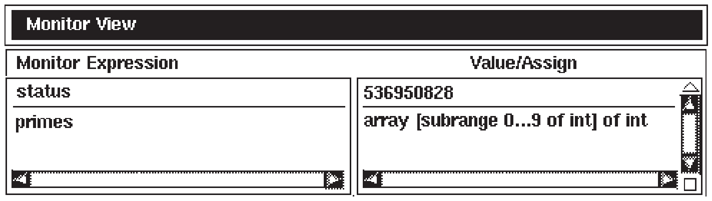
#### 4.5.3.3 Using the Monitor View

When you monitor a variable, the debugger displays the value in the Monitor View and automatically checks and updates the displayed value whenever the debugger regains control from your program (for example, after a step or at a breakpoint).

To monitor a variable or expression with the Monitor View:

1. Choose Views:Monitor View or Views:Manage Views... and toggle the Monitor View button. The Monitor View appears in the Optional Views Window (see Figure 4–10).
2. Find and select the variable name in a window, as explained in Section 4.5.2.
3. Click on the Monitor push button on the push-button panel of the Main Window. The debugger does the following:
   a. Puts the selected variable name or expression in the Monitor Expression column
   b. Puts the current value of the variable in the Value/Assign column or, if the variable is an aggregate, displays the full type name of the aggregate.

**Figure 4–10 Monitoring a Variable**



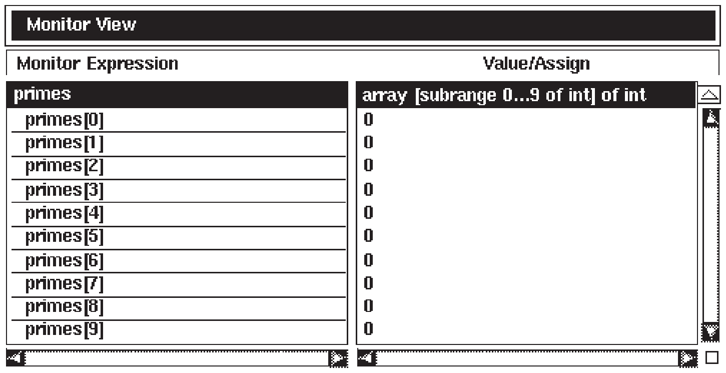
```
status      536950828
primes      array [subrange 0...9 of int] of int
```

#### 4.5.3.3.1 Monitoring an Aggregate (Array, Structure) Variable

If you select the name of an aggregate variable such as an array or structure (record) and click on the Monitor push button, the debugger displays the full type name of the aggregate in the Value/Assign column of the Monitor View. To display the values of all elements (components) of an aggregate variable, double click on the variable name in the Monitor Expression column.

The displayed element names are indented relative to the parent name (see Figure 4–11). If an element is also an aggregate, you can double click on its name to display its elements.

**Figure 4–11 Expanded Aggregate Variable (Array) in Monitor View**



```
primes          array [subrange 0...9 of int] of int
  primes[0]     0
  primes[1]     0
  primes[2]     0
  primes[3]     0
  primes[4]     0
  primes[5]     0
  primes[6]     0
  primes[7]     0
  primes[8]     0
  primes[9]     0
```

To collapse an expanded display so that only the aggregate parent name is shown in the Monitor View, double click on the name in the Monitor Expression column.

If you have selected a component of an aggregate variable, and the component expression is itself a variable, the debugger monitors the component that was active when you made the selection. For example, if you select the array component `arr[i]` and the current value of `i` is `9`, the debugger monitors `arr[9]` even if the value of `i` subsequently changes to `10`.

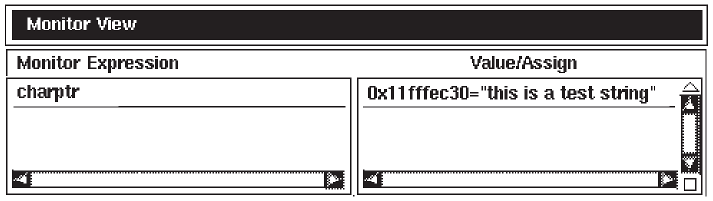
If the aggregate is a local variable, you can also use the Local Variables View to monitor the aggregate.

**4.5.3.3.2 Monitoring a Pointer (Access) Variable** If you select the name of a pointer (access) variable and click on the Monitor push button, the debugger displays the address and value of the referenced object in the Value/Assign column of the Monitor View (see Figure 4–12).

If a referenced object is an aggregate, you can double click on its name to display its elements.

If the pointer is local, you can also use the Local Variables View to monitor the pointer.

**Figure 4–12 Pointer Variable and Referenced Object in Monitor View**



charptr

0x11fffec30="this is a test string"

### 4.5.3.4 Using the Print Dialog Box

The Print dialog box allows you to request typecasting or an altered output radix in the displayed result.

To display the current value with the Print dialog box, select the variable or expression using the method described in Section 4.5.2 and click on Print or:

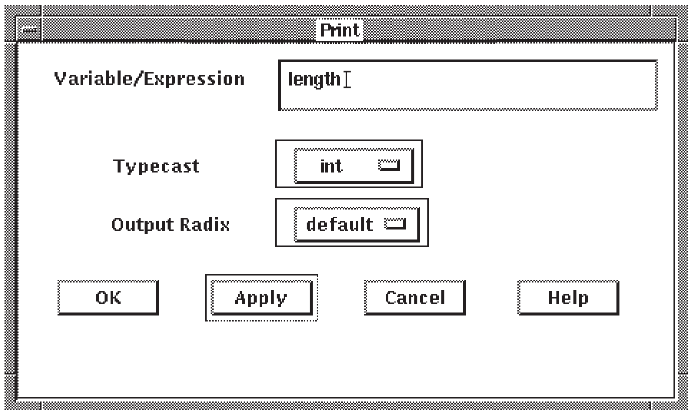
1. Choose Commands:Print... on the Main Window. The Print dialog box appears.
2. Enter the variable name or expression in the Variable/Expression entry box.
3. If you are changing the output type, pull down the menu in the Typecast entry box and click on the desired data type.
4. If you are changing the output radix, pull down the menu in the Output Radix entry box and click on the desired radix.
5. Click on OK.

The debugger displays the variable or expression and its current value in the Command Message View.

Your echoed command and the current value appear in the Command Message View.

Figure 4–13 shows a typecast to `int` for the variable `length`.

**Figure 4–13 Typecasting the Value of a Variable**



Variable/Expression: length

Typecast: int

Output Radix: default

The debugger displays your echoed command and the variable or expression and its current value in the Command Message View.

## 4.5.4 Changing the Current Value of a Variable

You can change the value of a variable with either of the following methods:

- Clicking on a monitored value within the Local Variables View or Monitor View (see Section 4.5.4.1)
- Using the Assign dialog box, accessed from the Commands menu on the Main window or from the source display pop-up menu (see Section 4.5.4.2)

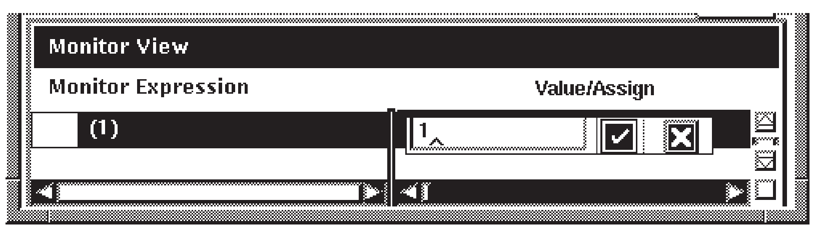
### 4.5.4.1 Clicking on a Monitored Value Within the Local Variables View or Monitor View

To change a value monitored within the Local Variables View or Monitor View (see Figure 4–14):

1. Select the variable as explained in Section 4.5.2.
2. Click on the variable's value in the Value/Assign column of the Local Variable View or Monitor View. A small text edit box appears over that value, which you can now edit.
3. Enter the new value in the text edit box.

4. Click on the check mark (OK) in the text edit box. The text edit box is removed and replaced by the new value, which indicates that the variable now has that value. The debugger notifies you if you try to enter a value that is incompatible with the variable's type and range.

**Figure 4–14 Changing the Value of a Monitored Variable**

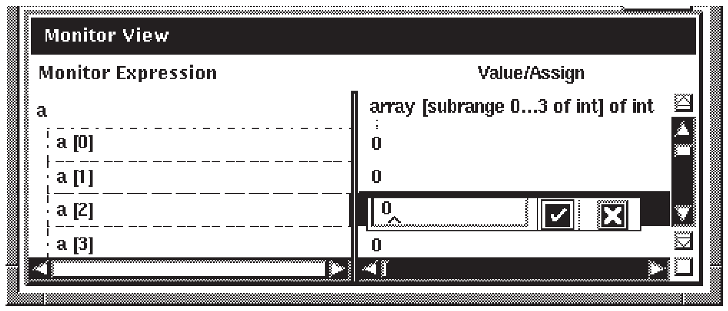


To cancel a text entry and dismiss the text edit box, click on X (Cancel).

You can change the value of only one component of an aggregate variable at a time (such as an array or structure). To change the value of an aggregate variable component (see Figure 4–15):

1. Display the value of the component as explained in Section 4.5.3.3.1.
2. Follow the procedure for changing the value of a scalar variable.

**Figure 4–15 Changing the Value of a Component of an Aggregate Variable**



#### 4.5.4.2 Changing the Value of a Variable with the Assign Dialog Box

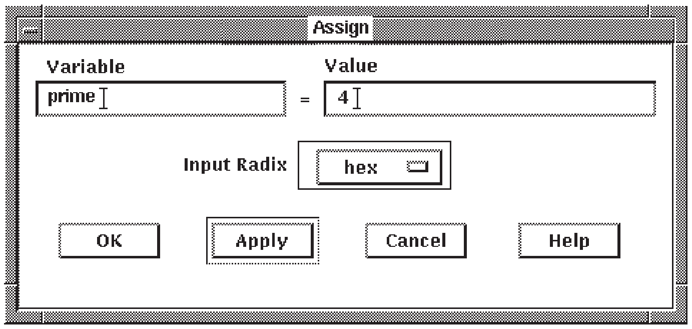
To change the value of a variable with the Assign dialog box:

1. Select the variable as explained in Section 4.5.2.
2. Choose Commands:Assign... on the Main window. The Assign dialog box appears.
3. Enter the variable name and new value in the Variable and Value entry boxes.
4. If you are changing the input radix, pull down the menu in the Input Radix entry box and click on the desired radix.
5. Click on OK.

Figure 4–16 shows a new value and input radix for the variable `prime`.

You can also display the Assign dialog box by clicking MB3 in the source pane and selecting Assign from the pop-up menu.

**Figure 4–16 Changing the Value of a Variable**


Assign

Variable: prime = Value: 4

Input Radix: hex

## 4.6 Accessing Program Variables

This section provides some general information about accessing program variables while debugging.

If your program was optimized during compilation, you might not have access to certain variables while debugging. When you compile a program for debugging, it is best to disable optimization if possible (see Section 1.4).

Before you check on the value of a variable, always execute the program beyond the point where the variable is declared and initialized. The value contained in any uninitialized variable should be considered invalid.

In most cases, the debugger resolves symbol ambiguities automatically, using the scope and visibility rules of the source programming language. For more information about resolving symbol ambiguities, see Chapter 8.

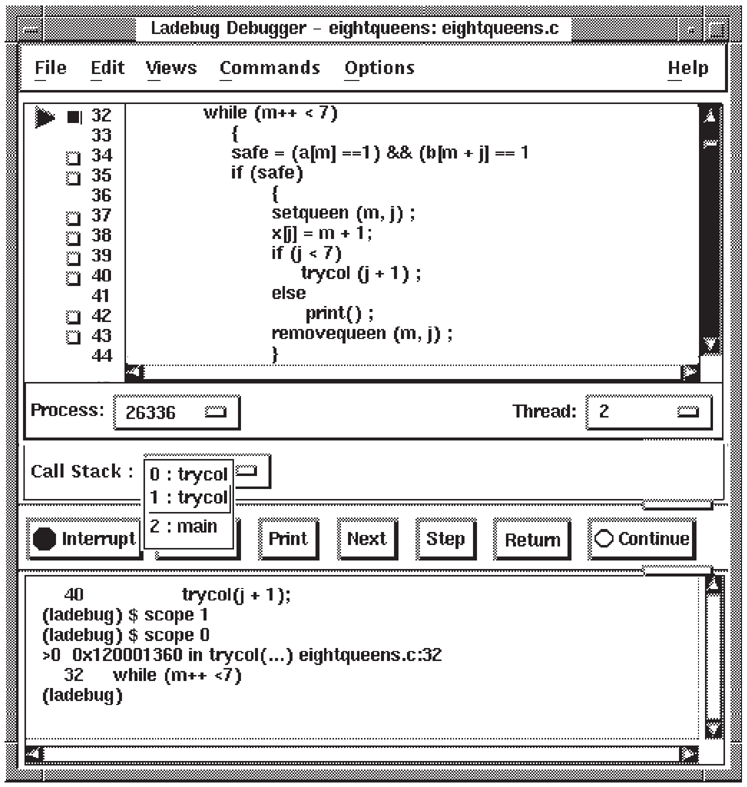
### 4.6.1 Setting the Current Scope Relative to the Call Stack

While debugging a routine in your program, you might want to set the current scope to a calling routine (a routine down the stack from the routine in which execution is currently paused). This enables you to:

- Determine where the current routine call originated
- Determine the value of a variable declared in a calling routine
- Determine the value of a variable during a particular invocation of a routine that is called recursively
- Change the value of a variable in the context of a routine call

The Call Stack menu, shown in Figure 4–17 lists the names of the routines of your program that are currently active on the stack, up to the maximum number of lines that can be displayed on your screen.

**Figure 4–17 Current Scope Set to a Calling Routine**



```
        while (m++ < 7)
            {
            safe = (a[m] ==1) && (b[m + j] == 1
            if (safe)
                {
                setqueen (m, j) ;
                x[j] = m + 1;
                if (j < 7)
                    trycol (j + 1) ;
                else
                     print() ;
                removequeen (m, j) ;
                }
```

Process: 26336    Thread: 2

Call Stack :
0 : trycol
1 : trycol
2 : main

```
   40              trycol(j + 1);
(ladebug) $ scope 1
(ladebug) $ scope 0
>0  0x120001360 in trycol(...) eightqueens.c:32
   32    while (m++ <7)
(ladebug)
```

The numbers on the left side of the menu indicate the level of each routine on the stack relative to level 0, which denotes the routine in which execution is paused.

To set the current scope to a particular routine on the stack, choose the routine's name from the Call Stack menu; for example, `1:trycol` in Figure 4–17. This causes the following to occur:

- The Call Stack menu, when released, shows the name and relative level of the routine that is now the current scope.
- The source display shows that routine's source code.
- The Instruction View (if displayed) shows that routine's decoded instructions.
- The Register View (if displayed) shows the register values associated with that routine call.

- If the scope is set to a calling routine (a Call Stack level other than 0), the debugger grays the current-location pointer.
- The debugger sets the scope for symbol searches to the chosen routine, so that you can examine factors such as variables, within the context of that scope.

When you set the scope to a calling routine, the current-location pointer (which is cleared) marks the source line to which execution will return in that routine. Depending on the source language and coding style used, this might be the line that contains the call statement or some subsequent line.

### 4.6.2 How the Debugger Searches for Variables and Other Symbols

Symbol ambiguities can occur when a symbol is defined in more than one routine or other program unit.

In most cases, the debugger automatically resolves symbol ambiguities. First, it uses the scope and visibility rules of the currently set language. The debugger uses the ordering of routine calls on the call stack to resolve symbol ambiguities.

In some cases, however, the debugger might respond as follows when you specify a symbol that is defined multiple times:

- It might issue a "symbol not unique" message because it is not able to determine the particular declaration of the symbol that you intended.
- It might reference the symbol declaration that is visible in the current scope, instead of the declaration you want.

To resolve such problems, you must specify a scope where the debugger searches for the declaration of the symbol you want:

- If the different declarations of the symbol are within routines that are currently active on the call stack, use the Call Stack menu on the Main window to reset the current scope (see Section 4.6.1).
- Otherwise, enter the appropriate command at the command prompt (`examine` or `monitor`, for example), specifying a pathname prefix with the symbol. For example, if the variable `x` is defined in two routines named `counter` and `swap`, the following command uses the path name `swap\x` to specify the declaration of `x` that is in routine `swap`:

  ```
  (ladebug) examine swap\x
  ```

## 4.7 Displaying and Modifying Values Stored in Registers

The Register View displays the current contents of all machine registers (see Figure 2–7).

To display the Register View, choose Views:Register View or Views:Manage Views... and toggle the Register View button. The Register View appears in the Optional Views Window.

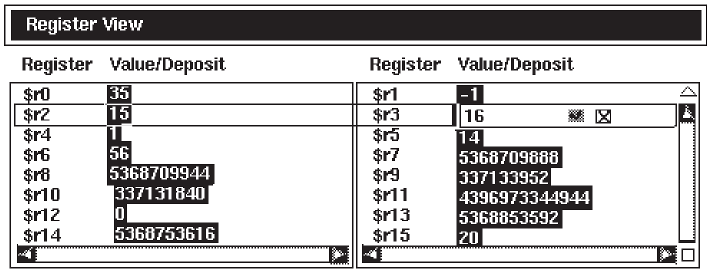
By default, the Register View automatically displays the register values associated with the routine in which execution is currently paused. Any values that change as your program executes are highlighted whenever the debugger regains control from your program.

To display the register values associated with any routine on the call stack, choose its name from the Call Stack menu on the Main Window (see Section 4.6.1).

To change the value stored in a register:

1. Click on the register value in the Register View. A text edit box appears over the current value, which you can now edit.
2. Enter the new value in the text edit box.
3. Click on the check mark (OK) in the text edit box, as shown in Figure 4–18.

**Figure 4–18 Changing a Value in the Register View**



| Register | Value/Deposit | Register | Value/Deposit |
|---|---|---|---|
| $r0 | 35 | $r1 | -1 |
| $r2 | 15 | $r3 | 16 |
| $r4 | 1 | $r5 | 14 |
| $r6 | 56 | $r7 | 5368709888 |
| $r8 | 5368709944 | $r9 | 337133952 |
| $r10 | 337131840 | $r11 | 4396973344944 |
| $r12 | 0 | $r13 | 5368853592 |
| $r14 | 5368753616 | $r15 | 20 |

The text edit box is removed and replaced by the new value, indicating that the register now contains that value.

To cancel a text entry and dismiss the text edit box, click on X (Cancel).

You can also change the radix for modifying values stored in a register as follows:

1. Position your mouse pointer in the Register View and press MB3. A pop-up menu appears.
2. Select Change Radix-> and choose from a list of radix choices for modifying values.

## 4.8 Displaying the Decoded Instruction Stream of Your Program

The Instruction View displays the decoded instruction stream of your program: the code that is actually executing (see Figure 4–19). This is useful if the program you are debugging has been optimized by the compiler. This means that the information in the Main Window does not exactly reflect the code that is executing (see Section 1.4).

To display the Instruction View, choose Views:Instruction View or Views:Manage Views... and toggle the Instruction View button on the Main or Optional Views Window.

By default, the Instruction View automatically displays the decoded instruction stream of the routine in which execution is currently paused. The current location pointer (to the left of the instructions) marks the instruction that will execute next.

By default, the debugger displays line numbers to the left of the instructions with which they are associated. You can choose not to display these line numbers so that more space is devoted to showing instructions. To hide or display line numbers, choose File:Show Instruction Addresses on the Instruction View.

To copy memory addresses or instructions into a command you are entering at the command-entry prompt, select text and choose Edit:Copy in the Instruction View. Then position your mouse pointer at the command you have entered and choose Edit:Paste on the Main Window. (You can also select instruction text to be used with a push-button command you click in the push-button panel of the Main Window.)

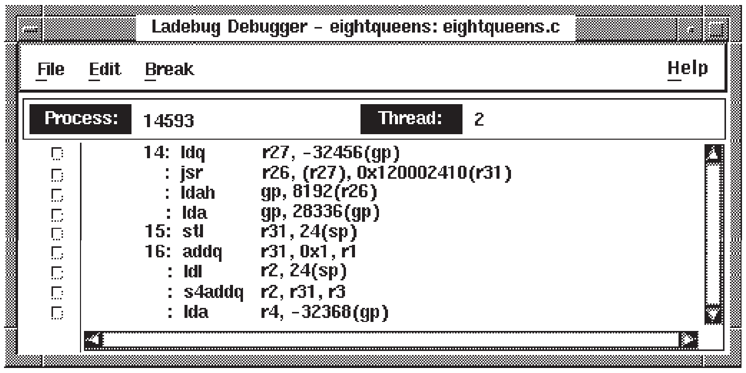
To set breakpoints from the Instruction View, toggle the breakpoint button next to the instruction of interest. The breakpoint is set in the source display, instruction display (if the Instruction View is open), and Breakpoint View (if the Breakpoint View is open). Information on the breakpoint is continuously updated in the source display, and in the Instruction View and Breakpoint View if they are open.

You can also set breakpoints and change breakpoint status by pulling down the Break menu from the Optional Views Window.

After navigating through the Instruction View, to redisplay the location at which execution is paused, click on the Call Stack menu.

To display the instruction stream of any routine on the call stack, choose the routine's name from the Call Stack menu on the Main Window (see Section 4.6.1).

**Figure 4–19 Instruction View**



Process: 14593 Thread: 2

```
14: ldq      r27, -32456(gp)
  : jsr      r26, (r27), 0x120002410(r31)
  : ldah     gp, 8192(r26)
  : lda      gp, 28336(gp)
15: stl      r31, 24(sp)
16: addq     r31, 0x1, r1
  : ldl      r2, 24(sp)
  : s4addq   r2, r31, r3
  : lda      r4, -32368(gp)
```

## 4.9 Customizing the Debugger's Window Interface

You can customize the debugger's window interface in two ways:

- Using the debugger Customize dialog box (see Section 4.9.3)
- Editing the `ladebugresource` resource file (see Section 4.9.4)

Table 4–2 shows the methods you use to customize parameters.

**Table 4–2 Customization Methods**

| Use This Method | To Change These Parameters |
|---|---|
| Customize... dialog box | • Modify, add, remove, or resequence push buttons and the associated debugger command. |
| Edit debugger (`ladebugresource`) resource file | • Define the key sequence to display the dialog box for conditional and action breakpoints.<br>• Define the key sequence to make text selection language-sensitive in the Main Window.<br>• Define the character font for text displayed in specific windows and views.<br>• Define or redefine the commands bound to individual keys on your computer's keypad.<br>• Shut off the Exit Confirmation dialog box.<br>• Shut off debugger command echo in the Command Message View for commands being sent to the engine.<br>• Set the initial breakpoint toggle in the Run New Program dialog box to off. |

The following sections discuss using these methods to customize the debugger window interface.

## 4.9.1 Defining the Startup Configuration for Debugger Windows and Views

To define the startup configuration of the debugger windows and views:

1. While using the debugger, set up the desired configuration of the windows and views.
2. Choose Options:Save Options on the Main or Optional Views Window. This creates a new version of the debugger resource file with the new settings.

When you later start the debugger, the new configuration appears automatically. Adding views to the startup configuration increases the startup time accordingly.

### 4.9.2 Displaying or Hiding Line Numbers by Default

The source pane displays source line numbers by default at debugger startup. To hide (or display) line numbers at debugger startup:

1. While using the debugger, choose File:Display Line Numbers on the Main Window. Line numbers are displayed when a filled-in button appears next to that menu item.
2. Choose Options:Save Options. This creates a new version of the debugger resource file with the new settings.

When you later start the debugger, line numbers are either displayed or hidden accordingly.

### 4.9.3 Modifying, Adding, Removing, and Resequencing Push Buttons

The push buttons on the push-button panel are associated with debugger commands. You can:

- Change a push button's label or the command associated with a push button
- Add a new push button and assign a command to that push button
- Remove a push button
- Resequence a push button

**Note**

You cannot modify or remove the Interrupt push button.

To save these modifications for subsequent debugger sessions, choose Options:Save Options on the Main Window or an optional view. This creates a new version of the debugger resource file with the new definitions.

The following sections explain how to customize push buttons interactively through the window interface. You can also customize push buttons by editing the resource file (see Section 4.9.4).

#### 4.9.3.1 Changing a Button's Label or Associated Command

To change a button's label or the debugger command associated with a push button:

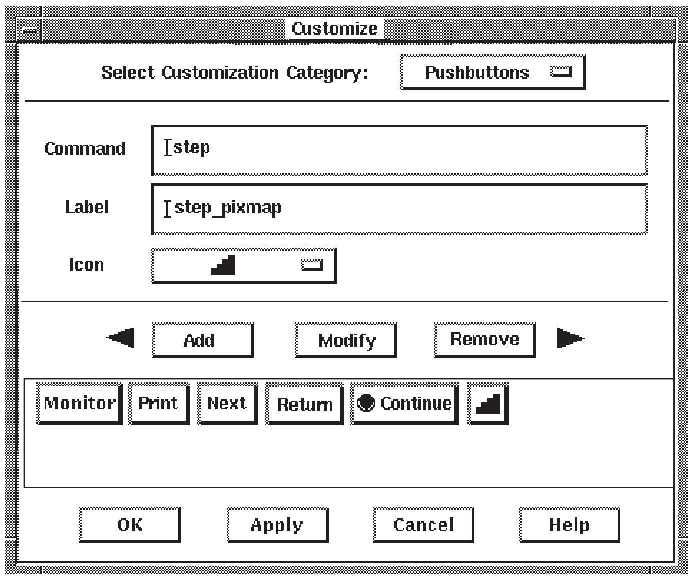
1. Choose Options:Customize... on the Main or Optional View Window. The Customize dialog box appears.
2. Select the Pushbuttons menu option. The Pushbuttons dialog box appears.
3. Click on the push button in the control panel of the dialog box.
4. If changing the push-button label, enter a new label in the Label field or choose a predefined icon from the Icon menu. (If changing the button label, verify that the Icon menu is set to None.)
5. If changing the command associated with the push button, enter the new command in the Command field. For online help about the commands, see Section 2.5.2.

   If the command is to operate on a name or language expression selected in a window, include `%S` in the command name. For example, the following command displays the current value of the variable that is currently selected:

   ```
   print %s
   ```

   If the command is to operate on a name that has a percent sign (%) as the first character, specify two percent signs.
6. Click on Modify. The push button is modified in the dialog box push-button display.
7. Click on Apply. The push button is modified in the push-button panel of the Main Window (see Figure 4–20).

**Figure 4–20 Changing the Step Button Label to an Icon**



Command: step

Label: step_pixmap

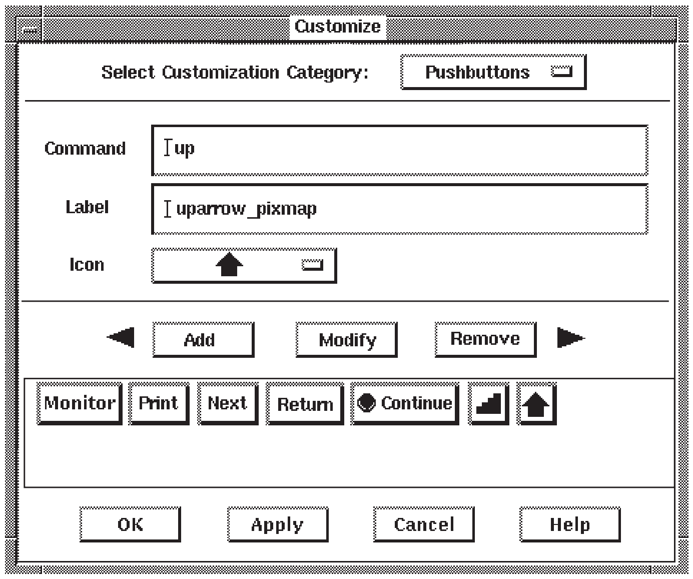
### 4.9.3.2 Adding a New Button and Associated Command

To add a new button to the push-button panel and assign a debugger command to that push button:

1. Choose Options:Customize... on the Main Window. The Customize dialog box appears (see Figure 4–21).
2. Select the Pushbuttons menu option and the Pushbuttons dialog box appears.
3. Enter the debugger command for that push button in the Command field (see Section 4.9.3.1). The command `up` was chosen. This command interprets an entity selected in a window as a zero-terminated ASCII string.
4. Enter a label for that push button in the Label field or choose a predefined icon from the Icon menu. The uparrow_pixmap label was chosen.
5. Click on Add. The push button is added to the control panel within the dialog box.

6. Click on OK. The push button is added to the push-button panel in the Main Window (see Figure 4–21).

**Figure 4–21 Adding a Button for the up Command**



### 4.9.3.3 Removing a Button

To remove a push button:

1. Choose Options:Customize... on the Main Window. The Customize dialog box appears.
2. Select the Pushbuttons menu option and the Pushbuttons dialog box appears.
3. Click on the push button in the control panel of the Customize dialog box.
4. Click on Remove. The push button is removed from the control panel within the dialog box.
5. Click on OK. The push button is removed from the push-button panel in the Main Window.

#### 4.9.3.4 Resequencing a Button

To resequence a button:

1. Choose Options:Customize... on the Main Window. The Customize dialog box appears.
2. Select the Pushbuttons menu option and the Pushbuttons dialog box appears.
3. Click on the button you are resequencing. This fills the Command and Label fields with the parameters for that button.
4. Click on the left or right arrow to move the button one position to the left or right. Continue to click until the button has moved, one position at a time, to its final position.
5. Click on OK to transfer this position to the push-button panel in the Main Window.

### 4.9.4 Customizing the Debugger Resource File

The debugger is installed on your system with a debugger resource file (`ladebugresource`) that defines startup defaults.

When you first choose Options:Save Options on the Main or Optional Views Window, the debugger creates your own local debugger resource file in your home directory.

When you subsequently start the debugger, it uses the settings defined in your local resource file (such as window configuration) and uses the system default resource file for the other settings (such as character fonts). Whenever you choose Save Window Configuration, a new version of your local resource file is created.

Using the system default file as reference, you can add customized resource settings to your own local file. When you subsequently choose Options:Save Options, the debugger automatically copies these added settings to the new version of your local file.

# 5 Advanced Debugging Techniques

Ladebug provides advanced debugging techniques for debugging your multiprocess and multithread applications.

This chapter explains how to:

- Display and select available processes and threads
- Attach and detach a process
- Debug a multithreaded application
- Debug a multiprocess application, including an example that forks a child process and execs a program

For more information about multithreaded application debugging, see Chapter 19. For more information on multiprocess application debugging, see Chapter 20.

## 5.1 Displaying and Selecting Available Processes and Threads

The Source View Context Panel contains menu buttons that list processes and threads. It allows you to change the process or thread context by selecting any process or thread on the menu and bringing it under debugger control.

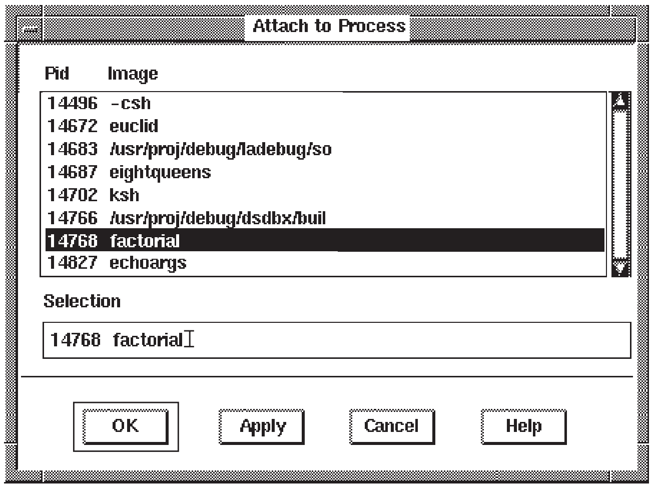
To select a process or thread, pull down the Process or Thread menu button and select the process or thread you want. The contents of the Source View and the Call Stack and any displayed optional views are updated to reflect the current context.

For multithreaded applications, when you select a particular process, the thread option menu button then displays the thread list of the current process.

## 5.2 Attaching and Detaching a Process

To attach to a process not currently under debugger control, choose Command:Attach to Process in the Main Window. This displays a Process Selection dialog box that shows you the list of active processes to which you have access (Figure 5–1).

**Figure 5–1 Process Selection Dialog Box**



```
14496  -csh
14672  euclid
14683  /usr/proj/debug/ladebug/so
14687  eightqueens
14702  ksh
14766  /usr/proj/debug/dsdbx/buil
14768  factorial
14827  echoargs
```

```
14768  factorial
```

Select the process you want to attach to. The Source View reflects the new process. If you have optional views displayed, they will be updated to reflect the new process.

If no breakpoint is set in the process to which you are attaching, the process will run to completion. However, you can set the `$stoponattach` variable to stop the process right after the debugger attaches to it by default. For more information, see Part V, Command Reference.

The Source View will not reflect the new process if the debugger cannot find the path to the actual source file. See Section 4.1.3 for information about specifying the location of source files.

You can also attach to a process using the `load` command at the command prompt. The views are not updated until the `run` command is executed. If a break is not set, the process will run to completion. Use the `stop in` or `stop at` command to set a break. For more information, see Section 9.13.

To detach a process, choose Commands:Detach Process in the Main Window. This displays a dialog box showing all the current processes under debugger control.

Click on the desired process to remove it from debugger control. If the process is current (displayed in the Source View and optional views), the process will run to completion and the views are reinitialized (blanked out). If the process you detach is not current, it will run to completion and the current process will continue to display in the Source View.

The views do not close automatically. To close the views, use the Views menu or exit the debugger.

## 5.3 Debugging a Multithreaded Application

This section describes how to select among several threads in a DECthreads multithreaded application and change the current thread context.

### 5.3.1 Setting the Thread Mode

The debugger thread display defaults to DECthreads if the application is multithreaded and uses DECthreads. Otherwise, the `$threadlevel` is set to `native`.

For more information on identifying and setting the thread mode, see Chapter 19.

### 5.3.2 Steps for Debugging a Multithreaded Application

This section shows how to debug a single process multithreaded application.

The general steps for debugging a multithreaded application are as follows:

1. Start the application as described in Chapter 3.
2. Pull down the Thread menu. The Thread menu displays the current thread on top. If the application contains native threads, the location for each thread displays next to each thread. If the application contains DECthreads, only the thread number displays.
3. Select a different thread by clicking on the thread you want to debug. (The Source View and the Call Stack are updated.)

Ladebug displays the current thread in the Source View.

The Register and Instruction Views show information pertaining to the selected thread. The Breakpoint View does *not* change when the thread is changed because it displays breakpoints for all processes.

## 5.4 Debugging a Multiprocess Application

This section describes how to use the debugger window interface to debug a multiprocess application. It presents an example application that forks a child process and execs a program.

Ladebug provides predefined debugger variables that you can set in order to debug multiprocess applications that fork and/or exec. In the example in Figure 5–2, the debugger variables `$catchforks`, `$catchexecs`, and `$stopparentonfork` have been set from the prompt as follows:

```
(ladebug)set $catchforks=1
(ladebug)set $catchexecs=1
(ladebug)set $stopparentonfork=1
```

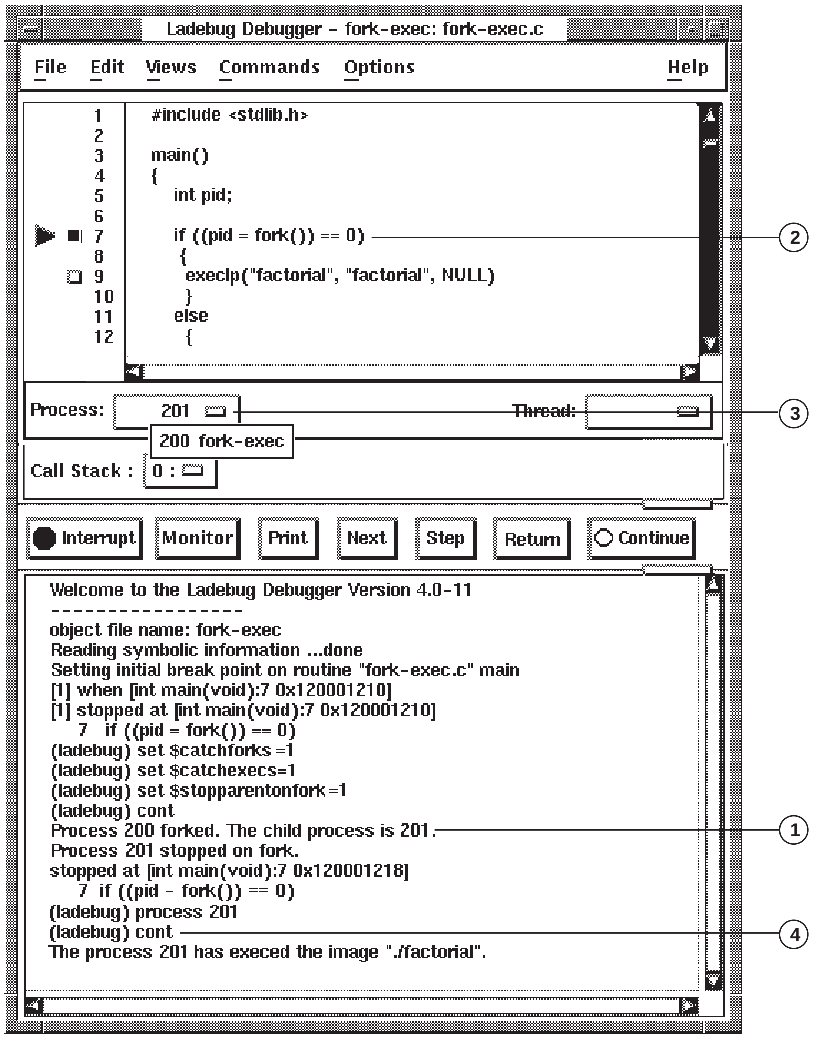
Setting `$catchforks` to 1 instructs Ladebug to notify the user when a program forks a child process. Setting `$catchexecs` to 1 instructs Ladebug to notify the user when a program execs. Setting `$stopparentonfork` to 1 stops the parent process when a program forks a child process.

For more information on debugger variables, see Section 20.7.1. For different scenarios in which these variables are used, see Section 20.7.2 and Section 20.7.3.

The general steps for debugging a multiprocess application are as follows:

1. Start the application as described in Chapter 3.
2. Pull down the Process menu. The Process menu displays the current process on top. The image name for each process displays next to each process.
3. Select a different process by clicking on the process you want to debug. (The Source View and the Call Stack are updated.)

**Figure 5–2 Multiprocess Application**



```
 1   #include <stdlib.h>
 2
 3   main()
 4   {
 5       int pid;
 6
 7       if ((pid = fork()) == 0)
 8        {
 9         execlp("factorial", "factorial", NULL)
10        }
11       else
12        {
```

```
Welcome to the Ladebug Debugger Version 4.0-11
-----------------
object file name: fork-exec
Reading symbolic information ...done
Setting initial break point on routine "fork-exec.c" main
[1] when [int main(void):7 0x120001210]
[1] stopped at [int main(void):7 0x120001210]
     7   if ((pid = fork()) == 0)
(ladebug) set $catchforks =1
(ladebug) set $catchexecs=1
(ladebug) set $stopparentonfork =1
(ladebug) cont
Process 200 forked. The child process is 201.
Process 201 stopped on fork.
stopped at [int main(void):7 0x120001218]
     7  if ((pid - fork()) == 0)
(ladebug) process 201
(ladebug) cont
The process 201 has execed the image "./factorial".
```

**1** In Figure 5–2, a parent process (200) forks a child process (201).

**2** When the parent process forks, the Source View shows that the parent process has stopped.

**3** The child process is selected from the process menu pulldown (process 201).

**4** Entering the `cont` command continues running the child process which execs the program `factorial`.

You can continue debugging the child process or return to the parent process by selecting it in the process menu pulldown. When the process has finished executing, a message displays that process has exited and the Source View reinitializes (blanks out).

# 6
# Using Ladebug Within the DEC FUSE Environment

This chapter introduces you to some of the features and differences of the Ladebug Debugger when run as part of the DEC FUSE environment. FUSE is Digital's software development environment (IDE) for UNIX workstations and servers. It integrates industry-standard UNIX tools with Digital tools and other industry-leading CASE tools.

Features include:

- Support for the compile-edit-debug cycle of software development in an entry-level CASE 3GL tool environment.
- A user interface based on OSF/Motif.
- Support for application development in C, C++, FORTRAN, COBOL, and DEC Ada.
- Integration with the Common Desktop Environment (CDE).

Because FUSE works with source code from any supported language, it is a highly effective tool for maintenance or re-engineering, as well as new development. The software visualization provided by FUSE reduces the time needed to understand complex applications, which results in cost savings and fewer code errors.

For detailed information on FUSE, including prerequisite software and ordering information, see the DEC FUSE Software Product Description (SPD) and the *DEC FUSE Handbook*.

## 6.1 Starting and Configuring the Debugger Within the DEC FUSE Environment

Assuming you have DEC FUSE installed, there are three possible ways you can choose to start the debugger as part of the DEC FUSE environment:

- Use the command line.

  You can start the debugger from the command line using the following command:

  ```
  % fusedebug [ -Xt-Options ] [ filepath ]
  ```

  If DEC FUSE is not running, it starts automatically as a minimized icon.

  See the *DEC FUSE Handbook* and the specific reference pages for descriptions of the command syntax and options.

- If you have CDE installed, directly manipulate the DEC FUSE icons.

  DEC FUSE provides a DEC FUSE application group icon and icons for the individual tools that make up the DEC FUSE environment. To invoke the debugger:

  1. Double click on the Application Manager icon in the CDE Front Panel to display the application group icons.
  2. Double click on the DEC FUSE application group icon to display the icons in the DEC FUSE application group.
  3. Double click on the debugger icon to start the tool.

- Use the DEC FUSE Control Panel once DEC FUSE is running.

  To start the debugger from the DEC FUSE Control Panel (assumes DEC FUSE is already running), select Ladebug Debugger from the Tools menu in the Control Panel.

  DEC FUSE first lists the tool in the Control Panel display area. Then, the tool main window appears.

## 6.2 Differences in the DEC FUSE Debugger Main Window

The DEC FUSE debugger Main Window is essentially the same as it appears in Ladebug with the exception of the following:

- Tools menu

  The Tools menu lets you start the other tools that are integrated within the DEC FUSE environment. You can also use this menu to configure and exit the debugger.

- Help menu differences

  DEC FUSE help is based on Bristol HyperHelp while Ladebug help is based on the OSF/Motif help widget.

- Quick help pane

  The quick help pane provides brief descriptions of menu items at the bottom of the DEC FUSE debugger Main Window.

- File:Exit Debugger was removed from the menu pulldown because you can use Exit Tool from the Tools menu.

## 6.3 Editing Your Program

When you start up the debugger, you specify an executable target. The debugger displays the source file for that target.

The debugger Source View is read-only. To edit a source file, you can access the editor either directly from the debugger or from the Tools menu.

If you want to edit the source file whose code is currently being displayed by the debugger, the quickest way is to choose Edit File from the Commands menu. This invokes your default DEC FUSE editor (FUSE Editor, `Emacs`, or `vi`). The editor displays the target's source code in its own window. As in the debugger Main Window, the source code in the editor window is centered where execution is currently paused.

When you then execute the program with the debugger, the Source View is updated in both the debugger and the editor.

You can use the DEC FUSE Editor and `Emacs` to set breakpoints. For further information, see the *DEC FUSE Handbook*.

# Part II

## Command Interface

This part describes the debugger's command interface, and explains the debugger's basic functions as accessed through the command interface.

# 7 Introduction to the Ladebug Debugger: Command Interface

This chapter introduces the features and functions of the Ladebug command interface. It describes how to:

- Invoke Ladebug and bringing a program under debugger control
- Enter multiple commands on a line
- Use debugger variables and aliases
- Repeat commands
- Execute system commands from the debugger
- Perform command-line editing

This chapter also presents an example debugging session from the command interface.

For an introduction to the Ladebug window interface, see Chapter 2.

## 7.1 Invoking Ladebug and Bringing a Program Under Debugger Control

There are several ways to invoke Ladebug from the command interface and bring your program under debugger control. For example, you can invoke Ladebug from the command interface and specify:

- Core files
- Local and remote kernel files
- Remote applications

Once you have invoked Ladebug, you can bring a process or program under debugger control from the Ladebug prompt by attaching to a process or loading a program. For applications that fork and/or exec, you can also control whether to bring the child process under debugger control.

### 7.1.1 Invoking Ladebug from the Shell

To bring a program or process under debugger control from the shell, choose the appropriate syntax for invoking Ladebug from among the following:

- Invoke Ladebug on a single program:

```
$ ladebug executable_file
```

- Invoke Ladebug on a core file:

```
$ ladebug executable_file core_file
```

- Invoke ladebug and attach to a running process:

```
$ ladebug -pid process_id executable_file
```

- Invoke Ladebug on the local kernel:

```
$ ladebug -k /umunix
```

- Invoke Ladebug on a remote kernel:

```
$ ladebug -remote
```

- Invoke Ladebug on the remote server:

```
$ ladebug -rn node_or_address [,udp_port]
```

### 7.1.2 Bringing a Program Under Debugger Control from the Ladebug Prompt

For information on using the Ladebug prompt from within the window interface, see Section 2.4.

You can bring a program or process under debugger control after invoking Ladebug from the command interface or window interface. From the Ladebug prompt, you can:

- Load a new process:

```
(ladebug) load image_file [core_file]
```

- Attach to a running process:

```
(ladebug) attach process-id image_file
```

For information about attaching to a process, see Section 7.8.9 and Section 9.13. For information about loading a program, see Section 20.4.

## 7.2 Entering Multiple Commands on a Single Line

You can enter several commands on a single line by separating the commands with a semicolon (;). The commands are executed in the order in which you enter them. Example 7–1 shows how to enter multiple commands on a single line.

**Example 7–1 Entering Multiple Commands on a Single Line**

```
(ladebug) stop in main;run
[#1: stop in main ]
[1] stopped at [main:4 0x120000a40]
      4        for (i=1 ; i<3 ; i++) {
(ladebug) where
>0  0x120000a40 in main() sample.c:4
(ladebug)
```

You can enter multiline commands by using a backslash ( \ ) at the end of a line to be continued.

## 7.3 Customizing the Debugger Environment: Debugger Variables

The debugger predefines a set of debugger variables. You can display and modify these variables to alter debugger settings. You can also create new debugger variables to use within other commands, or as placeholders of important information.

All debugger variable names start with a dollar sign ($). The `set` command, used alone, causes the display of all the debugger variables with their current values. The `set` command also lets you set the value of a debugger variable. Using this command, you can redefine an existing debugger variable or create a new debugger variable. The syntax for defining a debugger variable with this command is as follows:

**set** variable = value

If the value of the variable is a text string, enclose the string in quotes. Example 7–2 shows how to use the `set` command to display and redefine debugger variables. In this example, all the predefined variables are displayed by the `set` command. Then the `$lang` and `$historylines` variables are changed and displayed. (The `$lang` variable determines the language syntax and visibility rules the debugger uses; the value for the `$lang` variable is a string enclosed in quotation marks. The `$historylines` variable determines the number of lines listed by the `history` command.)

**Example 7–2 Displaying and Redefining Debugger Variables**

```
(ladebug) set
$ascii = 1
$beep = 1
$catchexecs = 0
$catchforks = 0
$curevent = 0
$curfile = (null)
$curline = 0
$curpc = 0
$cursrcline = 0
$curthread = 0
$decints = 0
$editline = 1
$eventecho = 1
$hasmeta = 0
$hexints = 0
$historylines = 20
$indent = 1
$lang = "C"
$listwindow = 20
$main = "main"
$maxstrlen = 128
$octints = 0
$overloadmenu = 1
$pimode = 0
$prompt = "(ladebug) "
$repeatmode = 1
$stackargs = 1
$stepg0 = 0
$stoponattach = 0
$stopparentonfork = 0
$threadlevel = "decthreads"
$verbose = 0
(ladebug) set $lang = "C++"
(ladebug) set $historylines = 40
(ladebug) print $lang
"C++"
(ladebug) print $historylines
40
(ladebug)
```

For more information on these debugger variables, see Part V, Command Reference.

Use the `unset` command to delete a debugger variable that you created, or to return a debugger variable to its default value. The syntax for the `unset` command is as follows:

**unset** variable

## 7.4 Using Command Abbreviations: Aliases

The debugger lets you use abbreviations for frequently used commands. These abbreviations are called **aliases**. You can list all available aliases by entering `alias` at the debugger prompt. To view the definition of a single alias, enter `alias` followed by the alias name.

To delete an alias, enter `unalias` followed by the alias name.

Several aliases are predefined by the debugger. The predefined aliases substitute one or two letters for whole commands. For example, `l` is the alias for `list` and `q` is the alias for `quit.` A complete list of predefined aliases is in the description of the `alias` command in Part V, Command Reference, and is also displayed by the debugger in response to the `alias` command with no arguments.

You can also create your own aliases. The `alias` command syntax for creating your own alias is as follows:

**alias** aliasname "string"

After you define the alias, entering `aliasname` is identical to entering `string.` Example 7–3 creates an alias that sets a breakpoint, runs your program, and performs a stack trace.

Aliases may also contain parameters. In Example 7–4, the alias defined in Example 7–3 is modified to specify the breakpoint's line number when you enter the abbreviated `alias` command.

Aliases may have multiple parameters. The `alias` command syntax for creating an alias with more than one parameter is as follows:

**alias** aliasname (arg1, arg2, [, . . . ]) "string"

**Example 7–3 Creating an Alias**

```
(ladebug) alias cs
alias cs is not defined
(ladebug) alias cs "stop at 5; run; where"
(ladebug) alias cs
cs      stop at 5; run; where
(ladebug) cs
.oS
[#1: stop at "sample.c":5 ]
[1] stopped at [main:5 0x120000b1c]
      5        f = factorial(i);
>0  0x120000b1c in main() sample.c:5
(ladebug)
```

**Example 7–4 Defining an Alias with a Parameter**

```
(ladebug) alias cs(x) "stop at x; run; where"
(ladebug) alias cs
cs(x)   stop at x; run; where
(ladebug) cs(5)
[#1: stop at "sample.c":5 ]
[1] stopped at [main:5 0x120000b1c]
      5        f = factorial(i);
>0  0x120000b1c in main() sample.c:5
(ladebug)
```

**Example 7–5 Nesting Aliases**

```
(ladebug) alias begin "bp main; run"
(ladebug) alias sp(x,v) ""begin; stop at x; p v""
(ladebug) alias sp
sp(x, v)         begin; stop at x; print v
(ladebug) sp(10,i)
[#4: stop in main ]
[4] stopped at [main:4 0x120001180]
      4    for (i=1 ; i<=3 ; i++) {
0
(ladebug)
```

You can nest aliases. You can define one alias and use that alias in the definition of another alias. In Example 7–5, such an alias is defined and then used in the definition of another alias.

Your definition of an alias can include a quoted string. See the `alias` command in Part V, Command Reference for an example.

## 7.5 Repeating Previously Used Commands: History

The debugger maintains a list of the commands you enter. Using an abbreviated command sequence, you can reenter a command without retyping the entire command (a history feature). Pressing the Return key at the debugger prompt repeats the last command, provided the `$repeatmode` variable is set to 1, which is the default. Entering two exclamation points (!!) at the debugger prompt also repeats the last command (regardless of the setting of the `$repeatmode` variable).

You can examine the list of recently entered commands by entering the `history` command at the debugger prompt. The last command entered is at the bottom of the numbered list. The number of commands listed by the `history` command is determined by the value of the `$historylines` debugger variable.

To enter a command on the list, type an exclamation point followed by the number of the command on the history list. You can also specify a command by indicating how recently the command was last entered; for example, `!-3` reenters the third-to-last command you entered.

You can also reenter a command by entering an exclamation point followed by the beginning of the command string. For example, to reenter the command `deactivate 3`, enter the command `!dea`. Example 7–6 uses the `history` mechanism to reenter commands. You can also use the arrow keys to reenter commands if `$editline` is set to 1, which is the default.

**Example 7–6 Reentering Commands on the History List**

```
(ladebug) stop in main
[#1: stop in main ]
(ladebug) run
[1] stopped at [main:4 0x120001180]
      4     for (i=1 ; i<3 ; i++) {
(ladebug) next
stopped at [main:5 0x120001188]
      5         f = factorial(i);
(ladebug) print i
1
(ladebug) next
stopped at [main:6 0x1200011a0]
      6     printf("%d! = %d\en",i,f);
(ladebug) print f
1
(ladebug) print factorial(f)
1
(ladebug) delete all
(ladebug) stop in factorial
[#2: stop in factorial ]
(ladebug) rerun
[2] stopped at [factorial:13 0x120001224]
     13     if (i<=1)
(ladebug) step
stopped at [factorial:14 0x120001230]
     14         return (1);
(ladebug) Return
stopped at [factorial:17 0x120001264]
     17 }
(ladebug) Return
stopped at [main:5 0x120001194]
      5         f = factorial(i);
(ladebug) print f
0
(ladebug) list $curline - 5: 10
      1 #include <stdio.h>
      2 main() {
      3     int i,f;
      4     for (i=1 ; i<3 ; i++) {
>     5         f = factorial(i);
      6     printf("%d! = %d\en",i,f);
      7     fflush(stdout);
      8     }
      9 }
     10 factorial(i)
(ladebug) cont
1! = 1
[2] stopped at [factorial:13 0x120001224]
     13     if (i<=1)
```

(continued on next page)

**Example 7–6 (Cont.) Reentering Commands on the History List**

```
(ladebug) where
>0  0x120001224 in factorial(i=2) sample.c:13
#1  0x120001194 in main() sample.c:5
(ladebug) cont
[2] stopped at [factorial:13 0x120001224]
     13     if (i<=1)
(ladebug) where
>0  0x120001224 in factorial(i=1) sample.c:13
#1  0x12000124c in factorial(i=2) sample.c:16
#2  0x120001194 in main() sample.c:5
(ladebug) history
10: print f
11: print factorial(f)
12: delete all
13: stop in factorial
14: rerun
15: step
16: step
17: step
18: print f
19: list $curline-5:10
20: cont
21: where
22: cont
23: where
24: history
(ladebug) !12
delete all
```

## 7.6 Executing System Commands from the Debugger

The `sh` command allows you to execute Bourne shell commands without exiting the debugger. The syntax for the `sh` command is as follows:

**sh** command

The *command* argument is a valid operating system command expression.

Do not enclose the command in quotes, even if it consists of multiple words separated by spaces.

After the command finishes, a debugger prompt appears and you can continue with your debugging session.

Example 7–7 uses the shell command `sh` to list information about a file.

**Example 7–7 Executing an Operating System Command**

```
(ladebug) sh ls -l sample.c
-rw-r----- 1 Ladebug       259 May 15 13:08 sample.c
(ladebug)
```

In Example 7–8 the `grep` shell command displays the lines containing `PROGRAM` in the Fortran source file `data2.f90`.

**Example 7–8 Displaying an Identifier Using an Operating System Command**

```
(ladebug) sh grep PROGRAM data2.f90
   PROGRAM DATA
    END PROGRAM DATA
```

You can also `spawn` the Bourne shell from the debugger by issuing:

```
(ladebug) sh sh
```

## 7.7 Using Command-Line Editing

Ladebug supports simple `emacs` style bindings for CTRL keys and arrow keys to edit a command line, as follows:

| | |
|---|---|
| `CTRL-A` | Move to the beginning of the line. |
| `CTRL-E` | Move to the end of the line. |
| `CTRL-D` | Delete a character in place. |
| `CTRL-K` | Kill (cut) from the cursor to the end of line, into the cut buffer. |
| `CTRL-Y` | Yank (paste) from the cut buffer, at the position of the cursor. |
| `CTRL-P` or up arrow | Access items in the history, backward. |
| `CTRL-N` or down arrow | Access items in the history, forward. |
| `CTRL-F` or right arrow | Move the cursor to the right. |
| `CTRL-B` or left arrow | Move the cursor to the left. |

The debugger variable `$editline` enables these key bindings. By default, this variable is set to 1, and they are enabled. (For backward compatibility, you can set `$editline` to 0. The `$editline` is also set to 0 when you use `emacs` with Ladebug.)

The debugger variable `$beep` controls whether a beep sounds when a user tries to perform an illegal action; for example, moving the cursor past the end of a line, or "yanking" from an empty cut buffer. By default, the `$beep` variable is set to 1, enabling the beep to sound.

For more information on using the `set` and `unset` commands with debugger variables, see Section 7.3.

Command-line editing features that are *not* supported in the current release include multiple editing modes (`emacs` and `vi`), multi-line editing, and binding arbitrary keys to actions.

## 7.8 Sample Debugging Session

This section describes the steps necessary to compile and debug a short C program. If you are new to source-level debugging, edit a file called `sample.c`, type in the sample program, and follow the instructions for the sample debugging session.

The sample debugging session shows you how to:

- Compile and execute your program.
- List source code.

- Set a breakpoint.
- Run the program.
- Examine the program state.
- Step through program execution.
- Perform a stack trace.
- Trace a variable.
- Terminate a process.
- Attach to a running process.
- Detach a process.

For information about basic debugging techniques, see Section 1.3. For information about the Ladebug commands used in this sample session, see Part V, Command Reference.

### 7.8.1 Compiling and Executing the Sample Program

The sample program in Example 7–9 uses only C constructs. The program is intended to print the factorials of `1, 2,` and `3` and then exit.

**Example 7–9 Sample C Program, sample.c**

```
#include <stdio.h>
main() {
    int i,f;
    for (i=1 ; i<3 ; i++) {
        f = factorial(i);
    printf("%d! = %d\n",i,f);
    fflush(stdout);
    }
}
factorial(i)
int i;
{
    if (i<=1)
        return (1);
    else
        return (i * factorial(i-1) );
}
```

Example 7–10 demonstrates the steps required to compile, link, and execute the sample program. These instructions assume that the sample program is named `sample.c` and that you are using the C compiler `cc` to build your executable file:

- Use the `-g` option on the compiler command line to instruct the compiler to include in the executable file symbol-table information useful to the debugger.
- Use the `-o` option to instruct the compiler to place the executable image in a file named `sample`.

**Example 7–10 Compiling and Executing the Sample C Program**

```
% cc -g sample.c -o sample
% sample
1! = 1
2! = 2
```

Something is wrong; the program compiles and runs without an error message but does not print the factorial of `3`. You can use the Ladebug debugger to determine the problem.

Example 7–11 demonstrates how to invoke the debugger on the program by using the `ladebug` command. When invoked, the debugger displays a startup banner and some information about the program being debugged. The debugger prompt, `(ladebug)`, is displayed when the debugger is waiting for your next command.

**Example 7–11 Invoking the Debugger on Your Program**

```
(%)  ladebug sample
Welcome to the Ladebug Debugger Version 4.0
------------------
object file name: sample
Reading symbolic information ...done
(ladebug)
```

At this point, you can examine the program being debugged, set breakpoints or tracepoints, or run the program under debugger control.

### 7.8.2 Listing Source Code

To look at the source code lines, enter the `list` command using the following syntax:

**list** line_number

The debugger displays the compiler-generated number for each line in the target program. Many debugger commands (including the `list` command) and messages refer to these line numbers. Example 7–12 shows how to use the `list` command to view the program source file.

**Example 7–12 Listing a Program**

```
(ladebug)  list 1
     1 #include       <stdio.h>
     2 main() {
     3     int i,f;
     4     for (i=1 ; i<3 ; i++) {
     5         f = factorial(i);
     6     printf("%d! = %d\n",i,f);
     7     fflush(stdout);
     8     }
     9 }
    10 factorial(i)
    11 int i;
    12 {
    13     if (i<=1)
    14         return (1);
    15     else
    16         return (i * factorial(i-1) );
    17 }
(ladebug)
```

By default, the debugger lists 20 lines of source code at a time. The `list 1` command in this example specifies that the listing is to begin with the first line of the program. The program is less than 20 lines, so a listing beginning with line 1 displays the entire program.

### 7.8.3 Setting a Breakpoint

If you have a rough idea of where the error is occurring in your program, you can set a breakpoint and run the program under debugger control. A breakpoint set at a line number causes the debugger to suspend program execution each time that line is encountered. To set the breakpoint at a particular line number, enter the `stop at` command using the following syntax:

**stop at** line_number

You can also set a breakpoint in a function so that the debugger suspends program execution when it enters the specified function. Using the following syntax, the `stop in` command lets you set a breakpoint at a function:

**stop in** function_name

According to the sample program listing in Example 7–12, the call to the function `factorial` is on line 5. The breakpoint command in Example 7–13 shows how to set a breakpoint on line 4.

**Example 7–13 Setting a Breakpoint**

```
(ladebug)  stop at 4
[#1: stop at "sample.c":4 ]
(ladebug)
```

The debugger confirms the breakpoint by assigning the breakpoint a reference number and reiterating the breakpoint command. In this example, the reference number is `1`.

## 7.8.4 Running Your Program

Use the `run` or `rerun` command to instruct the debugger to execute your program. During execution, you can examine your program's variables, trace the stack, or step through the program line by line.

When the debugger reaches a breakpoint, the debugger displays the breakpoint that suspended execution and the line of code that will be executed next, if normal program execution is continued. In Example 7–14, program execution is suspended because of the breakpoint. The next line of code that will be executed if line-by-line execution is continued is line 4.

**Example 7–14 Running Your Program Under Debugger Control**

```
(ladebug)  run
[1] stopped at [main:4 0x120001180]
      4    for (i=1 ; i<3 ; i++) {
(ladebug)
```

## 7.8.5 Examining the Program State

Use the `print` command to examine the program state. The syntax for the `print` command is as follows:

**print** expression

The *expression* argument is any expression containing one or more variables, constants, and operators that is valid in the current context. In Example 7–15 a variable is evaluated and the debugger prints the result.

**Example 7–15 Printing a Variable's Value**

```
(ladebug) print i
0
(ladebug)
```

The output shows that the value of the variable *i* is 0.

### 7.8.6 Stepping Through Program Execution

When program execution is suspended, you can continue execution on a line-by-line basis by using the `step` and `next` commands, or you can inspect the program state. After a line executes, the debugger prompt returns. Pressing the Return key repeats the previous command.

The `step` command executes the next line of code in the program. If that line of code calls another function, and if there is debugging information about that function available to the debugger, the `step` command executes the called function line by line. This is called stepping into a function. If debugging information is not available to the debugger, as is the case with the `printf` and `fflush` library routines, the debugger will execute the called function and stop at the next line of code in the function that initiated the call.

The `next` command also executes the next line of code in the program. If that line of code calls another function, the `next` command executes that function and stops at the next line of code in the function that initiated the call. This is called stepping over a function. The `next` command steps *over* called functions; the `step` command steps *into* called functions.

Example 7–16:

- Steps through the function `main`
- Verifies that the value of `i` changes to `1`
- Steps over the called function `factorial`

After the debugger executes the function `factorial`, the debugger returns to the function `main` and steps over the library routines `printf` and `fflush`.

**Example 7–16 Stepping Through Program Execution**

```
(ladebug)  step
stopped at [main:5 0x120000b1c]
      5         f = factorial(i);
(ladebug)  print i
1
(ladebug)  next
stopped at [main:6 0x120000b38]
      6     printf("%d! = %d\n",i,f);
(ladebug)  step
1! = 1
stopped at [main:7 0x120000a7c]
      7     fflush(stdout);
(ladebug)  step
stopped at [main:8 0x120000a48]
      8     }
(ladebug)
```

### 7.8.7 Displaying a Stack Trace

The stack trace lets you follow the dynamic call chain from function to function. The `where` command displays the stack trace. The stack trace displays the most recently called function on the top of the stack. Each function is followed by its calling function, until all active functions are displayed.

Each function on the stack is placed on a separate line, and is associated with a number that corresponds to an activation level relative to the top of the stack. The most recently called function is on level 0, at the top of the stack.

Example 7–17 continues to step through the sample program. As the program computes `2!`, control passes from `main` to `factorial` and back in a predictable fashion. A stack trace taken while stepping through the function `factorial()` for the second time (when `i` equals 2) contains an entry for function `main` and two entries for the recursive function `factorial`.

**Example 7–17 Stepping and Displaying a Stack Trace**

```
(ladebug)  step
stopped at [factorial:13 0x120001224]
     13     if (i<=1)
(ladebug)  Return
stopped at [factorial:16 0x12000123c]
     16         return (i * factorial(i-1) );
(ladebug)  Return
stopped at [factorial:13 0x120001224]
     13     if (i<=1)
(ladebug)  where
>0  0x120001224 in factorial(i=1) sample.c:13
#1  0x12000124c in factorial(i=2) sample.c:16
#2  0x120001194 in main() sample.c:5
(ladebug)  step
stopped at [factorial:14 0x120001230]
     14         return (1);
(ladebug)  Return
stopped at [factorial:17 0x120001264]
     17 }
(ladebug)  Return
stopped at [factorial:16 0x12000123c]
     16         return (i * factorial(i-1) );
(ladebug)  Return
stopped at [factorial:17 0x120001264]
     17 }
(ladebug)  Return
stopped at [main:5 0x120001194]
      5         f = factorial(i);
(ladebug)  Return
stopped at [main:6 0x1200011a0]
      6     printf("%d! = %d\n",i,f);
(ladebug)  Return
2! = 2
stopped at [main:7 0x1200011c0]
      7     fflush(stdout);
```

When you continue stepping through the program, control does not pass back to `factorial()` to compute the factorial of 3 (see Example 7–18). Instead, the next line that is executed is the last line of `main()`. The `cont` command instructs the debugger to resume running the program. The program finishes executing without printing the factorial of 3.

**Example 7–18 Stepping Through the Sample Program**

```
(ladebug)  step
stopped at [main:9 0x120000b84]
      9 }
(ladebug)  cont
Thread has finished executing
(ladebug)
```

The problem in Example 7–9 may lie in the bounds of the `for` construct:

```
for (i=1 ; i<3 ; i++) {
```

Careful examination confirms this hypothesis. When the variable `i` is incremented to `3`, the statement `i <3` is false and a loop exits. To fix the problem, change the statement as follows:

```
for (i=1 ; i<=3 ; i++) {
```

To make the change, edit the `for` construct in the source file and recompile the program.

To edit the source file, choose one of the following:

- If you are using a workstation or terminal that supports multiple windows, you can edit and recompile the program in a different window. When you enter the `run` or `rerun` command, the debugger will use the new program. Previously set breakpoints and traces will still be active.
- Use the `quit` command to leave the debugger then edit the source file, recompile the program, and test the program. If you want to retest the program under debugger control, invoke the debugger again. After you quit the debugger, all breakpoints, tracepoints, and other program-specific settings are lost.

### 7.8.8 Tracing a Variable: the trace Command

Another way to troubleshoot the problem in Example 7–9 is to trace the value of variable `i` as the program executes. If you have not yet exited the debugger, the breakpoint at line 4 is still active. You can enter the `delete` command to delete the breakpoint at line 4. In Example 7–19, the `status` command is used to list the breakpoints or tracepoints that are currently active and the `delete` command is used to remove the breakpoint at line 4.

**Example 7–19 Deleting a Breakpoint**

```
(ladebug)  status
#1 PC==0x120001180 in main "sample.c":4 { break }
(ladebug)  delete 1
(ladebug)  status
(ladebug)
```

You can use the `trace` command to monitor program variables and to monitor when functions are entered and exited. With this command, you can set a tracepoint on a variable that is visible from the current context at the time you set the breakpoint. The syntax is as follows:

**trace** variable

When you run your program under debugger control, the debugger will print a message when the variable changes value. The `trace` command works at the function level; when a traced variable is evaluated, and the subsequent message is printed when program execution *begins each function*, not at the line of code that caused the variable value to change.

In Example 7–20:

- Set a breakpoint at the beginning of the function `main`
- Execute the program under debugger control until the beginning of the function `main` using the `run` command
- Set the desired tracepoint using the `trace` command and resume program execution using the `cont` command

**Example 7–20 Tracing a Program Variable**

```
(ladebug)  stop in main
[#2: stop in main ]
(ladebug)  run
[2] stopped at [main:4 0x120000a40]
      4     for (i=1 ; i<3 ; i++) {
(ladebug)  trace i
[#3: trace i ]
(ladebug)  cont
[3] Value of i changed before "sample.c":13
        Old value = 0
        New value = 1
1! = 1
[3] Value of i changed before "sample.c":13
        Old value = 1
        New value = 2
2! = 2
[3] Value of i changed before "../exit.c":12
        Old value = 2
        New value = 3
Thread has finished executing
(ladebug) "
```

Tracing variable `i` reveals that the loop did not execute when `i` was equal to `3`. This was the same conclusion reached earlier by stepping through program execution.

In this example, the `trace` command works well, even though traces are computationally intensive and can dramatically slow down program execution. For this reason, using traces may not be appropriate in every situation. Each debugging problem is different, and you may need to try several different methods on a program before you uncover the problem.

### 7.8.9 Attaching to a Running Process

Ladebug allows you to attach to a running process that is not under debugger control and debug the process.

Use the `attach` command to connect to a running process by specifying the process ID and the associated image file. Ladebug allows you to switch to debugging another process in the same session by attaching to it or by loading it (see Section 20.4 for information about loading a program). After you attach to a running process, you debug the process as you would any process that is loaded by the debugger. For common user scenarios for debugging attached processes, see Section 9.13.

You cannot issue the `run` or `rerun` command on an attached process.

There are two ways to attach to a process. From the command line, the syntax is as follows:

$ **ladebug -pid** process_id

From the Ladebug prompt:

(ladebug) **attach** process_id image_file

The sample program in Example 7–9 has been modified slightly to show how to attach to (and detach from) a running process.

```
#include <stdio.h>
#include <unistd.h>
main() {
        int i,f;
        sleep (20);
        for (i=1; i<3; i++) {
            f = factorial(i);
            printf("%d! = %d\n",i,f);
            fflush(stdout);
        }
}
factorial(i)
int i;
{
        if (i<=1)
            return (1);
        else
            return (i * factorial(i-1) );
}
```

Ladebug does not stop the running process when attaching to it. You must type Ctrl/C to stop the process. You can also set the debugger variable `$stoponattach` to 1 to cause Ladebug to stop the attached process after attaching to it.

Example 7–21 shows how to attach to a process in a debugging session.

**Example 7–21 Attaching to a Running Process**

```
$ factorial &
[1]     32625 1
$ ladebug -pid 32625 factorial 2
Welcome to the Ladebug Debugger Version 4.0-10
------------------
object file name: factorial
Reading symbolic information ...done 3

Attached to process id 32625  .... 4

^CThread received signal INT 5
stopped at [<opaque> __usleep_thread(): ??? 0x3ff800ec850]

(ladebug) stop in factorial 6
[#1: stop in int factorial(int) ]

(ladebug) detach 7
```

**1** Program "factorial" is running on process 32625.

**2** Invoke Ladebug and ask to attach to process 32625.

**3** Ladebug reads the process symbol table.

**4** Ladebug is now attached to the running process. Ladebug will not stop until the user hits Ctrl/C.

**5** Upon Ctrl/C, the process stops. Ladebug shows where the program has stopped.

**6** Ladebug detaches from the process and lets it run freely again. All breakpoints are removed from the user program.

## 7.8.10 Detaching from a Process

The `detach` command lets you detach the debugger from the previously attached process, based on the process ID you specify from the process ID list. (See Example 7–21 for an example of attaching and detaching a process.)

The syntax for the `detach` command is as follows:

**detach** [process_id]

The `process_id` parameter indicates the process to which the debugger is attached. If you do not specify the `process_id` parameter, Ladebug detaches from the current process.

Detaching from a process removes the process information from the debugger, and disables your ability to debug the process.

Ladebug removes all user-specified breakpoints from the detached process. The process continues its program execution.

If other processes are attached to the process being debugged, the `detach` command will not change the process's state.

## 7.8.11 Terminating Processes

The `kill` command terminates the process that executes the program. If the process you terminate has child processes associated with it, they are terminated also.

When a process is terminated, its process objects are not deleted and Ladebug retains the symbolic debugging information. You can issue `run` and `rerun` commands on that application again.

Use the `kill` command to terminate a one or more processes. The syntax of the `kill` command does not take an argument.

The `quit` command will kill a process that was started by Ladebug.

While displaying a process, you can stop it at any time by typing Ctrl/C.

# 8 Examining Program Information

This chapter describes the methods for examining information in your program with the Ladebug debugger command interface.

## 8.1 Listing Source Code: the list Command

The `list` command displays source-code lines starting with one of the following:

- The source line corresponding to the position of the program counter
- The last line listed if multiple `list` commands are entered
- The line number specified as the first argument to the `list` command

When the `list` command is entered without an argument, the number of lines listed is determined by the value of the `$listwindow` debugger environment variable. The default value of the `$listwindow` variable is 20. Entering `list` (or pressing Return) lists the next 20 lines of code. Repeated `list` commands display your program's source code in 20-line segments.

To display a range of lines, enter a `list` command with the following syntax:

**list** begin_line_number, end_line_number

The *begin_line_number* argument specifies the number associated with the starting source-code line. The *end_line_number* argument specifies the number associated with the last line of source code you want listed.

Example 8–1 uses the `list` command to display the lines of a COBOL program source file numbered 43 through 50.

**Example 8–1 Listing Source Code in a Number Range**

```
(ladebug) list 43,50
     43     PERFORM LOOK-BACK  VARYING SUB-1 FROM 20 BY -1
     44         UNTIL TEMP-CHAR (SUB-1) NOT = SPACE.
     45     MOVE SUB-1 TO CHARCT.
     46     PERFORM MOVE-IT    VARYING SUB-2 FROM 1 BY 1   UNTIL SUB-1 = 0.
     47     MOVE HOLD-WORD TO TEMP-WORD.
     48 MOVE-IT.
     49     MOVE TEMP-CHAR (SUB-1) TO HOLD-CHAR (SUB-2).
     50     SUBTRACT 1 FROM SUB-1.
(ladebug)
```

To display a specific number of lines, beginning with a particular line, enter the `list` command with the following syntax:

**list** begin_line_number:count

The *count* argument specifies the number of lines to list.

Example 8–2 uses the `list` command to display the first 25 lines of an Ada program.

**Example 8–2 Listing Source Code By Counting from a Starting Line**

```
(ladebug) list 1:25
     1  with TEXT_IO; use TEXT_IO;
     2  with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
     3  with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
     4  procedure TEST is
     5
     6     procedure P (I : INTEGER);
     7     procedure P (F : FLOAT);
     8
     9     procedure P (I : INTEGER) is
    10        TWO : constant := 2;
    11     begin
    12        PUT (I * TWO);
    13        NEW_LINE;
    14     end;
    15
    16     procedure P (F : FLOAT) is
    17     begin
    18        PUT(F);
    19        NEW_LINE;
    20     end;
```

(continued on next page)

**Example 8–2 (Cont.) Listing Source Code By Counting from a Starting Line**

```
    21
    22  begin
    23     P(5);
    24     P(6.0);
    25  end;
(ladebug)
```

To display the source code for a particular function, enter the name of the function as an argument to the `list` command. The syntax is as follows:

**list** function

The debugger prints an error message if it cannot find the source file corresponding to the current function.

If the source file is not in the current directory, you can enter a `use` command to add directories to the list of directories the debugger will search. The syntax is as follows:

**use** [directory]

You can enter multiple `use` commands to ensure that the debugger finds all source files associated with your program.

The `unuse` command sets the search list to any of the following:

- The default
- The home directory
- The current directory
- The directory containing the executable file

Include the name of a directory to remove it from the search list or an asterisk (*) to remove all directories from the search list. The syntax is as follows:

**unuse** [directory]

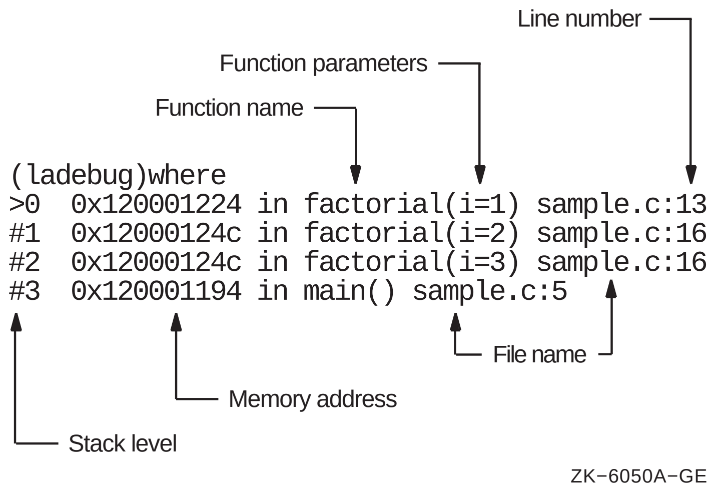
**unuse ***

You can also specify search directories as command-line arguments to the `ladebug` command with the `-I` option. (See the `ladebug(1)` reference page for more information about the options to the `ladebug` command.)

## 8.2 Displaying a Stack Trace: the where Command

As explained in Chapter 7, the stack trace lists the active functions in the program you are debugging. Figure 8–1 shows the elements of a stack trace that the debugger displays when you enter a `where` command.

**Figure 8–1 Sample Stack Trace**



```
(ladebug)where
>0  0x120001224 in factorial(i=1) sample.c:13
#1  0x12000124c in factorial(i=2) sample.c:16
#2  0x12000124c in factorial(i=3) sample.c:16
#3  0x120001194 in main() sample.c:5
```

The stack trace provides the following information for each activation level:

| | |
|---|---|
| **Stack level** | The number used to refer to an activation level on the stack. The function entered most recently is on activation level 0. |
| **Function name** | The name of the function active at this activation level of the stack. The values of any variables passed to this function are also listed. |
| **Function parameters** | The list of parameter names and values from previous function calls. |
| **Memory address** | The address of the next instruction to be executed in the named function. |
| **File name** | The name of the file containing the source code for the function. |
| **Line number** | The number of the last executed source line in the function. |

Example 8–3 uses the `where` command to display the stack trace of a COBOL program.

**Example 8–3 Displaying the Stack Trace in a COBOL Program**

```
(ladebug) where
>0  0x3ff81808744 in cob_acc_display() cob_accdis.c:349
#1  0x120001fbc in testa() testa.cob:20
#2  0x3ff8181f054 in main() cob_main.c:253
```

## 8.3 The Current Context

The debugger recognizes the scoping rules of the symbols in the target program by maintaining a current context and accessing program symbols based on this context. The context is determined by:

- The file scope
- The function scope
- The class scope

And, if applicable, by:

- The C++ class scope
- The process scope
- The thread scope

The context is automatically set to the current point of execution when the debugger is given control (for example, when the debugger stops at a breakpoint).

The debugger extends symbol accessibility beyond that allowed by the language rules when you change the current context or qualify a symbol name with full scope information.

If you do not qualify the symbol name with full scope information, the debugger looks for the symbol in the current context (including process context or thread context, if applicable). The current context is determined by the point at which execution is paused or by the context set when you change the function, file, or class scope.

If the symbol is not found, the debugger searches the calling function, and then its caller, and so on. Within each context, the debugger uses the visibility rules of the language to locate a symbol. If the symbol is still not found, the debugger searches the global symbol table.

### 8.3.1 The Current Function Scope: the func, up, and down Commands

When the debugger encounters a breakpoint and suspends program execution, the current function is at activation level 0. The stack trace marks the current function scope with an angle bracket (>). All other levels on the stack are marked with a pound sign (#). If you want to examine a function other than the current function, use the `func`, `up`, or `down` command to change the current function scope.

Changing the current function scope does not change the current point of program execution. When you continue program execution, the program begins where it left off, regardless of the current function scope.

Change the function scope to examine the value of program symbols or data structures that are not visible in the current scope.

The `func` command lets you specify a new current function scope by function name. In Example 8–4 the current function scope is changed to function `main` so that a variable in `main()` can be displayed.

**Example 8–4 Using the func Command**

```
(ladebug) stop at 13
[#1: stop at "sample.c":13 ]
(ladebug) run
[1] stopped at [factorial:13 0x120000ad4]
     13     if (i<=1)
(ladebug) cont
1! = 1
[1] stopped at [factorial:13 0x120000ad4]
     13     if (i<=1)
(ladebug) print f
Symbol f not visible in current scope.
Error: no value for f
(ladebug) where
>0  0x120000ad4 in factorial(i=2) sample.c:13
#1  0x120000a54 in main() sample.c:5
(ladebug) func main
main in sample.c line No. 5:
      5         f = factorial(i);
```

```
(ladebug) where
#0  0x120000ad4 in factorial(i=2) sample.c:13
>1  0x120000a54 in main() sample.c:5
(ladebug) print f
1
(ladebug)
```

The `up` and `down` commands let you specify a new current function scope by moving up or down a specified number of levels on the stack trace. If you enter the `up` or `down` commands without an argument, you move the current function pointer up or down one level.

In the previous example, instead of entering `func main` you can enter `up 1`. Note that the final stack trace in this example lists function `main` as the current function scope (denoted by the > character).

## 8.3.2 The Current File Scope: the file Command

The current file scope is automatically set to the name of the file containing the source code of the function you are debugging. When the function scope changes, the current file scope is updated automatically. Use the `file` command to display and change the current file scope. You can enter this command with no arguments to display the current file scope.

To change the file scope, enter the `file` command with the following syntax:

**file** filename

Enter the name of the file exactly as it was entered on the compile command line. Use a preceding directory path if necessary.

Changing the current file scope lets you list a function in the new file or set a breakpoint on a line in the function. After loading the file with the `file` command, you can enter the `list` command to examine it.

Example 8–5 uses the `file` command to set the debugger file scope back to the main COBOL program, and then stops at line number 20 in that file.

**Example 8–5 Using the file Command**

```
(ladebug) file testa.cob
(ladebug) stop at 20
[#6: stop at "testa.cob":20]
```

When you change the current file scope, the function scope is reset. Running or continuing your program nullifies the `file` command and reloads the source code corresponding to the program you are debugging.

### 8.3.3 The Current Language Context: the $lang Debugger Variable

The debugger automatically identifies the language of the current function or code segment based on information embedded in the executable file. For example, if program execution is suspended in a C function, the current language is C. If the program executes a C++ function, the current language becomes C++. The current language determines the valid expression syntax for the debugger.

The debugger sets the `$lang` variable to the language of the current function or code segment. If the program is not running, `$lang` is set to the language of the module in which the main entry point resides.

By manually setting the `$lang` debugger variable, you can force the debugger to interpret expressions used in commands by the rules and semantics of a particular language. For example, you check the current setting of `$lang` and change it as follows:

```
(ladebug) print $lang
"C++"
(ladebug) set $lang = "C"
```

## 8.4 Examining and Modifying Program Symbols

You can access program symbols when program execution is suspended in the function that defines the symbol or in any function called by the defining function. Before you can access a symbol, you must execute the program beyond the point where the symbol is declared. If you examine an expression before the variables used in the expression are initialized, the expression value may not be valid.

### 8.4.1 Evaluating Expressions: the print and whatis Commands

Use the `print` command to display the value of specific variables or expressions in functions active on the stack. You can also use the `print` command to evaluate complex expressions involving typecasts, pointer dereferences, multiple variables, constants, and any legal operators allowed by the language of the program you are debugging.

You cannot print a program symbol until you have started executing the program using the `run` or `rerun` command. You can use the `set` command to view the values of debugger variables before starting execution.

The syntax for the `print` command is as follows:

**print** *expression*

The debugger formats expression values according to the type defined for them in the program. You can print an expression's type using the `whatis` command. The syntax for the `whatis` command is as follows:

**whatis** expression

Example 8–6 uses the `whatis` command to examine the contexts of the data item `SUB-1` in the COBOL program `TESTA`.

**Example 8–6 Examining Data Items in a COBOL program**

```
(ladebug) whatis sub-1
 unsigned short SUB-1
(ladebug) print sub-1
0
(ladebug)
```

Consider the following declarations in an Ada program:

```
type DAY is (MON,TUES,WED,THURS,FRI,SAT,SUN);
MY_DAY : DAY := MON;
```

Example 8–7 uses the `whatis` command to determine the storage representation for the variable `MY_DAY`.

**Example 8–7 Determining the Type of a Variable**

```
(ladebug) whatis MY_DAY
enum DAY { MON, TUES, WED, THURS, FRI, SAT, SUN }  MY_DAY;
(ladebug) print MY_DAY
MON
```

For an array, the debugger prints every cell in the array if you do not specify a specific cell.

Consider the following declarations in an Ada program:

```
type CAR is (BUICK,FORD,HONDA);
type CAR_ARRAY is array (CAR) of INTEGER;

CAR_NUM : CAR_ARRAY := (1,2,3);
```

Example 8–8 uses the `print` command to display a nonstring array.

**Example 8–8 Printing Values of an Array**

```
(ladebug) print CAR_NUM
[0] = 1,[1] = 2,[2] = 3
```

Example 8–9 shows how to print individual values of an array.

**Example 8–9 Printing Individual Values of an Array**

```
(ladebug) list 1, 4
      1 main() {
      2    int c[5],d;
      3    for (d=0 ; d<=5 ; d++) {
      4        c[d] = d;
(ladebug) stop in main
[#1: stop in main ]
(ladebug) run
[1] stopped at [main:3 0x1200010b4]
      3    for (d=0 ; d<=5 ; d++) {
(ladebug) whatis c
array [subrange 0 ... 4 of int] of int c
(ladebug) print c
[0] = 0,[1] = 0,[2] = -2147474264,[3] = 1023,[4] = 0
(ladebug) step
stopped at [main:4 0x120001068]
      4              c[d]=d;
(ladebug) Return
stopped at [main:6 0x1200010dc]
      6 }
(ladebug) print c
[0] = 0,[1] = 1,[2] = 2,[3] = 3,[4] = 4
(ladebug) print c[4]
4
(ladebug)
```

Expressions containing labels are not supported. Variables involving static anonymous unions and enumerated types cannot be printed. Printing a structure that is declared but not defined in a compilation unit generates an error message indicating that the structure is opaque.

## 8.4.2 Dereferencing Pointers: the * Operator

Variables containing addresses are called pointers. By dereferencing a pointer, you can print the value at the address pointed to by the pointer. In C programs, variables containing a pointer are dereferenced using the * operator. Example 8–10 shows how to dereference a pointer in C programs.

**Example 8–10 Dereferencing a Pointer**

```
(ladebug) list 1
      1 main()
      2 {
      3     int x, *c;
      4
      5     c = &x;
      6     x = 2;
      7     *c = 0;
      8 }
(ladebug) stop in main
[#1: stop in main ]
(ladebug) run
[1] stopped at [main:5 0x12000045c]
      5     c = &x;
(ladebug) whatis c
int * c

(ladebug) print c
0x11ffffe4c
(ladebug) print *c
2
(ladebug)
```

## 8.4.3 Listing Variables: the dump Command

The `dump` command lists an active function's local variables and parameters. You can specify a dump of a specific function by naming the function as an argument to the command. You can show all the variables and parameters in all currently active functions by entering a period after the `dump` command. In Example 8–11, you obtain a dump of all active functions from the sample program in Example 7–9.

**Example 8–11 Displaying Information on Each Activation Level**

```
(ladebug) dump
>0  0x120001224 in factorial(i=1) sample.c:13
i=1
f=0
(ladebug)
```

The function `factorial` has the parameter `i` but it has no local variables. The function `main` has two local variables, `i` and `f`.

### 8.4.4 Displaying a Variable's Scope: the which and whereis Commands

Your program may declare a particular symbol more than once. For example, in the program `sample.c` the symbol `i` is declared in the function `main` and again in the function `factorial`.

Because the debugger extends the visibility and scope rules of the programming language, the debugger may have access to more than one declaration of a symbol name. You can use the `which` command to determine which declaration exists in the current context. The `which` command displays the variable with its scope information fully qualified.

The `whereis` command lists all declarations of a variable together with each declaration's scope information fully qualified. Example 8–12 shows how to use the `whereis` and `which` commands to determine a variable's scope.

**Example 8–12 Displaying a Variable's Scope**

```
(ladebug) where
>0  0x120001230 in factorial(i=1) sample.c:14
#1  0x12000124c in factorial(i=2) sample.c:16
#2  0x120001194 in main() sample.c:5
(ladebug) print i
1
(ladebug) which i
"sample.c"‘factorial.i
(ladebug) whereis i
"sample.c"‘factorial.i
"sample.c"‘main.i
(ladebug) func main
main in sample.c line No. 5:
      5         f = factorial(i);
```

(continued on next page)

**Example 8–12 (Cont.) Displaying a Variable's Scope**

```
(ladebug) print i
2
(ladebug) which i
"sample.c"‘main.i
(ladebug)
```

The `whereis` command is useful for obtaining information needed to differentiate overloaded identifiers that are in different units, or within different routines in the same unit.

Example 8–13 shows how to set breakpoints in two Ada program routines, both named `DO_PRINT`.

**Example 8–13 Determining Overloaded Identifiers**

```
(ladebug) whereis do_print
"overload.ada"‘OVERLOAD‘DO_PRINT
"overload.ada"‘OVERLOAD‘INNER‘DO_PRINT
(ladebug) stop in "overload.ada"‘OVERLOAD‘DO_PRINT
[#1: stop in DO_PRINT ]
(ladebug) stop in "overload.ada"‘OVERLOAD‘INNER‘DO_PRINT
[#2: stop in DO_PRINT ]
(ladebug)
```

## 8.4.5 Changing the Value of an Expression: the assign and patch Commands

Use the `assign` command to change the value associated with an expression in the debugger session. The new value is associated with the variable until the value is changed by the program during execution, or until you specifically change the value. The syntax for the `assign` command is as follows:

**assign** expression = expression

The *expression* argument can be any expression that is valid in the current context and language.

Example 8–14 shows how to deposit the value `-42` into the data item `SUB-2` of a COBOL program named `TESTA`.

**Example 8–14 Depositing a Value in an Expression**

```
(ladebug) assign sub-2=-42
(ladebug) print sub-2
-42
```

Example 8–15 shows how to change the value associated with a variable and the value associated with an expression.

**Example 8–15 Assigning Values to a Variable and an Expression**

```
(ladebug) assign i = 5
(ladebug) print i
5
(ladebug) assign f = factorial(8)   1
(ladebug) print f
40320
(ladebug) assign i = f
(ladebug) print i
40320
(ladebug)
```

**1** This command calls the function `factorial`.

Use the `patch` command to change the value associated with an expression in the object file on the disk as well as the debugger process you are running. The syntax is as follows:

**patch** expression1 = expression2

The *expression* argument can be any expression that is valid in the current context and language.

The `patch` command patches executable disk files to correct bad data or instructions. The text, initialized data, or read-only data areas can be patched. The `bss` segment cannot be patched because it does not exist in disk files.

For more information, see Chapter 22 and Part V, Command Reference.

# 9 Controlling Program Execution

This chapter descibes how to use Ladebug from the command interface to execute a program under debugger control and debug the program. Your source language may be C, C++, COBOL, Ada, or Fortran. Your program may have one or more processes or threads.

This chapter describes how to:

- Run a program using command-line arguments; terminate the program.
- Resume program execution after reaching a breakpoint.
- Resume program execution until a particular function finishes.
- Step through program execution one source instruction at a time.
- Execute program functions from the debugger prompt.
- Use the `pop` command.
- Control the debugging of attached processes.
- Debug programs with stripped images.
- Use environment variables.

This chapter describes basic program control and debugging of single-process applications. For more information on debugging multiprocess applications, see Chapter 20. For information on debugging multithreaded applications, see Chapter 19.

## 9.1 Starting Program Execution: the run and rerun Commands

Use the `run` and `rerun` commands to start execution of a program under debugger control. Ladebug picks up any new image on the disk and reloads the latest symbolic debugging information.

You can run the program with command-line arguments by entering them immediately after the `run` command. Command-line arguments can include flags and options recognized by the program as well as input and output redirections.

The `rerun` command runs the program with the arguments last used with the `run` command. Example 9–1 shows how to use the `run` and `rerun` commands.

**Example 9–1 Using run and rerun to Begin Program Execution**

```
(ladebug)  run -s > prog.output
Thread has finished executing
(ladebug)  stop in main
[#1: stop in main ]
(ladebug)  rerun
[1] stopped at [main:4 0x1200011c0]
      4         for (i=1 ; i<3 ; i++) {
(ladebug)
```

If you enter any command-line arguments with the `rerun` command, the debugger discards the previous set of arguments and uses the new arguments.

## 9.2 Terminating Program Execution: the kill Command

You can terminate program execution by using the `quit` or `kill` commands. The `quit` command terminates both the debugger and the debugged process and returns you to the shell.

The `kill` command terminates the debugged process but leaves the debugger running. Any breakpoints and traces previously set will still be set. You can rerun the program after it has been killed.

## 9.3 Stepping Through Functions

After program execution suspends at a breakpoint or due to a signal, you can direct the debugger to execute only the next line or to continue execution until the current function or a named function exits.

The `step` and `next` commands let you execute your source code line by line in the debugger, which gives you the opportunity to examine variables and data structures. If the source-code line contains a function call, and the debugging information about the function is available to the debugger, the `step` and `next` commands behave differently.

### 9.3.1 The step Command

The `step` command steps through your program one line at a time following the flow of control exactly. It steps into the function call and returns to the debugger prompt with the program counter pointing at the first line in the function that was called.

In Example 9–2, two `step` commands continue executing a Fortran program into lines 10 and 11 after program execution has paused at line 9.

**Example 9–2 Stepping Through Program Execution**

```
(ladebug) stop at 9
[#2: stop at "squares.f90":9 ]
(ladebug) run
[2] stopped at [squares_:4 0x1200016dc]
      9      DO 10 I = 1, N
(ladebug) step
stopped at [squares_:10 0x120001710]
     10       IF(INARR(I) .NE. 0)  THEN
(ladebug) step
stopped at [squares_:11 0x12000172c]
     11       OUTARR(K) =   INARR(I)**2
```

### 9.3.2 The next Command

The `next` command executes the called function to completion and returns to the debugger prompt with the program counter pointing at the source-code line immediately after the line containing the function call.

If the debugging information for the function is not available to the debugger, then the debugger will step over the function regardless of whether the `step` or the `next` command was used.

When you call a routine with limited symbolic information, you may not want to step into it. The debugger is under the control of the debugger variable `$stepg0`, which determines whether Ladebug will step into or bypass the routine. When `$stepg0` is set to 0 (the default), the debugger steps over calls to routines compiled without the option that includes full symbolic debugging information. This means the debugger behaves as if a `next` command were entered, instead of a `step` command.

Setting `$stepg0` to 1 causes the debugger to step into these calls, rather than over them.

## 9.4 Resuming Program Execution

Use the `cont` command to resume program execution that has been suspended by a breakpoint or signal. Execution then continues until the next breakpoint or signal, or until the end of the program.

By entering a signal parameter value with the `cont` command, you can direct the debugger to send a signal to the program resuming program execution. This feature allows you to test your program's signal handling characteristics.

The signal parameter values can be either a signal number or a string name (for example, SIGSEGV). If you do not specify a parameter value, the default is 0, which allows the program to continue execution without the signal.

In Example 9–3, a `cont` command resumes program execution after program execution is suspended by a breakpoint.

**Example 9–3 Continuing Program Execution**

```
(ladebug)  stop in main
[#1: stop in main ]
(ladebug)  run
[1] stopped at [main:4 0x120000b14]
      4     for (i=1 ; i<3 ; i++) {
(ladebug)  cont
1! = 1
2! = 2
Thread has finished executing
(ladebug)
```

## 9.5 Branching to a Specified Line: the goto Command

Use the `goto` command to branch to a specified line after execution is suspended. The source code between the line at which execution suspended and the line you specify is not executed.

To branch to a source code line, use the following syntax:

**goto** line_number

The *line_number* argument must be a line of source code located in the same function in which execution is suspended.

Example 9–4 shows an example that uses the `goto` command.

**Example 9–4 Branching to a Specified Line**

```
(ladebug)  list 1:9
      1 #include <stdio.h>
      2 main() {
      3     int i,f;
>     4     for (i=1 ; i<3 ; i++) {
      5         f = factorial(i);
      6     printf("%d! = %d\n",i,f);
      7     fflush(stdout);
      8     }
      9 }
(ladebug)  goto 6
(ladebug)  step
stopped at [main:7 0x1200011c0]
      7     fflush(stdout);
(ladebug)
```

## 9.6 Setting Breakpoints

The debugger lets you place **breakpoints** in your program. When program execution reaches one of these breakpoints, the debugger can either perform predefined actions and continue program execution or suspend program execution and return control to you. Breakpoints can:

- Suspend program execution (defined using the `stop` and `stopi` commands)
- Execute a set of debugger commands (defined using the `when` and `wheni` commands)

**Note**

The `stopi` and `wheni` commands are specifically used for machine-level debugging. As such, breakpoints are set based on machine instruction addresses rather than line numbers. For more information on machine-level debugging, see Chapter 18.

Any breakpoints you define remain active until you exit the debugger, disable them using the `disable` command, or delete them using the `delete` command.

Table 9–1 lists the commands used for setting breakpoints.

**Table 9–1 Commands for Setting Breakpoints**

| Command | Description |
|---|---|
| `stop` | Without a variable argument, suspends program execution and returns to the prompt. With a variable argument, suspends program execution when a variable changes. |
| `stop if` | Suspends execution when an expression evaluates to true. |
| `stop at` | Suspends execution when a specific line number is encountered. |
| `stopi`, `stopi if`, `stopi at` | For machine-level debugging; suspends program execution when the specified variable value changes, when an expression evaluates to true, or when a specified address is encountered. |
| `stop in` | For C++ programming; see the Command Reference for specific forms of this command. |

## 9.6.1 Breakpoints That Suspend Program Execution

Use the `stop` and `stopi` commands to set a breakpoint that suspends program execution. When specified conditions are met, the debugger:

- Suspends program execution
- Prints the first line of source code that will be executed when program execution continues
- Issues the debugger prompt

You can then use debugger commands to examine the program state, change program variable values, and continue program execution from the breakpoint.

To determine if program execution should halt, you can base the breakpoint on one or more of the following conditions:

- When program execution reaches an instruction corresponding to a particular line in the source code
- When program execution reaches a memory address
- When program execution reaches a particular function
- When a program variable changes value
- When a set of conditions you specify evaluate to true

To set a breakpoint that suspends program execution when an instruction corresponding to a line in the source code is reached, use the appropriate `stop at` command syntax.

#### 9.6.1.1 The stop at Command

**stop at** line_number

**stop at** "file_name":line_number

The *line_number* argument specifies the line number in the source code of the current source file.

If the source code for your program spans multiple files, make sure that the file context is set to the correct file before you set your breakpoint. For example:

**file** file_name

You must use quotation marks around the *file_name* argument as shown in the second form. For example, if the file name is `sample.c` and the line number is 4, you type

```
(ladebug) stop at "sample.c":4
```

After you enter a breakpoint command, the debugger confirms the breakpoint by displaying it and its reference number. Example 9–5 sets a breakpoint at line number 13 of the current source file.

**Example 9–5 Setting a Breakpoint at a Line in C Source Code**

```
(ladebug) stop at 13
[#1: stop at "sample.c":13 ]
(ladebug) run
[1] stopped at [factorial:13 0x120001224]
     13         if (i<=1)
(ladebug)
```

To set a breakpoint that suspends program execution when a memory address is reached, use the `stopi at` command.

#### 9.6.1.2 The stopi at Command

**stopi at** address

When program execution reaches the specified address, execution is suspended and the debugger prints the source instruction corresponding to the specified address. To specify the address in hexadecimal format, add the prefix `0x` to the hexadecimal number. The debugger converts the address to decimal when it confirms the breakpoint command. Example 9–6 sets a breakpoint at the address of a particular line in the sample program.

**Example 9–6 Setting a Breakpoint at an Address in the Source Code**

```
(ladebug) stopi at 0x120000b14
[#1: stopi at 4831841044 ]
(ladebug) run
[1] stopped at [main:4 0x120000b14]
      4     for (i=1 ; i<3 ; i++) {
(ladebug)
```

To set a breakpoint that suspends execution at the first instruction in a program function, procedure, or statement, use the (`stop in` command.

#### 9.6.1.3 The stop in Command

**stop in** *function*

**Note**

With Version 4.0 (or higher) of the debugger, the `stopi in` command is no longer valid, and results in an error message. Replace `stopi in` in your code with `stopi at` for an address or `stop in` for a routine.

Example 9–7 shows how to set a breakpoint in a function.

**Example 9–7 Setting a Breakpoint in a Function**

```
(ladebug)  stop in factorial
[#1: stop in factorial ]
(ladebug)  run
[1] stopped at [factorial:13 0x120001224]
     13     if (i<=1)
(ladebug)
```

Functions and procedures in your executable file are usually each preceded by a prolog that contains information about the function or procedure but which is not of interest to most programmers while debugging their programs. The `stop in` command halts program execution after the prolog, at the first real sourcecode statement.

In Example 9–8 the `stop in` command sets a breakpoint at the beginning of an Ada program procedure named `ADD_INTEGERS`.

**Example 9–8 Setting a Breakpoint at the Start of an Ada Procedure**

```
(ladebug) stop in add_integers
[2] stop in add_integers
(ladebug)run
[2] stopped at [add_integers:3 +0x20002a69,0x120002a68]
    procedure add_integers is
```

#### 9.6.1.4 The stop and stopi Commands

To set a breakpoint that stops program execution when the value of a program variable changes, use one of the following commands:

**stop** variable

**stopi** variable

The `stop` command checks the value of the specified variable every time program execution enters a function. The `stopi` command checks the value of the specified variable after every instruction. The `stopi` command enables you to determine precisely where a variable changes value, but slows program execution.

When you enter a `trace`, `when`, or `stop` command with a variable as an argument (but not in a conditional statement), the debugger records the address of the variable. Instead of looking up the variable name when checking the variable's value, the debugger dereferences the variable's absolute address. This ensures that the correct version of the variable is used regardless of the current context.

If the argument contains a variable as part of a larger expression that cannot be converted into an address and dereferenced, then the debugger must look up the variable name rather than dereference the variable's address. In this case, the debugger does the variable lookup in the current context, which may not be the context active at the time the `trace`, `when`, or `stop` command was entered.

Example 9–9 shows how to set a breakpoint on a variable.

**Example 9–9 Setting a Breakpoint on a Variable**

```
(ladebug)  stop i
[#2: stop if i changes ]
(ladebug)  run
[1] stopped at [main:4 0x120000b14]
      4     for (i=1 ; i<3 ; i++) {
(ladebug)  print i
0
(ladebug)  cont
Value of i changed before "sample.c":13
        Old value = 0
        New value = 1
[2] stopped at [factorial:13 0x120000bb8]
     13     if (i<=1)
(ladebug)  print i
1
(ladebug)
```

#### 9.6.1.5 The stop if and stopi if Commands

To set a breakpoint that suspends program execution when the specified conditional expression evaluates to true, use one of the following commands:

**stop if** (expression)

**stopi if** (expression)

The debugger accepts any valid conditional expression in the language of the program you are debugging. Breakpoints that contain conditional expressions are sometimes called conditional breakpoints.

The `stop if` command checks each time a *function is entered* to see if the expression evaluates to true. The `stopi if` command checks after each *machine instruction is executed* to see if the expression evaluates to true. The `stopi if` version slows program execution considerably.

A variable included in a conditional expression may be undefined in the current scope at some time during program execution. If this is the case, the debugger prints an error message and suspends program execution as if the condition evaluated to true. Global variables work best in conditional statements.

Example 9–10 shows how to set a breakpoint with a user-defined expression.

**Example 9–10 Setting a Conditional Breakpoint**

```
(ladebug)  stop if (iter==2)
[#1: stop if iter==2 ]
(ladebug)  run
[1] stopped at [doit:20 0x120000e14]
     20         ++iter;
(ladebug)  print iter
2
(ladebug)
```

In this example, the breakpoint is activated only when the program variable `iter` equals `2`.

#### 9.6.1.6 Combining Optional Conditions to Customize Breakpoint Command

You can combine the optional conditions to further customize a breakpoint command. By combining the line number or function syntax with a conditional expression, you can create a breakpoint that halts program execution only if a line number or function is reached and an expression evaluates to true. Use one of the following commands to set this type of a breakpoint:

**stop at** line_number **if** (expression)

**stop in** function **if** (expression)

**stopi at** line_number **if** (expression)

Example 9–11 creates a breakpoint that stops only if the program is executing the `factorial` function and the program variable `i` is equal to `2`.

**Example 9–11 Setting a Conditional Breakpoint in a Function**

```
(ladebug)  stop in factorial if (i==2)
[#1: stop in factorial if i==2 ]
(ladebug)  run
1! = 1
[1] stopped at [factorial:13 0x120000bb8]
     13     if (i<=1)
(ladebug)  print
2
(ladebug)
```

### 9.6.2 Breakpoints That Execute Debugger Commands: the when Commands

The `when` and `wheni` commands let you set breakpoints that execute debugger commands, (rather than suspend program execution) when the specified conditions are satisfied. To describe the conditions of the breakpoint, use the same condition syntaxes described in Section 9.6.1.

The syntax for the command that executes a set of debugger commands when a particular address or line number is reached is as follows:

**when** [variable] **at** line_number {command [; . . . ] }

Use the following syntax for an address-oriented command:

**wheni** [variable] **at** address {command [; . . . ] }

The commands in the argument must be enclosed in braces and separated by semicolons. These commands cause the statements in the command list to be executed immediately after the statements at the line number or address specified.

Example 9–12 creates a breakpoint that prints a stack trace when line 16 is reached.

**Example 9–12 Setting a Breakpoint That Executes a Stack Trace**

```
(ladebug)  when at 16 {where}
[#1: when at "sample.c":16 { where } ]
(ladebug)  run
1! = 1
[1] when [factorial:16 0x12000123c]
>0  0x12000123c in factorial(i=2) sample.c:16
#1  0x120001194 in main() sample.c:5
2! = 2
Thread has finished executing
(ladebug)
```

Example 9–13 creates a breakpoint that displays the contents of the expression `HOLD-CHARS` and `SUB-1` in a COBOL program when line 50 is reached and then resumes execution.

**Example 9–13 Setting a Breakpoint That Executes Multiple Commands**

```
(ladebug) when at 50 {print chars of hold-chars; print SUB-1; cont;}
[#3 when at "testa.cob":50 ( print CHARS of HOLD-CHARS; print SUB-1; ; cont ; } ]
```

You can use the `when` and `wheni` commands to set breakpoints at function entry points, as shown in these syntax lines:

**when** [variable] **in** function {command [; . . . ] }

**wheni** [variable] **in** function {command [; . . . ] }

## 9.7 Setting Tracepoints: the trace commands

Tracepoints instruct the debugger to print a message when certain events occur during program execution. You can use tracepoints to notify you when program execution enters and exits program functions or when program variables change value. Tracepoints do not halt program execution but they do slow it down.

The debugger lets you set the following kinds of tracepoints:

- Tracepoints that print a message when functions are entered and exited
- Tracepoints that print a message when a specified variable changes value

You can base a tracepoint on the following conditions:

- Trace only when program execution is within a specified function
- Trace only when program execution corresponds to a specified line of source code
- Trace only when a set of specified conditions evaluates to true

### 9.7.1 Tracepoints That Notify You of Function Entry and Exit

To set an entry/exit tracepoint unconditionally, enter either the `trace` or `tracei` command at the debugger prompt. An unconditional tracepoint is useful for following the execution flow of a program. If you use the `tracei` command, the debugger notifies you when each function's prolog (rather than the function itself) is entered. You can also specify conditions so tracing occurs only if program execution is:

- On a certain line number
- In a certain function
- If a set of conditions you specify evaluates to true

Use one of the following command syntaxes to establish a conditional tracepoint:

**trace at** line_number [**if** (expression)]

**tracei at** address [**if** (expression)]

The *expression* argument can be any valid conditional expression in the language of the program being debugged. The expression can contain debugger variables, program variables, and constants.

Example 9–14 sets a tracepoint at line number 15 of the COBOL program `TESTA`.

**Example 9–14 Setting a Tracepoint**

```
(ladebug) trace at 15
[#3: trace at "testa.cob":15]
```

Example 9–15 shows a tracepoint that traces only if the program variable `i` is equal to `2` and execution is on line 5. When program execution reaches line 5, and `i` is equal to `2`, the trace is activated and the debugger prints the current source line (line 5).

**Example 9–15 Setting a Conditional Tracepoint**

```
(ladebug) trace at 5 if (i==2)
[#1: trace at "sample.c":5 if i==2 ]
(ladebug) run
1! = 1
[1] trace [main:5 0x120000b1c]
>     5          f = factorial(i);
2! = 2
Thread has finished executing
(ladebug)
```

## 9.7.2 Tracepoints That Notify You of a Variable Value Change

You can set a tracepoint that prints a message if the value of a variable changes. Use one of the following commands to establish a variable tracepoint:

**trace** variable

**tracei** variable

When you use the `trace` command, the debugger evaluates the variable when execution enters a function. The debugger prints a message if the value of the variable is different than the value associated with that variable when the previously executed function was entered.

When you use the `tracei` command, the debugger evaluates the variable after each instruction is executed, and prints a message after the statement in which the value changed. Example 9–16 shows the difference between these two commands.

**Example 9–16 Tracing Variables**

```
(ladebug)  trace i
[#2: trace i ]
(ladebug)  tracei i
[#3: tracei i ]
(ladebug)  step
[3] Value of i changed before "sample.c":5
        Old value = 0
        New value = 1
stopped at [main:5 0x120001188]
      5         f = factorial(i);
(ladebug)  step
[2] Value of i changed before "sample.c":13
        Old value = 0
        New value = 1
stopped at [factorial:13 0x120001224]
     13     if (i<=1)
(ladebug)
```

You can also set a conditional tracepoint that notifies you of variable value changes, only if program execution is:

- Within a specified function
- On a certain source-code line
- At a certain memory address
- If a set of conditions you specify evaluates to true

Use one of the following commands to establish a conditional variable tracepoint:

**trace** variable [**at** line_number | **in** function] [**if** (expression)]

**tracei** variable [**at** address | **in** function] [**if** (expression)]

## 9.8 Displaying, Deleting, Disabling, and Enabling Breakpoints and Tracepoints

This section describes the commands used to display, delete, disable, and enable breakpoints and tracepoints. It is written in terms of breakpoints but all of these commands are also applicable to tracepoints.

To list all the breakpoints known to the debugger, enter the `status` command.

To delete, disable, or enable a breakpoint, identify the breakpoint by its reference number. When breakpoints are created, they are associated with a reference number. This reference number is shown when:

- The debugger confirms the creation of a new breakpoint
- The breakpoint is encountered
- You list all of the breakpoints with the `status` command

Example 9–17 uses the `status` command to display active breakpoints in a COBOL program.

**Example 9–17 Using status to Display Breakpoints**

```
(ladebug) status
#1 PC==0x120001e14 in testa "testa.cob":2 {break}
#2 PC==0x120001ba4 in TESTB "testa.cob":47 {break}
#3 PC==0x120001c1c in TESTB "testa.cob":50
   {print CHARS of HOLD-CHARS; print SUB-1; ; cont ; ; }
```

### 9.8.1 Deleting Breakpoints and Tracepoints: the delete Commands

When a breakpoint is no longer needed, use the `delete` command to remove the breakpoint. To delete a single breakpoint, specify the reference number using the following syntax:

**delete** number

To delete more than one breakpoint, separate the reference numbers with commas using the following syntax:

**delete** number [, . . . ]

Example 9–18 shows breakpoints being deleted and the breakpoint status after each deletion.

**Example 9–18 Deleting Breakpoints**

```
(ladebug)  status
#1 PC==0x120001180 in main "sample.c":4 { break }
#2 (at Proc entry and if $trace0!=*0x11fffe48){trace-expr i;set $trace0=}
#3 if $trace1!=*0x11ffffe48 { trace-expr i; set $trace1 = *0x11ffffe48; }
(ladebug)  delete 1
(ladebug)  status
#2 (at Proc entry and if $trace0!=*0x11ffe48) {trace-expr i;set $trace0=}
#3 if $trace1!=*0x11ffffe48 { trace-expr i; set $trace1 = *0x11ffffe48; }
(ladebug)  delete 2,3
(ladebug)  status
(ladebug)
```

To delete all breakpoints known to the debugger, use one of the following commands:

**delete all**

**delete ***

## 9.8.2 Disabling Breakpoints and Tracepoints: the disable Commands

When you disable a breakpoint, the debugger ignores the breakpoint during program execution. A disabled breakpoint does not cause the program to suspend execution.

You can use the following `disable` commands to disable breakpoints:

**disable** *number*

**disable all**

**disable ***

The `disable *` command has the same effect as `disable all`. It disables all breakpoints and all traces.

The disabled breakpoint is still displayed by the `status` command, but it is listed as disabled. Example 9–19 shows breakpoints being disabled.

**Example 9–19 Disabling Breakpoints**

```
(ladebug)  status
#1 PC==0x120000b14 in main "sample.c":4 { break }
#2 PC==0x120000bb8 in factorial "sample.c":13 { break }
#3 PC==0x120000b14 in main "sample.c":4 { break }
(ladebug)  disable 1
(ladebug)  status
#1 PC==0x120000b14 in main "sample.c":4 { break } Disabled
#2 PC==0x120000bb8 in factorial "sample.c":13 { break }
#3 PC==0x120000b14 in main "sample.c":4 { break }
(ladebug)  disable 2,3
(ladebug)  status
#1 PC==0x120000b14 in main "sample.c":4 { break } Disabled
#2 PC==0x120000bb8 in factorial "sample.c":13 { break } Disabled
#3 PC==0x120000b14 in main "sample.c":4 { break } Disabled
(ladebug)
```

## 9.8.3 Enabling Breakpoints and Tracepoints: the enable Commands

Disabled breakpoints remain deactivated until you enter the `enable` command to reactivate the breakpoint. You can use the following `enable` commands to enable disabled breakpoints:

**enable** *number*

**enable all**

**enable ***

Example 9–20 shows the breakpoints that were disabled earlier and then reactivated with the `enable` command.

**Example 9–20 Enabling Breakpoints**

```
(ladebug)  status
#1 PC==0x4001b8 in main "sample.c":4 { break } Disabled
#2 PC==0x400250 in factorial "sample.c":13 { break } Disabled
#3 PC==0x4001b8 in main "sample.c":4 { break } Disabled
(ladebug)  enable 2
(ladebug)  status
#1 PC==0x4001b8 in main "sample.c":4 { break } Disabled
#2 PC==0x400250 in factorial "sample.c":13 { break }
#3 PC==0x4001b8 in main "sample.c":4 { break } Disabled
(ladebug)  enable all
(ladebug)  status
#1 PC==0x4001b8 in main "sample.c":4 { break }
#2 PC==0x400250 in factorial "sample.c":13 { break }
#3 PC==0x4001b8 in main "sample.c":4 { break }
(ladebug)
```

A breakpoint that has been reactivated with the `enable` command appears as an active breakpoint in the status list. Note that the last status list in this example is the same as the first status list in the preceding example (except for the PC values), showing three active breakpoints.

## 9.9 Returning from a Function: the return Command

The `return` command directs the debugger to continue program execution. The command syntax is as follows:

**return** [function]

The `return` command without any arguments directs the debugger to continue program execution until the current function exits. If you enter this command with an argument, the debugger continues program execution until the specified function returns.

The `return` command is also useful for finishing execution of a function you inadvertently stepped into with the `step` command.

In Example 9–21, the `step` command is used to step through program execution. When program execution enters the `factorial` function, the `return` command is used to finish the called function and return control to the program being debugged.

**Example 9–21 Using the return Command**

```
(ladebug) step
stopped at [main:5 0x120001188]
      5                 f = factorial(i);
(ladebug) Return
stopped at [factorial:13 0x120001224]
     13         if (i<=1)
(ladebug) return
stopped at [main:5 0x120001194]
      5         f = factorial(i);
(ladebug)
```

## 9.10 Calling Functions: the call Command

After a breakpoint or a signal suspends program execution, you can execute a single function in your program by using the `call` command, or by including a function call in the *expression* argument of a debugger command. Calling a function lets you test the function's operation with a specific set of parameters.

When the function you call completes normally, the debugger restores the stack and current context that existed before the function was called.

While the program counter is saved and restored, calling a function does not shield the program state from alteration if the function you call allocates memory or alters global variables. If the function affects global program variables, for instance, those variables will be permanently changed. Functions compiled without the debugger option to include debugging information may lack important parameter information and are less likely to yield consistent results when called.

The syntax for the `call` command is as follows:

**call** function ([parameter[, . . . ]])

Specify the function as if you were calling the function from within your program. You can use both constants and locally visible variables as calling parameters. If the function you are calling has no parameters, specify empty parentheses.

The `call` command executes the specified function with the parameters you supply and then returns control to you (at the debugger prompt) when the function returns. The `call` command discards the return value of the function. If you embed the function call in the *expression* argument of a `print` command, the debugger prints the return value after the function returns.

Example 9–22 shows both methods of calling a function.

**Example 9–22 Calling a Function from the Debugger Prompt**

```
(ladebug) call factorial(5)
(ladebug) print factorial(5)
120
(ladebug)
```

In this example, the `call` command results in the return value being discarded while the embedded call passes the return value of the function to the `print` command, which in turn prints the value. You can also embed the call within a more involved expression, as shown in Example 9–23.

**Example 9–23 Embedding a Function Call in an Expression**

```
(ladebug) print 341 + factorial(6) / 2
701
(ladebug)
```

All breakpoints or tracepoints defined during the session are active when executing a called function. When program execution halts during function execution, you can examine program information, execute one line or instruction, continue execution of the function, or call another function.

When you call a function when execution is suspended in a called function, you are nesting function calls, as shown in Example 9–24.

**Example 9–24 Nesting Function Calls**

```
(ladebug) status
#1 PC==0x120001180 in main "sample.c":4 { break }
#2 PC==0x12000123c in factorial "sample.c":16 { break }

(ladebug) call factorial(5)
[2] stopped at [factorial:16 0x12000123c]
     16        return (i * factorial(i-1) );
```

(continued on next page)

**Example 9–24 (Cont.) Nesting Function Calls**

```
(ladebug) where
>0  0x12000123c in factorial(i=5) sample.c:16
(ladebug) print i
5
(ladebug) s
stopped at [factorial:13 0x120001224]
     13     if (i<=1)
(ladebug) call factorial(15)
[2] stopped at [factorial:16 0x12000123c]
     16         return (i * factorial(i-1) );
(ladebug) where
>0  0x12000123c in factorial(i=15) sample.c:16
(ladebug) disable 2
(ladebug) return
Called Procedure Returned
stopped at [factorial:13 0x120001224]
     13     if (i<=1)
(ladebug) where
>0  0x120001224 in factorial(i=4) sample.c:13
#1  0x12000124c in factorial(i=5) sample.c:16
(ladebug) cont
Called Procedure Returned
stopped at [main:4 0x120001180]
      4     for (i=1 ; i<=3 ; i++) {
(ladebug) where
>0  0x120001180 in main() sample.c:4
(ladebug)
```

The Ladebug debugger supports function calls and expression evaluations that call functions, with the following limitations:

- The debugger does not support passing and returning structures by value.
- The debugger does not implicitly construct temporary objects for call parameters.
- Optimization can prevent the debugger from knowing the type of a function return. Therefore, the debugger assumes returns are of the type `int` if the functions are optimized. If the returns are a different type, try using casts when calling the optimized functions.

## 9.11 Unaligned Data Accesses: the catch and ignore Commands

Unaligned data can slow program execution. You can use the `catch` command to cause Ladebug to stop on each unaligned data access or the `ignore` command so the debugger does *not* stop.

**catch unaligned**

Enter the `catch unaligned` command to instruct the debugger to stop when unaligned access occurs in the debuggee process. The debugger:

- Stops at the instruction immediately following the instruction where the unaligned access occurs
- Issues a message

Example 9–25 shows this:

**Example 9–25 Catching Unaligned Access**

```
(ladebug)  catch unaligned
(ladebug)  run

Unaligned access pid=12538 <unaligned_test> va=140002901
pc=120001168 ra=12000114c type=stl
Thread received signal BUS
stopped at [main:8 0x12000116c]
     8        temp = *j;         /* unaligned access */
```

You can distinguish between the SIGBUS of unaligned access and a normal SIGBUS because of the "unaligned access" message (issued by the kernel).

**ignore unaligned**

Enter the `ignore unaligned` command to instruct the debugger not to stop when unaligned access occurs. This is the default.

The unalignment functionality is implemented through the `setsysinfo` system call and SIGBUS. Ladebug makes the `setsysinfo` system call in the debuggee process so that unaligned accesses for attached processes are also caught.

Ladebug will preserve the previous user settings when these commands are issued.

If the user program executes a `setsysinfo` or `uac` (unaligned access control flag) that is inconsistent with the `catch unaligned` or `ignore unaligned` command, however, then the behavior of these commands will be affected.

If SIGBUS is ignored and then the `catch unaligned` command is issued:

```
(ladebug) ignore sigbus
(ladebug) catch unaligned

Warning: SIGBUS is currently being ignored  -
SIGBUS is being caught to catch Unaligned accesses.
```

If Ladebug is catching unaligned accesses and then SIGBUS is ignored:

```
(ladebug) catch unaligned
(ladebug) ignore sigbus

Warning: SIGBUS is currently being ignored  -
Cannot catch Unaligned accesses.
```

## 9.12 Using the pop Command

The `pop` command removes one or more execution frames from the call stack. The `pop` command undoes the work already done by the removed execution frames. It does not, however, reverse side effects such as changes to global variables. You may need to use the `assign` command to restore the values of global variables.

The `pop` command is useful when execution has already passed an error that needs to be corrected.

The syntax of the command is as follows:

**pop** [number_of_frames]

The optional argument is the number of execution frames to remove from the call stack. If you do not specify the argument, one frame is removed. If specified, the number must be a positive integer less than or equal to the number of frames currently on the call stack.

It is an error to issue the `pop` command when there is no running program.

The following fragment of a debugger session shows the use of the `pop` command:

```
Reading symbolic information ...done
(ladebug) bp factorial
[#1: stop in int factorial(int) ]
(ladebug) r
Factorial time has begun.
[1] stopped at [factorial:30 0x120001524]
     30   printf("entered factorial\n");
(ladebug) where
>0  0x120001524 in factorial(ii=1) c_listfunc_factorial.c:30
#1  0x12000140c in main() c_listfunc.c:41
(ladebug) pop
[1] stopped at [main:41 0x120001410]
     41     f = factorial(i);
(ladebug) c
[1] stopped at [factorial:30 0x120001524]
     30   printf("entered factorial\n");
```

## 9.13 Controlling the Debugging of Attached Processes: the attach and detach Commands

To attach to a running process, first invoke Ladebug with a process ID number and the matching image file from the command line or from within Ladebug using the `attach` command followed by the process ID and image file. For information on invoking Ladebug to attach to a running process, see Section 7.8.9.

After Ladebug attaches to a process, control is returned to the debugger when the process stops (for example, after having received a signal). You can also manually return control to the debugger by pressing Ctrl/C or by setting the debugger variable `$stoponattach` to 1 to stop the attached process.

Use the `detach` command to detach the debugger from the previously attached process, based on the process ID you specify. Ladebug only detaches the specified process and removes all the user-specified breakpoints from that process.

The following restrictions apply when you debug an attached process:

- In the context of an attached process, the `run` and `rerun` commands are disabled.
- The `quit` command will terminate any process Ladebug created when invoking the debugger with an image file or when using the `load` *image_file* command. If it is attached to any process, Ladebug will detach itself from the process.

## 9.14 Debugging Programs with Stripped Images

The `strip` command removes the symbol table and relocation information ordinarily attached to the output of the assembler and loader. If you are debugging a binary image that has been stripped, only machine-level debugging is supported. For information on machine-level debugging, see Chapter 18.

## 9.15 Using Environment Variables Within the Debugger

Ladebug provides commands for manipulating the environment of subsequent debuggees with environment variables. From within the debugger, you can use the following commands:

**setenv** [env_variable [value]]
**export** [env_variable [= value]]

Sets the value of the specified environment variable. If no variable is specified, the command displays the values of all environment variables. If a variable is specified but no value is specified, then the variable is set to NULL.

`setenv` and `export` are synonyms.

**printenv** [env_variable]

Displays the value of the specified environment variable. If none is specified, the command displays the value of all environment variables.

**unsetenv** [env_variable]

Removes the specified environment variable. If no variable is specified, all environment variables are removed.

### Manipulation of Subsequent Environments Without Affecting the Current Environment

The environment-manipulation commands apply to any subsequent debuggee environment but not to the current environment. For example:

```
% ladebug a.out
(ladebug) bpmain; run
Stopped in main
(ladebug) setenv LD_LIBRARY_PATH /usr/proj/libraries
```

At this point, the setting of the environment variable LD_LIBRARY_PATH is not in effect; it does not apply to the current execution of the file `a.out`, which was started by the `run` command. The environment variable will be applied after you create a new debuggee by means of one of the following:

- `run` command

- `rerun` command
- `fork(2)`

This functionality is useful because it allows you to have different environments: it allows an environment for processes created by Ladebug that is different from Ladebug's environment, and also different from the environment of the shell from which Ladebug was invoked.

**Note**

The environment-manipulation commands have no effect on other environments. For example, the debuggee program cannot affect the output of the Ladebug command `setenv`. Nor can the Ladebug command `setenv` affect subsequent debuggee calls to `getenv(3)`, which, like `putenv(3)` and `clearenv(3)`, is an environment manipulation routine in `libc.a`.

### Differences in Prior Versions

For Ladebug prior to Version 4.0, the only way to set environment variables was to set them in the shell, before invoking the debugger, using one of the following commands:

- `export`, for the Bourne shell and the Korn shell
- `setenv`, for the C shell

The environment thus created in the shell applies to the debugger and also is inherited by any debuggees subsequently created.

Two notable disadvantages of this method are as follows:

- The debugger and debuggee environments have to be the same and cannot be independently manipulated.
- A change to the environment variable `LD_LIBRARY_PATH` can prevent future debugger calls to `dlopen(3)` from operating correctly.

### Differences from dbx Debugger Functionality

If you are familiar with the `dbx` debugger, it is important to be aware of the following differences:

- The `dbx` debugger also has a `setenv` command, which works differently. In `dbx`, unlike Ladebug, the `setenv` command changes the debugger's current environment. The current environment is inherited by any subsequently created debuggee processes.

- In `dbx`, you can use the `setenv` command to change the behavior of subsequent commands passed to the shell with `sh`. For example,

  ```
  (dbx) setenv FOO=value
  (dbx) sh command-using-FOO
  ```

  In Ladebug, by contrast, the `setenv` command does not change the current debugger environment, and so the command does not affect a shell invoked by the `sh` command. In Ladebug, you accomplish the same thing in the following way:

  1. Invoke the shell.
  2. Define the environment variable FOO.
  3. In the shell, execute a command using FOO.

  For example,

  ```
  (ladebug) sh FOO=value; command-using-FOO
  ```

# Part III

## Language-Specific Topics

This part describes topics specific to the programming languages Ada, C++, COBOL, Fortran, and Fortran 90.

# 10 Debugging DEC C++ Programs

## 10.1 Significant Supported Features

The Digital Ladebug debugger debugs programs compiled with the DEC C++ compiler on the Digital UNIX operating system. The following features are supported:

- C++ names and expressions, including:
  - Explicit and implicit `this` pointer to refer to class members
  - Scope resolution operator (::)
  - Member access operators: period (.) and right arrow (->)
  - Reference types
  - Template instantiations
  - C++ exception handling
- Setting breakpoints in:
  - Member functions, including static and virtual functions
  - A member function in a particular object
  - Overloaded functions
  - Constructors and destructors
  - Template instantiations
  - Exception handlers
- Changing the current class scope to set breakpoints and examine static members of a class that are not currently in scope
- Calling overloaded functions
- Debugging programs containing a mixture of C++ code and code in other languages

Some of these features are discussed further in this chapter. This chapter also explains the limitations on Ladebug support for the C++ language.

## 10.2 DEC C++ Flags for Debugging

To prepare to use the Ladebug debugger on a C++ program, invoke the compiler with the appropriate debugging flag: `-g`, `-g2`, or `-g3`. For example:

```
% cxx -g sample.cxx -o sample
```

The `-g` flag on the compiler command instructs the compiler to write the program's debugger symbol table into the executable image. This flag also turns off the default optimization, which could cause a confusing debugging session.

Refer to the `cxx(1)` reference page or other compiler documentation for information about the various `-g`*n* flags and their relationship to optimization.

Traceback information and symbol table information are both necessary for debugging. They:

- Enable the debugger to translate virtual addresses into source program routine names and compiler-generated line numbers.
- Provide symbol definitions for user-defined variables, arrays (including dimension information), structures, and labels of executable statements.

Traceback and symbol table information result in a larger object file. When you have finished debugging your program, you can remove traceback information with the the `strip` command (see `strip(1)` ). To remove symbol table information, you can compile and link again with `-g0` or `-g1` to create a new executable program.

Typical uses of the debugging flags at the various stages of program development are as follows:

- During the early stages of program development, use the `-g` (or `-g2` ) flag. You can specify `-O0` (which is the default with the `-g` or `-g2` flag) to enable debugging and to create unoptimized code. This flag also might be chosen later to debug reported problems from later stages.
- During the later stages of program development, use `-g0` or `-g1` to reduce the size of the object file† (and therefore the memory needed for program execution), usually with optimized code.

† `g1` results in a larger object file than `-g0` but smaller than the other `g`*n* flags.

## 10.3 Calling Overloaded Functions

To call overloaded functions from the debugger, you must set `$overloadmenu` to 1 (the default), as follows:

```
(ladebug) set $overloadmenu = 1
```

Then, use the following `call` command syntax:

**call** function ([parameter[, . . . ]])

For example:

```
(ladebug) call foo(15)
```

The debugger will call the function that you select from the menu of overloaded names.

## 10.4 Setting the Class Scope

The debugger maintains the concept of a current context in which to perform lookup of program variable names. The current context includes a file scope and either a function scope or a class scope. The debugger automatically updates the current context when program execution suspends.

The `class` command lets you set the scope to a class in the program you are debugging. The syntax for the `class` command is as follows:

**class** class_name

Explicitly setting the debugger's current context to a class allows for visibility into a class to:

- Set a breakpoint in a member function.
- Print static data members.
- Examine any data member's type.

After the class scope is set, you can set breakpoints in the class's member functions and examine data without explicitly mentioning the class name. If you do not want to affect the current context, you can use the scope resolution operator (::) to access a class whose members are not currently visible.

To display the current class scope (if one exists), enter the `class` command with no argument.

Example 10–1 shows the use of the `class` command to set the class scope to `S` in order to make member function `foo` visible so a breakpoint can be set in `foo`.

**Example 10–1 Setting the Class Scope**

```
(ladebug) stop in main; run
[#1: stop in main ]
[1] stopped at [int main(void):26 0x120000744]
     26      int result = s.bar();
(ladebug) stop in foo
Symbol foo not visible in current scope.
foo has no valid breakpoint address
Warning: Breakpoint not set
(ladebug)  class S
class S  {
  int i;
  int j;
  S (void);
  ~S (void);
  int foo (void);
  virtual int bar (void);
}
(ladebug) stop in foo
[#2: stop in foo (void) ]
(ladebug)
```

## 10.5 Displaying Class Information

The `whatis` and `print` commands display information on a class. Use the `whatis` command to display static information about the classes. Use the `print` command to view dynamic information about class objects.

The `whatis` command displays the class type declaration, including:

Data members
Member functions
Constructors
Destructors
Static data members
Static member functions

For classes that are derived from other classes, the data members and member functions inherited from the base class *are not* displayed. Any member functions that *are* redefined from the base class *are* displayed.

The `whatis` command used on a class name displays all class information, including constructors. To use this command on a constructor only, use the following syntax:

**whatis** class_name::class_name

Constructors and destructors of nested classes must be accessed with the `class` command. The following syntax is for a constructor:

**class** class_name::(type signature)

The following syntax is for a destructor:

**class** class_name~::(type signature)

The `print` command lets you display the value of data members and static members. Information regarding the public, private, or protected status of class members is not provided, since the debugger relaxes the related status rules to be more helpful to users.

The type signatures of member functions, constructors, and destructors are displayed in a form that is appropriate for later use in resolving references to overloaded functions.

Example 10–2 shows the `whatis` and `print` commands in conjunction with a class.

**Example 10–2 Displaying Class Information**

```
(ladebug) list 1, 9
      1 class S {
      2 public:
      3     int i;
      4     int j;
      5     S() { i = 1; j = 2; }
      6     ~S() { }
      7     int foo ();
      8     virtual int bar();
      9 };
(ladebug) whatis S
class S  {
  int i;
  int j;
  S (void);
  ~S (void);
  int foo (void);
  virtual int bar (void);
} S
(ladebug) whatis S :: bar
int bar (void)
(ladebug) stop in S :: foo
[#2: stop in S :: foo ]
(ladebug) run
[2] stopped at [int S::foo(void):13 0x120000648]
     13     return i;
(ladebug) print S :: i
1
(ladebug)
```

## 10.6 Displaying Object Information

The `print` and `whatis` commands display information on instances of classes (objects). Use the `whatis` command to display the class type of an object. Use the `print` command to display the current value of an object. You can print an object's contents all at one time by using the following `print` command syntax:

**print** object

You can also display individual object members using the member access operators, period (.) and right arrow (->), in a `print` command.

You can use the scope resolution operator (::) to reference global variables, to reference hidden members in base classes, to explicitly reference a member that is inherited, or otherwise to name a member hidden by the current context.

When you are in the context of a nested class, you must use the scope resolution operator to access members of the enclosing class.

Example 10–3 shows how to use the `print` and `whatis` commands to display object information.

**Example 10–3 Displaying Object Information**

```
(ladebug) whatis s
class S  {
  int i;
  int j;
  S (void);
  ~S (void);
  int foo (void);
  virtual int bar (void);
} s
(ladebug) stop in S::foo; run
[#1: stop in s.foo ]
[1] stopped at [int S::foo(void):13 0x120000638]
     35      return i;
(ladebug) print *this
class {
        i = 1;
        j = 2;
    }
(ladebug) print i, j
1 2
(ladebug) print this->i, this->j
1 2
(ladebug)
```

## 10.7 Displaying Virtual and Inherited Class Information

When you use the `print` command to display information on an instance of a derived class, Ladebug displays both the new class members as well as the members inherited from a base class. Base class member information is nested within the inherited class information. Ladebug displays members that are inherited from a base class, using the following notation:

baseclass:{var1:value1,var2:value2, . . . varN:valueN}

Pointers to members of a class are not supported.

If you have two members in an object with the same name but different base class types (multiple inheritance), you can refer to the members using the following syntax:

object.class::member

This syntax is more effective than using the `object.member` and `object->member` syntaxes, which can be ambiguous. In all cases, the Digital Ladebug debugger uses the C++ language rules as defined in *The Annotated C++ Reference Manual* to determine which member you are specifying.

Example 10–4 shows a case where the expanded syntax is necessary.

**Example 10–4 Resolving References to Objects of Multiple Inherited Classes**

```
(ladebug) print dinst.ambig
Ambiguous reference
Selecting 'ambig' failed!
Error: no value for dinst.ambig
(ladebug) print dinst.B::ambig
2
(ladebug)
```

Trying to examine an inlined member function that is not called results in the following error:

```
Member function has been inlined.
```

Ladebug will report this error regardless of the setting of the `-noinline_auto` compilation flag. As a workaround, include a call to the given member function somewhere in your program. (The call does not need to be executed.)

If a program is not compiled with the `-g` flag, a breakpoint set on an inline member function may confuse the debugger.

## 10.8 Member Functions on the Stack Trace

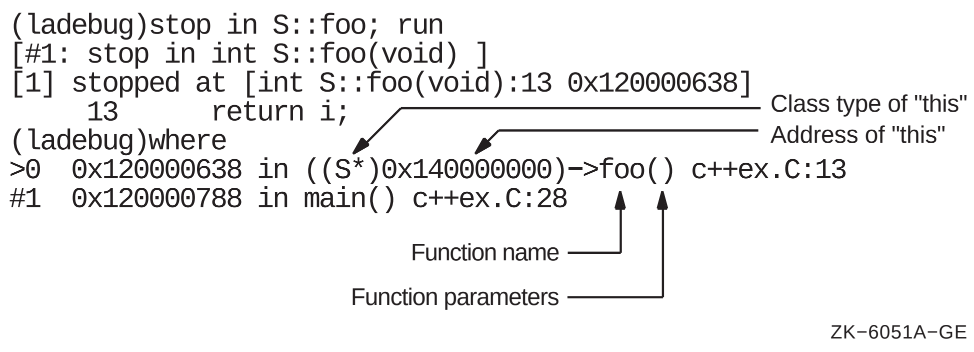
The implicit `this` pointer, which is a part of all nonstatic member functions, is displayed as the address on the stack trace. The class type of the object is also given.

Sometimes the debugger does not see class type names with internal linkage. When this happens, the debugger issues the following error message:

```
Name is overloaded.
```

The stack trace in Figure 10–1 displays a member function `foo` of an object declared with class type `S`.

**Figure 10–1 A Stack Trace Displaying a Member Function**



```
(ladebug)stop in S::foo; run
[#1: stop in int S::foo(void) ]
[1] stopped at [int S::foo(void):13 0x120000638]
     13      return i;
(ladebug)where
>0  0x120000638 in ((S*)0x140000000)->foo() c++ex.C:13
#1  0x120000788 in main() c++ex.C:28
```

## 10.9 Resolving Ambiguous References to Overloaded Functions

In most cases, the debugger works with one specific function at a time. In the case of overloaded function names, you must specify the desired overloaded function. There are two ways to resolve references to overloaded function names, both under the the control of the `$overloadmenu` debugger variable (the default setting of this debugger variable is 1):

- Choose the correct reference from a selection menu.

  If you had changed the default value for the `$overloadmenu` variable, to enable menu selection of overloaded names, set the `$overloadmenu` to 1 as shown in Example 10–5. Using this method, whenever you specify a function name that is overloaded, a menu will appear with all the possible functions; you must select from this menu. In Example 10–5, a breakpoint is set in `foo`, which is overloaded.

**Example 10–5 Resolving Overloaded Functions by Selection Menu**

```
(ladebug) set $overloadmenu = 1
(ladebug) stop in foo
Enter the number of the overloaded function you want
----------------------------------------------------
     1 int foo (void)
     2 void foo (const char *)
     3 void foo (char *)
     4 void foo (double)
     5 int foo (const double *)
     6 None of the above
----------------------------------------------------
5
[#1: stop in int S::foo(const double*) ]
(ladebug)
```

- Enter the function name with its full type signature.

  If you prefer this method, set the `$overloadmenu` variable to 0. To see the possible type signatures for the overloaded function, first display all the declarations of an overloaded function by using one of the following `whatis` commands:

  **whatis** function

  **whatis** class_name::function

  You cannot select a version of an overloaded function that has a type signature containing ellipsis points (...). Pointers to functions with type signatures that contain the `list` parameter or ellipsis arguments are not supported.

  Use one of the displayed function type signatures to refer to the desired version of the overloaded function. If a function has no parameter, include the `void` parameter as the function's type signature. In Example 10–6, the function context is set to `foo( )`, which is overloaded.

**Example 10–6 Resolving Overloaded Functions by Type Signature**

```
(ladebug) func foo
Error: foo is overloaded
(ladebug) func foo(double)
S::foo(double) in c++over.C line No. 156:
    156     printf ("void S::foo (double d = %f)\en", d);
(ladebug)
```

## 10.10 Setting Breakpoints

When you set a breakpoint in a C function, the debugger confirms it by echoing the breakpoint command along with the status number for the breakpoint. When you set a breakpoint in a C++ function, the debugger also prints the type signature of the function in which the breakpoint was set.

The following sections describe setting breakpoints in member functions, in overloaded functions, and in constructors and destructors. See Section 10.12.1 for information on setting breakpoints in exception handlers.

### 10.10.1 Setting Breakpoints in Member Functions

To set a breakpoint that stops in a member function, use one of the following `stop in` commands:

**stop in** function

**stop in** class_name::function

These forms of specifying a breakpoint in a function use the static class type information to determine the address of the function at which to set the breakpoint, and presume that no run-time information from an object is needed.

In Example 10–7, a breakpoint is set for the `bar` member function of class `S`.

**Example 10–7 Setting Breakpoints in Member Functions**

```
(ladebug) stop in S :: bar
[#1: stop in S::bar(void) ]
(ladebug) status
#1 PC==0x120000658 in S::bar(void) "c++ex.C":18 { break }
(ladebug) run
[1] stopped at [S::bar(void):18 0x120000658]
     18      return j;
```

```
(ladebug) where
>0  0x120000658 in ((S*)0x120000658)->bar() c++ex.C:18
#1  0x120000750 in main() c++ex.C:26
(ladebug)
```

If you need run-time information from the object to determine the correct virtual function at which to set a breakpoint, qualify the function name with the object, using one of the following `stop in` commands:

**stop in** object.function

**stop in** objectpointer->function

These forms of setting the breakpoint cause the debugger to stop at the member function in all objects declared with the same class type as the specified object. In Example 10–8, objects `s` and `t` are both declared to be of class type `S`. A breakpoint is set for the `bar` member function. The first time the debugger stops at `bar()` is for object `s`. The second time the debugger stops in `bar()` is for object `t`.

**Example 10–8 Setting Breakpoints in Virtual Member Functions**

```
(ladebug) stop in main
[#1: stop in main(void) ]
(ladebug) run
[1] stopped at [main(void):26 0x120000744]
     26      int result = s.bar();
(ladebug) stop in s.bar
[#2: stop in S::bar(void) ]
(ladebug) status
#1 PC==0x120000744 in main(void) "c++ex.C":26 { break }
#2 PC==0x120000658 in S::bar(void) "c++ex.C":18 { break }
(ladebug) print &s
0x140000000
(ladebug) print &t
0x140000008
(ladebug) cont
[2] stopped at [S::bar(void):18 0x120000658]
     18      return j;
(ladebug) where
>0  0x120000658 in ((S*)0x140000000)->bar() c++ex.C:18
#1  0x120000750 in main() c++ex.C:26
(ladebug) cont
[2] stopped at [S::bar(void):18 0x120000658]
     18      return j;
```

(continued on next page)

**Example 10–8 (Cont.) Setting Breakpoints in Virtual Member Functions**

```
(ladebug) where
>0  0x120000658 in ((S*)0x140000008)->bar() c++ex.C:18
#1  0x12000076c in main() c++ex.C:27
(ladebug)
```

To set a breakpoint that stops only in the member function for this specific object and not all instances of the same class type, you must specify `this` as an additional conditional clause to the `stop` command. Use one of the following `stop in` commands:

**stop in** object.function **if &** object **==** **this**

**stop in** objectpointer->function **if &** objectpointer **==** **this**

These forms of the `stop in` command instruct the debugger to stop in the function only for the object specified by the `this` pointer. In Example 10–9, which is running the same program as Example 10–8, the breakpoint is set for the member function for object `s` only. After stopping in `bar()` for object `s`, further execution of the program results in the program running to completion.

**Example 10–9 Setting Breakpoints in Member Functions for a Specific Object**

```
(ladebug) stop in s.bar if &s==this
[#2: stop in s.bar if &s==this ]
(ladebug) status
#1 PC==0x120000744 in main(void) "c++ex.C":26 { break }
#2 (PC==0x120000658 in S::bar(void) "c++ex.C":18 and if &s==this) {break}
(ladebug) print &s
0x140000000
(ladebug) cont
[2] stopped at [S::bar(void):18 0x120000658]
     18      return j;
(ladebug) where
>0  0x120000658 in ((S*)0x10000010)->bar() c++ex.C:18
#1  0x120000750 in main() c++ex.C:26
(ladebug) cont
Thread has finished executing
(ladebug)
```

### 10.10.2 Setting Breakpoints in Overloaded Functions

To set a breakpoint in an overloaded function, you must provide the full type signature of the function. Use one of the following `stop in` commands:

**stop in** function (type_signature)

**stop in** class_name::function (type_signature)

If the desired version of the function has no parameters, you must enter `void` for the type signature. In Example 10–10, the breakpoint is set for specific versions of the overloaded function `foo`.

**Example 10–10 Setting Breakpoints in Specific Overloaded Functions**

```
(ladebug) stop in foo(double)
[#1: stop in void S::foo(double) ]
(ladebug) stop in foo(void)
[#2: stop in int S::foo(void) ]
(ladebug) status
#1 PC==0x120001508 in void S::foo(double) "c++over.C":156 { break }
#2 PC==0x120000ef4 in int S::foo(void) "c++over.C":59 { break }
(ladebug)
```

To set a breakpoint that stops in all versions of an overloaded function, use one of the following `stop in all` commands:

**stop in all** function

**stop in all** class_name::function

In Example 10–11, the breakpoint is set for all versions of the overloaded function `foo`.

**Example 10–11 Setting Breakpoints in All Versions of an Overloaded Function**

```
(ladebug) stop in all foo
[#1: stop in all foo ]
(ladebug)
```

You can also set a breakpoint in an overloaded function by setting a breakpoint at the line number where the function begins. Be sure the current file context points to the file containing the function’s source code before you set the breakpoint. In Example 10–12, the breakpoint is set for the overloaded functions by line number.

**Example 10–12 Setting Breakpoints in Overloaded Functions by Line Number**

```
(ladebug) stop at 59
[#1: stop at "c++over.C":59 ]
(ladebug) stop at 156
[#2: stop at "c++over.C":156 ]
(ladebug) status
#1 PC==0x120000ef4 in S::foo(void) "c++over.C":59 { break }
#2 PC==0x120001508 in S::foo(double) "c++over.C":156 { break }
(ladebug)
```

### 10.10.3 Setting Breakpoints in Constructors and Destructors

To set a breakpoint in a constructor, use one of the following `stop in` commands:

**stop in** class_name::class_name [(type_signature)]

**stop in** class_name [(type_signature)]

The type signature is only necessary to resolve an ambiguous reference to a constructor that is overloaded. In Example 10–13, a breakpoint is set in a constructor.

**Example 10–13 Setting Breakpoints in Constructors**

```
(ladebug) class S
class S  {
  int i;
  int j;
  S (void);
  ~S (void);
  int foo (void);
  virtual int bar (void);
}
(ladebug) stop in S
[#1: stop in S::NS(void) ]
```

```
(ladebug) status
#1 PC==0x1200005b8 in S::S(void) "c++ex.C":5 { break }
(ladebug)
```

You can similarly set a breakpoint in a destructor using the following `stop in` command syntax:

**stop in** ~class_name

In Example 10–14, the breakpoint is set for the destructor.

**Example 10–14 Setting Breakpoints in Destructors**

```
(ladebug) stop in ~S
[#1: stop in ~S::S(void) ]
(ladebug) status
#1 PC==0x1200005f8 in S::~S(void) "c++ex.C":6 { break }
(ladebug)
```

As with any function's type signature specification, constructors and destructors that have no parameters must be referenced with a type signature of `void`.

## 10.11 Class Templates and Function Templates

The debugger provides support for debugging class templates and function templates in much the same way as other classes and functions in C++, with the limitations described in Section 10.14.

You can use the `whatis` command on an instantiation of the function template as shown in Example 10–15.

**Example 10–15 Example of a Function Template**

```
(ladebug) list 1
      1 // remember to compile with -define_templates
      2 template<class T> int compare(T t1, T t2)
      3 {
      4         if (t1 < t2) return 0;
      5         else         return 1;
      6 }
      7
      8 main()
      9 {
>    10         int i = compare(1,2);
     11 }
(ladebug) whatis compare
int compare (int, int)
(ladebug)
```

You can set a breakpoint in a template function as shown in Example 10–16.

**Example 10–16 Setting a Breakpoint in the Template Function**

```
(ladebug) stop in compare
[#2: stop in compare(int, int) ]
(ladebug) run
[2] stopped at [compare(int, int):4 0x120000560]
      4         if (t1 < t2) return 0;
(ladebug) stop in S.pop
[#1: stop in stack<int,100>::pop(void) ]
(ladebug) run
stopped at [stack<int,100>::pop(void):17 0x120001e0c]
     17         return s[--top];
(ladebug) func
stack<int,100>::pop(void) in c++classtemp.C line No. 17:
     17         return s[--top];
(ladebug) print top
2
(ladebug)
```

Example 10–17 displays the class definition of a particular instantiation of a parameterized stack.

**Example 10–17 Displaying an Instantiated Class Template**

```
(ladebug) whatis stack<int,100>
class stack<int,100>  {
  array [subrange 0 ... 99 of int] of int s;
  int top;
  stack<int,100> (void);
  void push (int);
  int pop (void);
} stack<int,100>
(ladebug)
```

You can explicitly set your current class scope to a particular instantiation of a class template if you are not in the proper class scope. See Example 10–18.

**Example 10–18 Setting Current Class Scope to an Instantiated Class**

```
(ladebug) stop in push
Symbol push not visible in current scope.
push has no valid breakpoint address
Warning: Breakpoint not set
(ladebug) class
Current context is not a class
(ladebug) class S
class stack<int,100>  {
  array [subrange 0 ... 99 of int] of int s;
  int top;
  stack<int,100> (void);
  ~stack<int,100> (void);
  void push (int);
  int pop (void);
}
(ladebug) stop in push
[#4: stop in stack<int,100>::push(int) ]
(ladebug) run
[4] stopped at [stack<int,100>::push(int):10 0x120001cd0]
     10               s[top++] = item;
(ladebug)
```

As an alternative, you can use the following syntax:

```
(ladebug)  class stack<int,100>
```

## 10.12 Debugging C++ Exception Handlers

You can debug C++ exception handlers in programs by setting breakpoints in the exception handler or in the predefined C++ functions that are used when exceptions occur. You can also examine and modify variables that are used in exception handlers.

### 10.12.1 Setting Breakpoints in Exception Handlers

As shown in Example 10–19, you can set a breakpoint for an exception handler at the line number where the code for the exception handler begins. You can then step through the exception handler, examine or modify variables, or continue executing the program. You can also set breakpoints in C++ functions used to handle exceptions as follows:

| | |
|---|---|
| `terminate` | Gains control when any unhandled exception occurs and terminates the progrm |
| `unexpected` | Gains control when a function containing an exception specification tries to throw an exception that is not in the exception specification |

**Example 10–19 Setting Breakpoints in Exception Handlers**

```
(ladebug) list 24
     24     try
     25     {
     26          foo();
     27     }
     28     catch(char * str) { printf("Caught %s.\n",str); }
     29     catch(...) { printf("Caught something.\n"); }
     30
     31 return 0;
     32 }
(ladebug) stop at 24
[#1: stop at "except.C":26 ]
(ladebug) stop in unexpected
[#2: stop in unexpected ]
(ladebug) run
[1] stopped at [int main(void):26 0x400370]
     26          foo();
(ladebug) cont
[2] stopped at [unexpected:631 0x4010a8]
(Cannot find source file cxx_exc.c)
```

(continued on next page)

**Example 10–19 (Cont.) Setting Breakpoints in Exception Handlers**

```
(ladebug) cont
In my_unexpected().
Caught HELP.
Thread has finished executing
(ladebug)
```

### 10.12.2 Examining and Modifying Variables in Exception Handlers

After you set a breakpoint to stop the execution in the exception handler, you can access the variables used in the exception handler the same way you would examine and modify other program variables. See Section 8.3.

## 10.13 Advanced Program Information: Verbose Mode

By default, the debugger gives no information on virtual base class pointers for the following:

- Derived classes
- Virtual pointer tables for virtual functions
- Compiler-generated function members
- Compiler-generated temporary variables
- Implicit parameters in member functions

By setting the `$verbose` debugger variable to 1, you can request that this information be printed in subsequent debugger responses.

When the `$verbose` debugger variable is set to 1 and you display the contents of a class using the `whatis` command, several of the class members listed are not in the source code of the original class definition. The following line shows sample output from the `whatis` command:

```
array [subrange 0 ... 0 of int] of vtable * _\|_vptr;
```

The `vtable` variable contains the addresses of all virtual functions associated with the class. Several other class members are generated by the compiler for internal use.

The compiler generates additional parameters for nonstatic member functions. When the `$verbose` debugger variable is set to 1, these extra parameters are displayed as part of each member function's type signature. If you specify a version of an overloaded function by entering its type signature and the variable is set to 1, you must include these parameters. Do not include these parameters if the variable is set to 0.

When the `$verbose` variable is set to 1, the output of the `dump` command includes not only standard program variables but also compiler-generated temporary variables.

Example 10–20 prints class information using the `whatis` command when the `$verbose` variable is set to 1.

**Example 10–20 Printing a Class Description in Verbose Mode**

```
(ladebug) print $verbose
0
(ladebug) whatis S
class S  {
  int i;
  int j;
  S (void);
  ~S (void);
  int foo (void);
  virtual int bar (void);
} S
(ladebug) set $verbose = 1
(ladebug) print $verbose
1
(ladebug) whatis S
class S  {
  int i;
  int j;
  array [subrange 0 ... 0 of int] of vtbl * _\|_vptr;
  S (S* const);
  S (S* const, const S&);
  ~S (S* const, int);
  int foo (S* const);
  S& operator = (S* const, const S&);
  virtual int bar (S* const);
} S
(ladebug)
```

When displaying information on virtual base classes, the debugger prints pointers to the table describing the base class for each virtual base class object member. This pointer is known as the `bptr` base pointer. This pointer is printed after the class member information.

## 10.14 Limitations on Ladebug Support for C++

Ladebug interprets C++ names and expressions using the language rules described in *The Annotated C++ Reference Manual* (Ellis and Stroustrup, 1990, Addison-Wesley). C++ is a distinct language, rather than a superset of C. Where the semantics of C and C++ differ, Ladebug provides the interpretation appropriate for the language of the program being debugged.

To make Ladebug more useful, it relaxes some standard C++ name visibility rules. For example, you can reference both public and private class members.

The following limitations apply when you debug a C++ program:

- If a program is not compiled with the `-g` flag, do not set a breakpoint on an inline member function; it may confuse the debugger.
- When you use the debugger to display virtual and inherited class information, the debugger does not support pointers to members of a class.
- The debugger does not support calling the C++ constructs `new` and `delete`. As alternatives, use the `malloc( )` and `free( )` routines from C.
- Sometimes the debugger does not see class type names with internal linkage, and it issues an error message stating that the name is overloaded.
- You cannot select a version of an overloaded function that has a type signature containing ellipsis points (...). Pointers to functions with type signatures that contain parameter list or ellipsis arguments are not supported.

Limitations for debugging templates include:

- You cannot specify a template by name in a debugger command. You must use the name of the instantiation of the template.
- In C++, expressions in the instantiated template name can be full constant expressions, such as `stack<double,f*10>`. This form is not yet supported in the Digital Ladebug debugger; you must enter the value of the expression (for example, if `f` is 10 in the stack example, you must enter 100).
- Setting a breakpoint at a line number that is inside a template function will not necessarily stop at all instantiations of the function within the given file, but only a randomly chosen few. This limitation is due to the limited symbol information generated by the compiler for templates.

# 11 Debugging DEC Fortran and DEC Fortran 90 Programs

## 11.1 Significant Supported Features

You can use Ladebug to debug Fortran programs on the Digital UNIX operating system. The following features are supported:

- Much of the Fortran language syntax is built into the Ladebug debugger. You can specify the following language elements to the debugger using Fortran language syntax, including case insensitivity:
  - Type names
  - Identifiers
  - Program names
  - Subroutine names
  - Array sections
  - Relational operators
  - Logical operators

  It is an error to specify a trailing underscore, as Ladebug removes trailing underscores when reading a Fortran symbol table.
- You can control the execution of individual source lines in a program.
- You can set stops (breakpoints) at specific source lines, routine names, or under various conditions.
- You can refer to program locations by their symbolic names, using the debugger's knowledge of the DEC Fortran language to determine the proper scoping rules and how the values should be evaluated and displayed.

- You can examine and print the values of variables and set a trace (tracepoint) to notify you when the value of a variable changes. You can change the value of variables within the debugging environment.
- You can perform other functions, such as examining `core` files, examining the call stack, or displaying registers.
- You can debug programs with alternate entry points (see Section 11.5).
- You can debug mixed-language programs (see Section 11.6).
- You can debug programs with optimized code (see Section 11.8).

This chapter also discusses flags used with the Fortran compiler commands,† Fortran data types in Ladebug, limitations on Ladebug support for Fortran, debugging a Fortran program that generates an exception, and locating unaligned data.

## 11.2 Fortran Flags for Debugging

To prepare to use the Ladebug debugger on a Fortran program, invoke the compiler with the appropriate debugging flag: `-g`, `-g2`, or `-g3` and, for DEC Fortran 90 only, `-ladebug`. For example, for DEC Fortran 90:

```
% f90 -g -ladebug -o squares squares.f90
```

For DEC Fortran:

```
% f77 -g -o squares squares.for
```

This `f77` command compiles and links the program `squares.for` without optimization but with symbol table information needed for symbolic debugging with Ladebug. The executable file is named `squares` instead of `a.out`.

The `-g`*n* flags control the amount of information placed in the object file for debugging.

For Digital UNIX Version 3.2 systems, Table 11–1 summarizes the information provided by the debugger-related flags and their relationship to the `-O`*n* flags, which control optimization. Refer to your language compiler documentation for the latest information on compiler flags used on Digital UNIX Version 4.0 systems.

† `f77` is the DEC Fortran compile command, and `f90` is the DEC Fortran 90 compile command. In this chapter, *Fortran* refers to both DEC Fortran and DEC Fortran 90.

**Note**

Ladebug cannot distinguish between a Fortran-77 and a Fortran-90 compilation unit because the respective compilers do not make such distinctions. The value of `$lang` is always "Fortran" in both cases.

---

**Table 11–1 Summary of Symbol Table Flags**

| Flag | Traceback Information | Debugging Symbol Table Information | Effect on -O*n* Flags |
|---|---|---|---|
| `-g0` | No | No | Default is `-O4` (full optimization). |
| `-g1` (default) | Yes | No | Default is `-O4` (full optimization). |
| `-g2` or `-g` | Yes | Yes. For unoptimized code only. | Changes default to `-O0` (no optimization). |
| `-g3` | Yes | Yes | Default is `-O4`. |
| `-ladebug (f90 only)` | Yes | Describes Fortran-90 arrays in a manner that Ladebug understands. | No effect. |

Traceback information and symbol table information are both necessary for debugging. They enable the debugger to:

- Translate virtual addresses into source program routine names and compiler-generated line numbers
- Provide symbol definitions for:
  - User-defined variables
  - Arrays (including dimension information)
  - Structures
  - Labels of executable statements

Typical uses of the debugging flags at the various stages of program development are as follows:

- During the early stages of program development, use the `-g` (or `-g2`) flag, perhaps specifying `-O0` (which is the default with the `-g` or `-g2` flag) to enable debugging and to create unoptimized code.
- During the later stages of program development, use `-g0` or `-g1` to minimize the object file size† and, as a result, the memory needed for program execution, usually with optimized code.

  If further debugging is needed, compile and link using `-g` to create an unoptimized debugging version of the same application for debugging purposes.
- During the later stages of program development, if you need to debug optimized code, use `-g3`.

Traceback and symbol table information result in a larger object file. When you have finished debugging your program, you can remove traceback information with the the `strip` command (see `strip(1)` ). To remove symbol table information, you can compile and link again with `-g0` or `-g1` to create a new executable program.

If your program generates an exception, recompile using the `-fpe`*n* flag (see Section 11.7).

For more information on program development and run-time environments, see the DEC Fortran or DEC Fortran 90 user manual.

## 11.3 Displaying Fortran Variables

When the `$lang` debugger variable is set to Fortran, command names and program identifiers are case insensitive. For more information about the `$lang` debugger variable, see Section 11.6.

To refer to a variable named `J`, use either the uppercase letter `J` or its lowercase equivalent, `j`.

You can also enter Ladebug command names in uppercase or lowercase. For example, the following two commands are equivalent:

```
(ladebug) PRINT J
```

```
(ladebug) print j
```

† `-g1` results in a larger object file than `-g0` but smaller than the other `g`*n* flags

### 11.3.1 Fortran Common Block Variables

You can display the values of variables in a Fortran common block using Ladebug commands such as `print` or `whatis`.

To display the entire common block, use the common block name. For example:

```
(ladebug) list 1,11
      1       PROGRAM EXAMPLE
      2
      3       INTEGER*4 INT4
      4       CHARACTER*1 CHR
      5       COMMON /COM_STRA/ INT4, CHR
      6
      7       CHR = 'L'
      8
>     9       PRINT *, INT4, CHR
     10
     11       END
(ladebug) print com_stra
COMMON
    INT4 = 0
    CHR = "L"
```

To display a specific variable in the common block, use only the field name. For example:

```
(ladebug) PRINT CHR
"L"
```

If your program contains a common block and a member of the same name, the common block will be occluded by the member.

### 11.3.2 Fortran Derived-Type Variables

Variables in a Fortran 90 derived-type (TYPE statement) are represented in Ladebug commands, such as `print` or `whatis`, using Fortran 90 syntax form.

For derived-type structures, use:

- The derived-type variable name
- A percent sign (%)
- The member name

For example:

```
(ladebug) list 3,11
      3       TYPE X
      4       INTEGER A(5)
      5       END TYPE X
      6
      7       TYPE(X) Z
      8
      9       Z%A = 1
     10
>    11       PRINT*, Z%A
(ladebug) print Z%A
(1) 1
(2) 1
(3) 1
(4) 1
(5) 1
```

### 11.3.3 Fortran Record Variables

Variables in a Fortran record structure (STRUCTURE statement) are represented in a fashion similar to derived-type.

For record structures, use:

- The record name
- A percent sign (%) or a period (.)
- The field name

For example:

```
(ladebug) l 3,13
      3       STRUCTURE /STRA/
      4          INTEGER*4 INT4
      5          CHARACTER*1 CHR
      6       END STRUCTURE
      7
      8       RECORD /STRA/ REC
      9
     10       REC.CHR = 'L'
     11       REC.INT4 = 6
     12
>    13       PRINT *, REC.CHR, REC.INT4
(ladebug) print rec%int4
6
(ladebug) print rec.int4
6
```

To view all fields in the record structure, type the name of the record structure, such as `rec` (instead of `rec%int4` or `rec.int4` in the previous example).

### 11.3.4 Fortran Array Variables

For array variables, put subscripts within parentheses, as with Fortran source statements. For example:

```
(ladebug) assign arrayc(1)=1
```

You can print out all elements of an array using its name. For example:

```
(ladebug) print arrayc
```

```
(1) 1
(2) 0
(3) 0
```

Avoid displaying all elements of a large array. Instead, display specific array elements or array sections. For example, to print array element arrayc(2):

```
(ladebug) print arrayc(2)
```

```
(2) 0
```

#### 11.3.4.1 Array Sections

Fortran provides an array notation known as an array section. Array sections consist of a three parts, a starting element, an ending element, and a stride. For more information on arrays, see the DEC Fortran or DEC Fortran 90 user manual.

Consider the following array declarations:

```
INTEGER, DIMENSION(0:99)    :: arr
INTEGER, DIMENSION(0:9,0:9)  :: TenByTen
```

Assume that each array has been initialized to have the value of the index in each position, for example, TenByTen(5,5) = 55, arr(43) = 43. The following expressions will be accepted by the debugger:

```
(ladebug) print arr(2)
```

```
2
(ladebug) print arr(0:9:2)
```

```
(0) = 0
(2) = 2
(4) = 4
(6) = 6
(8) = 8
(ladebug) print TenByTen(:,3)
```

```
(0,3) = 3
(1,3) = 13
(2,3) = 23
(3,3) = 33
(4,3) = 43
(5,3) = 53
(6,3) = 63
(7,3) = 73
(8,3) = 83
(9,3) = 93
```

The only operations permissible on array sections are `whatis` and `print`.

Ladebug supports the array section notation for both Fortran-90 and Fortran-77 programs.

#### 11.3.4.2 Assignment to Arrays

Ladebug does not support assignments to whole arrays, only single array elements.

### 11.3.5 DEC Fortran 90 Module Variables

To refer to a variable defined in a module, insert a dollar sign ($), the module name, and another dollar sign ($) before the variable name. For example, with a variable named J defined in a module named modfile (statement MODULE MODFILE), enter the following command to display its value:

```
(ladebug) list 5,7
        5     USE MODFILE
        6     INTEGER*4 J
        7     CHARACTER*1 CHR
```

### 11.3.6 DEC Fortran 90 Pointer Variables

DEC Fortran 90 supports two types of pointers:

- Fortran 90 pointers (standard-conforming)
- DEC Fortran CRAY-style pointers (extension to the Fortran 90 standard)

The following example shows Fortran 90 pointers displayed in their corresponding source form with a `whatis` command. In the following example, the `-ladebug` switch is used to allow display of pointers to arrays. Only the display of pointers is currently supported; it is not currently possible to change the location to which the pointer points:

```
% f90 -g -ladebug ptr.f90
% ladebug ./a.out
Welcome to the Ladebug Debugger Version x.y-zz
------------------
object file name: ./a.out
Reading symbolic information ...done
(ladebug)stop in ptr
[#1: stop in ptr ]
(ladebug) list 1:13
      1       program ptr
      2
      3       integer, target :: x(3)
      4       integer, pointer :: xp(:)
      5
      6       x = (/ 1, 2, 3/)
      7       xp => x
      8
      9       print *, "x = ", x
     10       print *, "xp = ", xp
     11
     12       end
(ladebug) run
[1] stopped at [ptr:6 0x120001838]
      6       x = (/ 1, 2, 3/)
(ladebug) whatis x
integer*4 x (1:3)
```

Since xp has not been assigned to point to anything yet, it is still a generic pointer:

```
(ladebug) whatis xp
integer*4 (:) xp
(ladebug) S
stopped at [ptr:7 0x120001880]
      7       xp => x
(ladebug)
stopped at [ptr:9 0x120001954]
      9       print *, "x = ", x
(ladebug)
 x =            1           2           3
stopped at [ptr:10 0x1200019c8]
     10       print *, "xp = ", xp
(ladebug)
 xp =           1           2           3
stopped at [ptr:12 0x120001ad8]
     12       end
```

Now that xp points to x, it takes on the size, shape and values of x:

```
(ladebug) whatis xp
integer*4 xp (1:3)
(ladebug) print xp
(1) 1
(2) 2
(3) 3

(ladebug) quit
%
```

The following example shows DEC Fortran CRAY-style pointers displayed in their corresponding source form with a `whatis` command:

```
% f90 -g -ladebug cray.f90
% ladebug ./a.out
Welcome to the Ladebug Debugger Version x.y-zz
------------------
object file name: ./a.out
Reading symbolic information ...done
(ladebug) stop at 14
[#1: stop at "cray.f90":14 ]
(ladebug) run
[1] stopped at [cray:14 0x1200017e4]
     14       end
(ladebug) whatis p
real*4 (1:10) pointer p
(ladebug) print p
0x140002c00 = (1) 10
(2) 20
(3) 30
(4) 40
(5) 50
(6) 60
(7) 70
(8) 80
(9) 90
(10) 100
```

```
(ladebug) l 1:14
      1       program cray
      2
      3       real i(10)
      4       pointer (p,i)
      5
      6       n = 5
      7
      8       p = malloc(sizeof(i(1))*n)
      9
     10       do j = 1,10
     11          i(j) = 10*j
     12       end do
     13
>    14       end
(ladebug) quit
```

### 11.3.7 Complex Variable Support

Ladebug supports COMPLEX, COMPLEX*8, and COMPLEX*16 variables and constants in expressions.

Consider the following Fortran program:

```
 PROGRAM complextest
    COMPLEX*8 C8 /(2.0,8.0)/
    COMPLEX*16 C16 /(1.23,-4.56)/
    REAL*4     R4 /2.0/
    REAL*8     R8 /2.0/
    REAL*16    R16 /2.0/
    integer*2  i2 /2/
    integer*4  i4 /2/
    integer*8  i8 /2/

    TYPE *, "C8=", C8
    TYPE *, "C16=", C16

    end
```

Ladebug supports basic arithmetic operators, display and assignment on variables and constants of the COMPLEX type. For example:

```
Welcome to the Ladebug Debugger Version x.y-zz
------------------
object file name: complex
Reading symbolic information ...done
(ladebug) stop in complextest
[#1: stop in complextest ]
(ladebug) run
[1] stopped at [complextest:15 0x1200017b4]
     15       TYPE *, "C8=", C8
(ladebug) whatis c8
complex c8
(ladebug) whatis c16
double complex c16
(ladebug) print c8
(2, 8)
(ladebug) print c16
(1.23, -4.56)
(ladebug) whatis (-5,6.78E+12)
double complex
(ladebug) print (c8*c16)/(c16*c8)
(1, 0)
(ladebug) a c16=(-2.3E+10,4.5e-2)
(ladebug) print c16
(-23000000512, 0.04500000178813934)
(ladebug)
```

## 11.4 Limitations on Ladebug Support for Fortran

Ladebug and the Digital UNIX operating system support the Fortran language with certain limitations, that are described in the following sections.

Be aware of the following data-type limitations when you debug a Fortran program:

- Ladebug does not set breakpoints correctly on alternate entry points for DEC Fortran 90 Version 1.2 or earlier, or for DEC Fortran versions prior to Version 3.5 (see Section 11.5).
- Ladebug does not allow setting a breakpoint on a program routine named MAIN.
- Variables that have spellings that match C language type names must be entered in uppercase to distinguish them from the C language built in types.
- Substring notation is not supported.

The following limitations apply only to DEC Fortran 90:

- DEC Fortran 90 array constructors, structure constructors, adjustable arrays, and vector subscripts are not yet supported.

- DEC Fortran 90 user-defined (derived) operators are not yet supported.
- Ladebug does not handle variables of STR16 data types.
- A main program routine, without the program statement, will not have its line number information interpreted correctly. A breakpoint set on such a main program will get set one line later than is expected. For example:

  ```
  % cat noprog.f90
          i = 5
          end
  ```

  A breakpoint at main$noprog will get set at line 2, rather than line 1.

## 11.5 Use of Alternate Entry Points

If a subprogram uses alternate entry points (ENTRY statement within the subprogram), Ladebug handles alternate entry points as a separate subprogram, including:

- Use of the ENTRY statement name as a breakpoint (`stop in` command).
- Use of the `where` command at an alternate entry point breakpoint location.

For example, the following is a fragment of a Fortran program and a debugging session:

```
program aep

call foo(1,2,3,4)
call bar(4,5,6)
end

subroutine foo(I,J,K,I1)
INTEGER*4 I,J,K,L,I1

write (6,*) 'Entered via foo: ', i, j, k, I1
write (6,*) '*****************************'
goto 1000

entry bar(J,K,L)

write (6,*) 'Entered via bar: ', j, k, l
write (6,*) '*****************************'

1000 write (6,*) 'Exiting foo & bar'
return
end
```

```
Welcome to the Ladebug Debugger Version x.y-zz
------------------
object file name: aep
Reading symbolic information ...done
(ladebug) stop in foo
[#1: stop in foo ]
(ladebug) stop in bar
[#2: stop in bar ]
(ladebug) status
#1 PC==0x120001868 in foo "aep.f":10 { break }
#2 PC==0x1200019ac in bar "aep.f":16 { break }
(ladebug) run
[1] stopped at [foo:10 0x120001868]
     10      write (6,*) 'Entered via foo: ', i, j, k, I1
(ladebug) where
>0  0x120001868 in foo(I=1, J=2, K=3, I1=4) aep.f:10
#1  0x120001814 in aep() aep.f:3
#2  0x1200017b0 in main() for_main.c:203
(ladebug) c
Entered via foo:           1           2           3           4
*****************************
Exiting foo & bar
[2] stopped at [bar:16 0x1200019ac]
     16      write (6,*) 'Entered via bar: ', j, k, l
(ladebug) where
>0  0x1200019ac in bar(J=4, K=5, L=6) aep.f:16
#1  0x12000182c in aep() aep.f:4
#2  0x1200017b0 in main() for_main.c:203
(ladebug) c
Entered via bar:           4           5           6
*****************************
Exiting foo & bar
Thread has finished executing
(ladebug) quit
```

## 11.6 Debugging Mixed-Language Programs

The Ladebug debugger lets you debug mixed-language programs. Program flow of control across subprograms written in different languages is transparent.

The debugger automatically identifies the language of the current subprogram or code segment on the basis of information embedded in the executable file. For example, if program execution is suspended in a subprogram in Fortran, the current language is Fortran. If the debugger stops the program in a C function, the current language becomes C. The current language determines for the debugger the valid expression syntax and the semantics used to evaluate an expression.

The debugger sets the `$lang` variable to the language of the current subprogram or code segment. By manually setting the `$lang` debugger variable, you can force the debugger to interpret expressions used in commands by the rules and semantics of a particular language. For example, you can check the current setting of `$lang` and change it as follows:

```
(ladebug) print $lang
"C++"
(ladebug) set $lang = "Fortran"
```

When the debugger reaches the end of your program, the `$lang` variable is set to the language of the main program.

## 11.7 Debugging a Program That Generates an Exception

If your program encounters an exception at run time, to make it easier to debug the program, recompile and relink with the following `f90` flags or `f77` flags *before* debugging the cause of the exception:

- Use the `-fpe`*`n`* flag to control the handling of exceptions (see `f77(1)` or `f90(1)` ).
- As with other debugging tasks, use the `-g` flag to generate sufficient symbol table information and debug unoptimized code (see Section 11.2).

Use the Ladebug commands `catch` and `ignore` to control whether Ladebug displays and handles exceptions (catches them), or ignores exceptions so that they are handled by the Fortran run-time library.

To obtain the appropriate Fortran run-time error message when debugging a program that generates an exception, you may need to use the appropriate `ignore` command before running the program. For instance, use the following command to tell Ladebug to ignore floating-point exceptions and pass them through to the Fortran run-time library:

```
(ladebug) ignore fpe
```

Because the `where` command works only if the debugger is catching the signal, the `ignore` command prevents the `where` command from working. You may want to use the `where` command when an exception occurs to locate the part of the program causing the error. You need to consider this factor when you use the `ignore` command.

## 11.8 Debugging Optimized Programs

The Fortran compiler performs code optimizations (`-O4`) by default unless you specify `-g2` (or `-g`). See the DEC Fortran or DEC Fortran 90 user manual for a discussion of compiler optimizations.

Debugging optimized code is recommended only under special circumstances (such as a problem that shows up in an optimized program may disappear when you specify the `-O0` flag). Before you attempt to debug optimized code, read Section 11.2.

One aid to debugging optimized code is the `-show code` flag. This flag generates a listing file that shows the compiled code produced for your program. By referring to a listing of the generated code, you can see exactly how the compiler optimizations affected your code. This lets you determine the debugging commands you need in order to isolate the problem.

Another aid is a set of messages displayed by Ladebug when you try to perform a Ladebug operation on a language construct that has been optimized. For example, if the Fortran compiler can determine that an entire Fortran 90 statement is not needed for correct operation of the program (such as an unnecessary CONTINUE statement), that statement is not represented in the object code. As a result, Ladebug will use the next available line.

For more information on optimizations, see the DEC Fortran or DEC Fortran 90 user manual.

# 12 Debugging DEC Ada Programs

## 12.1 Significant Supported Features

You can use Ladebug to debug Ada programs on the Digital UNIX operating system. Supported features include the following:

- Debugging multilanguage programs
- Using case-insensitive commands and variable names
- Printing ISO Latin-1 characters
- Displaying the source code of generic units
- Debugging multiple units in one source file
- Debugging elaboration code
- Accessing unconstrained array types
- Accessing incomplete types completed in another compilation unit

Some of these features are further discussed in this chapter. The chapter also explains Ada compiler options, debugging limitations, and specific debugging tasks.

## 12.2 Compiling and Linking for Debugging

To compile units for debugging, specify the `-g` compiler option on the `ada` command, as follows:

```
% ada -g reservations_.ada reservations.ada hotel.ada
```

The `-g` option automatically suppresses code optimization. Nonoptimized code is easier to debug (although it is possible to debug optimized code). Nonoptimized code more closely resembles your program as you wrote it.

Alternatively, if your program has been compiled once and then modified, you can recompile the program by entering the `amake` command, as follows:

```
% amake -C′ -g′ hotel
```

To link units for debugging, enter the `ald` command, as follows:

```
% ald -o hotel hotel
```

The `ald` command automatically copies debugging information into the executable file.

For information the `amake` and `ald` commands, see manpages `amake(1)` and `ald(1)`.

## 12.3 Debugging Multilanguage Programs

The debugger allows you to debug mixed-language programs. Program flow of control across functions written in different languages is transparent.

The debugger automatically identifies the language of the current function or code segment based on information embedded in the executable file. If program execution is suspended in an Ada function, the current language is Ada. If the program executes a C function, the current language becomes C.

The current language determines the valid expression syntax for the debugger. It also affects the semantics used to evaluate an expression.

The debugger sets the `$lang` variable to the language of the current function or code segment. By manually setting the debugger variable `$lang`, you can force the debugger to interpret expresssions used in commands by the rules and semantics of a particular language.

**Note**

C language syntax is used when `$lang` is set to Ada; and no thread-related commands are available.

## 12.4 Using Case-Insensitive Commands and Variable Names

When the `$lang` debugger variable is set to Ada, command names and program identifiers are case-insensitive.

The `$lang` variable is set to "Ada" at Ladebug startup if your program has been linked using the ald linker. In this case, the main routine appears in a pseudo-file with a file name of the following form:

```
ald_mmmmmmmm_nnnnnnnnnn.ada
```

Otherwise, `$lang` reflects the the language of the main routine. When the `$lang` debugger variable is set to a language other than Ada (which may occur if most, but not all, of your routines are written in languages other than Ada), command names and program identifiers follow the case conventions of the `$lang` setting.

## 12.5 Printing ISO Latin-1 Characters

The `$ascii` debugger variable allows you to print ASCII characters in the 7-bit ASCII character set or in the 8-bit Latin-1 character set (a superset of the 7-bit ASCII character set). The ISO Latin-1 standard calls for character representation in 8 bits (256 values), rather than 7 bits, so the extended character set can include additional characters, such as those commonly found in Western European languages. (This character set is not coextensive with the DEC Multinational Character Set, but is very similar.)

To choose a character set, define `$ascii` as follows:

- When `$ascii` is set to 1, Ladebug prints the 7-bit ASCII character set characters. All other alphanumerics appear in octal notation. (This is the default.)
- When `$ascii` is set to 0, Ladebug prints the Latin-1 character set characters. All other alphanumerics appear in octal notation.

The Ada CHARACTER data type can contain any of the Latin-1 characters.

## 12.6 Displaying the Source Code of Generic Units

The instantiation of a generic unit is the code corresponding to the generic unit itself. Ladebug displays the correct source code for the instantiation of a generic unit by recognizing the unique, argumented file names that the DEC Ada compiler records for generic units. For example:

```
generic                         -- declaration of generic unit
   type T is range <>;          --
procedure init (X: out T);      -- formal parameter within declaration

procedure init (X: out T) is
begin
   X := 0;
end;
```

```
with init;
procedure test is
      procedure int_init is new init(integer);
                           -- Ada instantiation of generic unit
      I: integer;
begin
      int_init(I);
end;
```

In this example, when you step into `int_init`, you will see the source code that corresponds to the generic procedure `init`:

```
X := 0;
```

You will not see source code corresponding to the procedure call:

```
int_init(I);
```

## 12.7 Debugging Multiple Units in One Source File

In Ada, unlike some other languages, the basic compilation unit need not correspond to a source file. For example:

```
generic                         -- first compilation unit in file
   type T is range <>;          --
procedure init (X: out T);      --

procedure init (X: out T) is    -- second compilation unit in file
begin                           --
   X := 0;                      --
end;                            --

with init;
procedure test is               -- third compilation unit in file
   procedure int_init is new init(integer);
   I: integer;                  --
begin                           --
   int_init(I);                 --
end;                            --
```

In earlier versions of Ladebug, each compilation unit yielded a separate object file and debug symbol table, while referring to the same source file. However, new naming conventions now provide for the renaming of the source files for better correlation with object files and debug symbol tables, as follows:

- First compilation unit: `program-name.ada`
- N compilation units: `program-name.ada~nnn~decada_XXXXXXXX`

You can expect to see these augmented file names when you enter commands that result in file-name output (for example, `file`, `whereis`, `where` ). This naming convention is a temporary solution and may change in future releases.

## 12.8 Debugging Ada Elaboration Code

In Ada, as in other languages, the initial entry point is the procedure `main`, but unlike other languages, `main` in Ada is not the top-level application procedure. Instead, the elaboration code, which contains initialization routines for Ada-specific constructs such as packages, is called from a fabricated `main` routine before the top-level application is called.

If you wish to debug elaboration code, you can set a breakpoint on `main`, step to elaboration code initialization routine calls, and step into these routines. To ignore elaboration code, you can set a breakpoint in the Ada subprogram.

## 12.9 Accessing Unconstrained Array Types

Accesses to unconstrained arrays are implemented as pointers to structures known as descriptors or dope vectors. For example:

```
procedure Dbg_30 is
   type A1 is access String;
   X1 : A1 := new String’("123");
begin
   null;
end;
```

When you enter the `print` command, the debugger displays the pointer address (the address of the first (`lo1`) component) and the values of the first and last components. For example:

```
(ladebug) p *X1
struct {
    pointer = 0x14000c620;
    lo1 = 1;
    hi1 = 3;
}
```

To examine individual components, use the dereferencing operator (->) as follows:

```
(ladebug) p X1->pointer[0]; p X1->pointer[2]
struct {
   value = ’1’;
}
struct {
   value = ’3’;
}
```

## 12.10 Accessing Incomplete Types Completed in Another Compilation Unit

The full type often appears in a different compilation unit than the access type, which makes values of these types difficult to examine. Except in cases where the full type is an array type, you can examine values by carefully setting scopes and explicit type conversion. For example:

```
package Tmp_Pkg is
   type A_T is private;
   X : A_T;
private
   type T;
   type A_T is access T;
end Tmp_Pkg;

package body Tmp_Pkg is
   type T is record C1, C2 : Integer; end record;
begin
   X := new T'(71,72);
end Tmp_Pkg;

with Tmp_Pkg;
procedure Tmp is
begin
   null;
   --
   -- (ladebug) whereis X
   -- "tmp_pkg_.ada"‘X
   --
   -- (ladebug) p "/c/project/aosf_ft2/tmp_pkg_.ada"‘X
   -- 0x14000c600
   --
   -- (ladebug) file tmp_pkg.ada
   --
   -- (ladebug) p * ( (T*) (0x14000c600) )
   -- struct {
   --         C1 = 71;
   --         C2 = 72;
   --     }
   --
end;
```

## 12.11 Limitations on Ladebug Support for DEC Ada

Ladebug and the Digital UNIX operating system support the DEC Ada language with certain limitations, which are described in the following sections.

### 12.11.1 Limitations for Expressions in Ladebug Commands

Expressions in Ladebug commands use C source language syntax for operators and expressions. Data is printed as the equivalent C data type.

Table 12–1 shows Ada expressions and the debugger equivalents.

**Table 12–1 Ada Expressions and Debugger Equivalents**

| Ada Expression | Debugger Equivalent |
|---|---|
| Name | See Section 12.4. |
| Binary operations and unary operations | Only integer, floating, and Boolean expressions are easily expressed. |
| a+b,-,* | a+b,-,* |
| a/b | a/b |
| a = b /= < <= > >= | a = = b != < <= > >= |
| a and b | a&&b |
| a or b | a \| \| b |
| a rem b | a%b |
| not (a=b) | !(a= =b) |
| –a | –a |
| Qualified expressions | None. There is no easy way of evaluating subtype bounds. |
| Type conversions | Only simple numeric conversions are supported, and the bounds checking cannot be done. Furthermore, float -> integer truncates rather than rounds.<br><br>integer -> float<br><br>`(ladebug) print (float) (2147483647)`<br>`2147483648.0`<br>`(ladebug) print (double) (2147483647)`<br>`2147483647.0` |
| Attributes | None, but if E is an enumeration type with default representations for the values then E'PRED(X) is the same as x-1. E'SUCC(X) is the same as x+1 |

(continued on next page)

**Table 12–1 (Cont.) Ada Expressions and Debugger Equivalents**

| Ada Expression | Debugger Equivalent |
|---|---|
| p.all | *p (pointer reference) |
| p.m | p -> m (member of an "access record" type) |

### 12.11.2 Limitations in Data Types

This section lists the limitation notes by data type. For more information on these types, with examples, see the *Developing Ada Programs on Digital UNIX Systems* manual. Also see the the DEC Ada release notes for detailed information on debugging.

#### All Types

The debugger, unlike the Ada language, allows out-of-bounds assignments to be performed.

#### Integer Types

If integer types of different sizes are mixed (for example, byte-integer and word-integer), the one with the smaller size is converted to the larger size.

#### Floating-Point Types

If integer and floating-point types are mixed in an expression, the debugger converts the integer type to a floating-point type.

The debugger displays floating-point values that are exact integers in integer literal format.

#### Fixed-Point Types

The debugger displays fixed-point values as real-type literals or as structures. The structure contains values for the sign and the mantissa. To display the structure's value, multiply the sign and mantissa values. For example:

```
procedure Tmp3 is
    type F is delta 0.1 range -3.0 .. 9.0;
    X : F := 1.0;
begin
    X := X+1.0;
end;
```

```
(ladebug) s
stopped at [Tmp3:5 0x1200023dc]
    5       X := X+1.0;
```

```
(ladebug) print X
struct {
    fixed_point, small = 0.625E-1 * mantissa = 32;
}
```

#### Enumeration Types

The debugger displays enumeration values as the actual enumeral or its position.

Enumeration values must be manually converted to `'pos` values before you can use them as array indices.

#### Array Types

The debugger displays string array values in horizontal ASCII format, enclosed in quotation ("x") marks. A single component (character) is displayed within single quotation ('x') marks.

The debugger allows you to assign a component value to a single component; you cannot assign using an entire array or array aggregate.

Arrays whose components are neither a single bit nor a multiple of bytes are described to the debugger as structures; a `print` command displays only the first component of such arrays.

#### Records

The debugger cannot display record components whose offsets from the start of the record are not known at compile time.

For variant records, however, the debugger can display the entire record object that has been declared with the default variant value. The debugger allows you to print or assign a value to a component of a record variant that is not active.

#### Access Types

The debugger does not support allocators, so you cannot create new access objects with the debugger. When you specify the name of an access object, the debugger displays the memory location of the object it designates. You can examine the memory location value.

### 12.11.3 Limitations for Tasking Programs

When you debug Ada tasking programs, you use the debugger and the DEC Ada `ada_debug` routine.

## 12.12 Debugging Programs That Generates an Exception

Ada exceptions can be raised in the following cases:

- By explicit raise `exception-name;` statements in program code
- By implicit language checks on, for example, range, bounds, and so on

When an exception occurs, you need to determine the following:

1. Where the exception is being raised
2. How the exception is being raised (by the `raise` statement or by language check)
3. Why the exception occurred, by either:
   - Examining the conditions of the `raise` statement.
   - Examining the local environment to further pinpoint the language check generation.

The following sections give more detail.

### Determining Where the Exception Is Being Raised

There are two ways to determine where an exception is being raised:

- Finding the value of the Program Counter (PC) for an instruction near where the exception was raised

  This method is not always helpful, especially if the code has been optimized. The raising of an exception terminates the current sequence of instructions, and the raising of some exceptions (for example, arithmetic traps) is delayed, so that the PC rests on an instruction just before or just after the instruction causing the exception.

- Using breakpoints to intercept the exception at the point where it is being raised

  You can intercept all exceptions being raised by your program by using the Ladebug command `stop in exc_dispatch_exception.` This sets a breakpoint in the fundamental run-time system routine that raises all exceptions.

  When Ladebug stops in this routine, a `where` command will show you the place where the exception was raised.

  ```
  (ladebug) where
  >0  0x120007f0c in Dbg_24a$ELAB(=0x14000e400, =0x14000e3c8) dbg_24a.ada:13
  ```

You can examine more of the adjacent instructions by using the <expression>/i command.

```
(ladebug) 0x120007f0c-4 /2i
[Dbg_24a$ELAB:13, 0x120007f08] ornot   zero, zero, t1
*[Dbg_24a$ELAB:13, 0x120007f0c] srl     t1, 0x21, t1
```

If you intercept an exception other than the one you intended to catch, you can continue the program's execution and catch the next one. If there are many exceptions before the one you wish to catch, you need to set a breakpoint closer to the raising of the exception, then get to it before starting to intercept exceptions.

#### Determining How the Exception Is Being Raised

Having determined where the exception is occurring, examine your program code to see whether an explicit `raise` statement has caused the exception. If a `raise` statement does not appear, then the exception is the result of an Ada language check.

#### Determining Why the Exception Is Being Raised

If the exception is raised by an explicit `raise` statement, examine your code to determine why the `raise` statement was executed.

If the exception is raised by a language check, see *Developing Ada Programs on Digital UNIX Systems* for tips on pinpointing the error.

## 12.13 Debugging Optimized Programs

The DEC Ada compiler performs code optimizations by default, unless you specify the `-g` flag. Debugging optimized code is recommended only if unoptimized code is unavailable. It is extremely difficult to understand your program by examining the workings of its optimized form.

If you must debug optimized code, then note the following changes that optimization may make in your source code:

- The instructions for a source line may be interspersed with those for other source lines from distant locations.
- Some of the instructions for several source lines may have been merged.
- Variables may be assigned multiple different storage locations or registers.
- Variables may have some or all of their fetches and stores moved or eliminated.

- Subprogram calls may have been replaced with inlined instructions, so setting a breakpoint in them may not catch all calls in the source. The inlined subprograms will not show up in the call stack.

You may wish to make use of the following aids when debugging optimized code:

- Compiled-in debugging support, such as dump subprograms, assertions, and Text_IO.Put. These entities print correct values, and examining their output is easier than examining variables within optimized code.
- The `volatile` pragma. This pragma suppresses some of the optimization associated with a particular variable, forcing its location or its associated fetch and store instructions to be more predictable.

# 13
# Debugging DEC COBOL Programs

## 13.1 Significant Supported Features

To help you debug DEC COBOL programs on the Digital UNIX operating system, Ladebug supports:

- Much of the COBOL language syntax is built into the Ladebug debugger. You can specify the following language elements to the debugger with COBOL language syntax:
  - Identifiers, including subscripting and qualification with some limitations (see Section 13.6)
  - Numeric and nonnumeric literals
  - Arithmetic expressions
- You can examine the values of all COBOL data types with the debugger's `print` command.
- You can assign new values to all data types with the debugger's `assign` command, including numeric literals and other program items. However, there are some limitations to assignment (see Section 13.5).
- Simple arithmetic operations are supported, but full COBOL expression evaluation is not (see Section 13.6). For example, the following simple addition can be done:

  ```
  (ladebug) print itema + itemb
  ```

The following features are also supported for DEC COBOL debugging:

- Scoping and symbol lookup
- Numeric edited items (with limitations; see Section 13.5)
- Scaled binary (COMP) items
- All decimal types
- Both signed and unsigned items

- COBOL-specific initial debugger context
- Multiple groups
- COBOL-specific group/table examination of data items with the `print` command
- Mixed-language programs (see Section 13.4)

Some of the features are further described in this chapter. This chapter also:

- Explains the flags you use on the DEC COBOL compiler command line to enable debugging.
- Describes the debugger's support for DEC COBOL identifiers.
- Lists limitations on debugging DEC COBOL programs.

## 13.2 DEC COBOL Flags for Debugging

To use the Ladebug debugger on a COBOL program, invoke the COBOL compiler with the appropriate debugging flag: `-g`, `-g2`, or `-g3`. For example:

```
% cobol -g -o sample sample.cob
```

The `-g` flag on the compiler command instructs the compiler to write the program's debugger symbol table into the executable image. This flag also turns off optimization; optimization (which is the default for nondebugger compilations) could cause a confusing debugging session.

For Digital UNIX Vesion 3.2 systems, Table 13–1 summarizes the information provided by the `-g`*n* flags and their relationship to the `-O`*n* flags, which control optimization. Refer to your language compiler documentation for information about compiler flag defaults for Digital UNIX Version 4.0.

**Table 13–1 Summary of Symbol Table Flags**

| Flag | Traceback Information | Debugging Symbol Table Information | Effect on -O*n* Flags |
|---|---|---|---|
| `-g0` | No | No | Default is `-O4` (full optimization). |
| `-g1` (default) | Yes | No | Default is `-O4` (full optimization). |
| `-g2` or `-g` | Yes | Yes. For unoptimized code only. | Changes default to `-O0` (no optimization). |
| `-g3` | Yes | Yes. Use with optimized code. Inaccuracies may result. | Default is `-O4` (full optimization). |

If you specify `-g`, `-g2`, or `-g3`, the compiler provides symbol table information for symbolic debugging. The symbol table allows the debugger to translate virtual addresses into source program routine names and compiler-generated line numbers.

Later, to remove this symbol table information, you can compile and link it again (without the `-g` flag) to create a new executable program or use the `strip` command (see `strip(1)` ) on the existing executable program.

The `-g2` or `-g` flag provides symbol table information for symbolic debugging unoptimized code. If you use the `-g2` or `-g` flag and do not specify an `-O`*n* flag, the default optimization level changes to `-O0` (in all other cases, the default optimization level is `-O4`) . If you use this flag and specify an `-O`*n* flag other than `-O0`, a warning message is displayed. For example:

```
% cobol -g -O sample sample.cob
cobol: Warning: file not optimized; use -g3 for debug with optimize
%
```

The `-g3` flag is for symbolic debugging with optimized code.

Typical uses of the debugging flags at the various stages of program development are as follows:

- During early stages of program development, use the `-g` (or `-g2`) flag, perhaps specifying `-O0` (which is the default with `-g` or `-g2`) to enable debugging and create unoptimized code. This flag also might be chosen later to debug reported problems from later stages.
- During the middle stages of program development, use optimized code with `-g3`. This flag might also be used during later stages to debug reported problems. (Certain problems may be reproducible only when the code is optimized, but be aware that debugging inaccuracies can result from the optimization.)
- During the later and postrelease stages of program development, use `-g0` to minimize the object file size and, as a result, the memory needed for program execution, with fully optimized code for best performance.

## 13.3 Support for COBOL Identifiers

Ladebug supports the case insensitivity of COBOL identifiers and the hyphen-underscore exchanges made by the DEC COBOL compiler.

In case-insensitive languages (such as COBOL, Fortran, and Ada), you can enter identifiers in either uppercase or lowercase, or a combination of both. For example, the following are all legal and equivalent identifiers:

```
Identifier IDENTIFIER identifier IdEnTiFiEr
```

Ladebug and the DEC COBOL compiler treat all forms of identifers as references to the same object.

The DEC COBOL compiler performs transformations on identifiers with regard to both case and occurrences of hyphens and underscores. These transformed identifiers are visible in the listing file and in the symbol table of an image compiled with the `-g` flag. The rules for transformations are as follows:

- For an externally visible name (one that is explicitly declared as EXTERNAL), except for program IDs, the following transformations occur:
  - All lowercase letters are replaced by their uppercase equivalents.
  - All occurrences of the hyphen (-) are replaced by an underscore (_).
- For a locally visible name:
  - All lowercase letters are replaced by their uppercase equivalents.
  - All occurrences of the underscore (_) are replaced by hyphen (-).
- For any program ID:
  - All uppercase letters are replaced by their lowercase equivalents.
  - All occurrences of the hyphen (-) are replaced by underscore (_).

Ladebug transforms all identifiers according to rule 2. When such a transformation causes a namespace conflict, an identifier is considered *overloaded*. When overloading occurs, it is necessary that you qualify an identifier to make it unique, as shown in Example 13–1, which demonstrates the application of the rules for transformation. Example 13–2 shows how Ladebug handles the COBOL identifiers.

**Example 13–1 Sample COBOL Program**

```
example.cob:
* Ladebug Version 4.0
*
* Demonstrates usage of COBOL expressions with Ladebug
*
* There are three procedures in this file, namely cobol_example,
* overloaded_name and b-2.
*
* (Rule #1)
* In cobol_example, note the symbols B-2 and C_3: these two are given as
* external and appear in the symbol table as B_2 and C_3 respectively.
* C-3 and C_3 are equivalent as are D-3 and D_3
*
* (Rule #2)
* In the procedure COBOL_EXAMPLE is the symbol overloaded-name, which appears
* in the symbol table as OVERLOADED-NAME
*
* (Rule #3)
* The procedure OverLoaded-NAME appears in the symbol table as
* overloaded_name.

* The procedure b-2 appears in the symbol table as b_2
*
* Note that there are three names referred to as B-2:
*
* "example.cob"‘cobol_example‘B_2    -- PIC X external
* "example.cob"‘overloaded_name‘B-2  -- PIX 99
* "example.cob"‘b_2                  -- program id
*

IDENTIFICATION DIVISION.
PROGRAM-ID. COBOL_EXAMPLE.          1

DATA DIVISION.
WORKING-STORAGE SECTION.
01 A_1 PIC X VALUE IS "1".
01 B-2 PIC X             external.  2
01 C_3 PIC X             external.
01 D-4 PIC X VALUE IS "4".
01 overloaded-name pic 99.          3
```

(continued on next page)

**Example 13–1 (Cont.) Sample COBOL Program**

```
PROCEDURE DIVISION.
P0-lab.
 DISPLAY "*** Ladebug COBOL Example ***".
 MOVE "2" TO B_2.
 MOVE "3" TO C-3.
 DISPLAY "A_1 = " A_1.
 DISPLAY "B-2 = " B-2.
 DISPLAY "C_3 = " C_3.       4
 DISPLAY "C-3 = " C-3.
P0_lab.
 DISPLAY "D-4 = " D-4.       5
 DISPLAY "D_4 = " D_4.
 CALL "Overloaded-Name".     6
 CALL "B-2".
 DISPLAY "***END Ladebug COBOL Example***".
 STOP RUN.
end program cobol-example.



identification division.
program-id. OverLoaded-NAME.       6
data division.
working-storage section.
01 b_2 pic 99 value is 12.         7
procedure division.
beg1.
        display "*** Overloaded-Name ***".
 display "b_2 = " b-2.       7
 display "*** end of Overloaded-Name ***".
end program overloaded-name.



identification division.
program-id. b-2.                   8
data division.
working-storage section.
procedure division.
beg1.
```

(continued on next page)

**Example 13–1 (Cont.) Sample COBOL Program**

```
        display "*** b_2 ***".
 display "*** end of b_2 ***".
end program b-2.
```

1 The program ID `COBOL_EXAMPLE` is implicitly external and is emitted as `cobol_example`.

2 The use of `external` causes `B-2` to be emitted as `B_2` and `C_3` to be emitted as `C_3`.

3 This is the first occurrence of `overloaded-name`. Because it is a local symbol, it is emitted as `OVERLOADED-NAME`.

4 Both `C_3` and `C-3` refer to the same object, which is `C_3` in **2** .

5 Both `D-4` and `D_4` refer to the local symbol emitted as `D-4`.

6 This is the second occurrence of `overloaded-name`. Since it is a program ID, it is implicitly external and is emitted as `overloaded_name`.

7 This is the second occurrence of `b_2`. Because it is a local symbol, it is emitted as `B-2`.

8 This is the third occurrence of `b-2`. Since it is a program ID, it is implicitly external and is emitted as `b_2`.

The sample debugging session in Example 13–2 demonstrates how Ladebug handles the symbols from the sample COBOL program in Example 13–1.

**Example 13–2 Sample COBOL Debugging Session**

```
Welcome to the Ladebug Debugger Version 4.0
------------------
object file name: example
Reading symbolic information ...done

(ladebug) stop at 51
[#1: stop at "example.cob":51 ]
(ladebug) run
```

(continued on next page)

**Example 13–2 (Cont.) Sample COBOL Debugging Session**

```
*** Ladebug COBOL Example ***
A_1 = 1
B-2 = 2
C_3 = 3
C-3 = 3
D-4 = 4
D_4 = 4
[1] stopped at [cobol_example:51 0x12000cd14]
     51         CALL "Overloaded-Name".
(ladebug) whatis d-4   1
array [subrange 1 ... 1 of int] of char d-4
(ladebug) whatis D_4   1
array [subrange 1 ... 1 of int] of char D_4
(ladebug) print d_4    1
"4"
(ladebug) whereis d-4  1
"example.cob"‘cobol_example‘D-4

(ladebug) whereis b-2  2
"B_2"
B_2
"example.cob"‘cobol_example‘B_2
"example.cob"‘overloaded_name‘B-2
"example.cob"‘b_2
(ladebug) which b-2    3
"example.cob"‘cobol_example‘B_2
(ladebug) print b-2    4
2
(ladebug) step
stopped at [overloaded_name:61 0x12000ce04]
     61 program-id. OverLoaded-NAME.
(ladebug) stop at 78
[2] stopped at [cobol_example:78 0x12000cf20]

(ladebug) next
stopped at [overloaded_name:69 0x12000ce30]
     69         display "*** Overloaded-Name ***".
(ladebug) which b-2    5
"example.cob"‘overloaded_name‘B-2
(ladebug) whatis b-2
 pic 99 usage display b-2
(ladebug) print b-2
12
```

(continued on next page)

**Example 13–2 (Cont.) Sample COBOL Debugging Session**

```
(ladebug) return
*** Overloaded-Name ***
b_2 = 12
*** end of Overloaded-Name ***
stopped at [cobol_example:51 0x12000cd58]
     51         CALL "Overloaded-Name".
(ladebug) n
stopped at [cobol_example:52 0x12000cd68]
     52         CALL "B-2".
(ladebug) s
stopped at [b_2:78 0x12000cef4]
     78 program-id. b-2.
(ladebug) which b-2
"c_example6-2.cob"‘b_2
(ladebug) whatis c_example6-2.cob‘b_2
void b_2(void)

(ladebug) cont
*** b_2 ***
*** end of b_2 ***
***END Ladebug COBOL Example***
Thread has finished executing
(ladebug) q
```

1 Since there is only one object named `D-4`, all four spellings are resolved to the same address. The `whereis` command displays only one instance of `d-4`, namely `"example.cob"‘cobol_example‘D-4`.

2 There are three different instances of `b-2` in this example. The `whereis` command lists all three.

```
"example.cob"‘cobol_example‘B_2    -- PIC X external
"example.cob"‘overloaded_name‘B-2  -- PIX 99
"example.cob"‘b_2                  -- program id
```

3 Based on the current context, the `B_2 PIC X external` instance of `b-2` is the one visibile in the current scope.

4 Based on callout **3** , `b-2` refers to `"example.cob"‘cobol_example‘B_2` , which has a value of 2.

5 Since the current context is in the procedure `Overloaded-name`, the `which` command refers to the `B-2` local to `Overloaded-name` (`"example.cob"‘overloaded_name‘B-2‘`) .

For another example of debugging a COBOL program with Ladebug, see the appendix on tools in the *DEC COBOL User Manual*.

## 13.4 Debugging Mixed-Language Programs

The Ladebug debugger lets you debug mixed-language programs. The flow of control across programs written in different languages in your executable image is transparent.

The debugger automatically identifies the language of the current program or code segment on the basis of information embedded in the executable file. For example, if program execution is suspended in a code segment in COBOL, the current language is COBOL. If the program executes a C function, the current language becomes C. The current language determines for the debugger the valid expression syntax and the semantics used to evaluate an expression.

The debugger sets the `$lang` variable to the language of the current program or code segment. By manually setting the `$lang` debugger variable, you can force the debugger to interpret expressions used in commands by the rules and semantics of a particular language. For example, you can check the current setting of `$lang` and change it as follows:

```
(ladebug) print $lang
"C++"
(ladebug) set $lang = "Cobol"
```

When the debugger reaches the end of your program, the `$lang` variable is *not* set to the language of the `_exit` routine, which is written in machine code.

## 13.5 Limitations on Assignment

The following limitations apply to assignment in COBOL debugging:

- You cannot assign new values to character string items (PIC X or PIC A).
- The scales of the items in an assignment must match. Consider the following declarations:

  ```
  01 itema pic 9(9)v99.
  01 itemb pic 9999v99.
  01 bigitem pic 9(13)v9(5).
  ```

  The debugger allows assignment of the values 1.23 or 8765.22 to `itema`, but does not allow assignment of the value 1.2 to `itema`. The following debugger commands are supported because the quantities on both sides of the assignment operator (=) have the same scale:

  ```
  (ladebug) assign itema = 1.23
  (ladebug) assign itema = 8765.22
  (ladebug) assign itema = itemb
  ```

The following debugger commands are not supported because the quantities involved are of different scales:

```
(ladebug) assign itema = 1.2
(ladebug) assign bigitem = itema
```

- You cannot assign a value of greater precision to an item. Given the declarations, the following debugger command is not supported, because `itema` has greater precision than `itemb`:

  ```
  (ladebug) assign itemb = itema
  ```

- With numeric edited items, the precisions of the quantities on both sides of the assignment must be the same.
- A numeric edited item cannot be assigned to other numeric items.

## 13.6 Other Limitations

Other limitations when you debug COBOL programs include:

- Subordinate items must be fully qualified for the debugger to find them, and also must be specified in uppercase. Consider the following example:

  ```
  01 A.
    03 B.
       05 C pic 9.
  ```

  For this group, the debugger cannot find C unless you fully qualify it, as follows:

  ```
  C of B of a
  ```

- Qualification involving more than one intervening level produces a debugger error if the intervening level is an OCCURS item. For example, `C of B of a` for the group is supported, but not if `B` is an OCCURS item.

  For the transformation rules, see Section 13.3.

- Some Ladebug command usage is affected by COBOL language syntax when execution is stopped within a COBOL procedure. One effect is that expressions typed on the debugger command line must include spaces around arithmetic operators like "+" and "-".
- Another effect of COBOL language syntax is in the debugger memory-examine command. For example, to look at the next 10 program instructions, you would normally use:

```
(ladebug) ./10i
```

This debugger command says "from the current program location" (signified by the dot), examine the next 10 program locations (10 being the count) in instruction mode (signified by the "i").

When debugging COBOL programs, you need to enter this command as follows:

```
(ladebug) ./10 i
```

Add a space between the count (`10`) and the mode indicator (`i`) .

- Reference modification is not supported.
- The OF qualifier keyword is supported, but the IN keyword is not supported.
- Tables (OCCURS items) with variable upper bounds are not supported.
- Breakpoints on labels are not supported.
- Subscripting is supported, but only for tables of one dimension.
- Stopping program execution at a program label (paragraph or section name) is not supported.
- The debugger does not correctly evaluate all COBOL condition expressions.
- The debugger supports only simple arithmetic operations in COBOL. It does not support full expression evaluation, including exponentiation, reference modification, and arithmetic operations with operands of different scales.

# Part IV

## Advanced Topics

This part describes advanced debugging topics:

- Core files
- Shared libraries
- Programs with limited symbolic information
- Machine-level debugging
- Multithreaded applications
- Multiprocessing
- Remote debugging
- Kernel debugging

# 14 Debugging Core Files

When the operating system encounters an irrecoverable error running a program, the system creates a file named `core` and places it in the current directory. The core file is not an executable file; it is a snapshot of the state of your program at the time the error occurred. It allows you to analyze the program at the point it crashed. This chapter describes a technique you can use to debug core files using the Digital Ladebug debugger.

Kernel debugging can be very helpful in analyzing kernel crash dumps. For information on kernel debugging, see Chapter 22.

## 14.1 Invoking the Debugger on a Core File

You can use the debugger to examine the program information in a core file. Use the following `ladebug` command syntax to invoke the debugger on a core file:

```
ladebug   executable_file core_file
```

Specify the name of the executable file (the program that was running at the time the core file was generated) in addition to the core file.

## 14.2 Core File Debugging Technique

When debugging a core file, use the debugger to obtain a stack trace and the values of a few variables.

The stack trace lists the functions in your program that were active when the dump occurred. By examining the values of a few program variables along with the stack trace, you should be able to pinpoint the program state and the cause of the core dump. (Core files cannot be executed, and so a `rerun`, `step`, `cont`, etc. will not work until a `run` is issued.)

In addition, if the program is multithreaded, you can examine the native (that is, kernel-level) thread information with the `show thread` and `thread` commands. You can examine the stack trace for a particular thread or all threadswith the `where thread` command. (See Section 14.3. Also see Chapter 19.)

The program shown in Example 14–1 is almost identical to the program used in Chapter 7 but adds a null pointer reference in the `factorial` function. This reference causes the program to abort and dump the core when it is executed. The `dump` command prints the value of the `x` variable as a null, and the `print *x` command reveals that you cannot dereference a null pointer.

**Example 14–1 Debugging a Core File**

```
% cat testProgram.c
main() {
        int i,f;
        for (i=1 ; i<3 ; i++) {
                f = factorial(i);
                printf("%d! = %d\en",i,f);
        }
}
factorial(i)
int i;
{
int *x;
        x = 0;
        printf("%d",*x);
        if (i<=1)
                return (1);
        else
                return (i * factorial(i-1) );
}
% cc -o testProgram -g testProgram.c
% testProgram
Memory fault - core dumped.
% ladebug testProgram core
Welcome to the Ladebug Debugger Version 4.0
------------------
object file name: testProgram
core file name: core
Reading symbolic information ...done
Core file produced from executable testProgram
Thread terminated at PC 0x120000dc4 by signal SEGV
(ladebug) where
>0  0x120000dc4 in factorial(i=1) testProgram.c:13
#1  0x120000d44 in main() testProgram.c:4
```

(continued on next page)

**Example 14–1 (Cont.) Debugging a Core File**

```
(ladebug) dump
>0  0x120000dc4 in factorial(i=1) testProgram.c:13
printf("%d",*x);
(ladebug) print *x
Cannot dereference 0x0
Error: no value for *x
(ladebug)
```

## 14.3 Core Thread Debugging of Native Threads

Ladebug can debug core files generated by multithreaded applications. For this kind of core file debugging, Ladebug sets the `$threadlevel` variable to `native` in the following initializations:

- When Ladebug is invoked for core file debugging.
- When a `load` command is issued for core file debugging.
- When the `process` command is used for core file debugging.

**Note**

Ladebug will not set the `$threadlevel` variable correctly in the following case: Process A, not a core file, is attached with the `$threadlevel` set to `native`. Switching to Process B and then back to Process A resets the variable to `decthreads` rather than `native`.

Example 14–2 presents an example of core file kernel thread debugging.

**Example 14–2 Debugging a Multithreaded Kernel Core File**

```
Welcome to the Ladebug Debugger Version 4.0-10
------------------
object file name: ../bin/c_threadcore
Reading symbolic information ...done
(ladebug) record io out
(ladebug) unload
(ladebug) load ../bin/c_threadcore ../bin/c_threadcore_core
Reading symbolic information ...done
Core file produced from executable c_threadcore
Thread 0xffffffff836934d0 terminated at PC 0x3ff805065c8 by signal SEGV
1
(ladebug) print $threadlevel2
"native"
(ladebug) show thread3
  Thread #  Id                  State
> 1         0xffffffff836934d0  dead
  2         0xffffffff83596120  dead
(ladebug) thread4
Thread 0x1
(ladebug) where5
>0  0x3ff805065c8 in __kill(0xb, 0x2, 0x3, 0x47c18, 0x4, 0x6)
 ../../../../../src/usr/ccs/lib/libc/alpha/kill.s:41
#1  0x3ff8055c560 in exc_raise(0x3ffc01e7e78, 0x0, 0x1, 0x0, 0x309a9a07,
 0xb97c60) ../../../../../src/usr/ccs/lib/DECthreads/COMMON/
 exc_handling.c:649
#2  0x3ff8055c764 in exc_pop_ctx(0x3ffc01e7bd0, 0x0, 0x0, 0x3ff80567b40,
 0x0, 0x0) ../../../../../src/usr/ccs/lib/DECthreads/COMMON/
 exc_handling.c:805
#3  0x3ff8056e374 in cma__thread_base(0x0, 0x0, 0x0, 0x0, 0x0,
 0x1200136e0) ../../../../../src/usr/ccs/lib/DECthreads/COMMON/
 cma_thread.c:1623
(ladebug) thread 26
Thread 0x2
(ladebug) where7
>0  0x3ff8051df04 in msg_receive_trap(0xe56b0, 0x0, 0x3ff804b500c,
 0x100000000000, 0x3ff804b5058, 0x18157f0d0d) /usr/sde/osf1/build/
 goldminos.bld/export/alpha/usr/include/mach/syscall_sw.h:74
#1  0x3ff80514764 in msg_receive(0xe0df0, 0x0, 0x0, 0x240000000000000,
 0x61746164702e, 0x3ff800b9fe0) ../../../../../src/usr/ccs/lib/libmach
 /msg.c:95
#2  0x3ff80574b00 in cma__vp_sleep(0x2801000000, 0x3ff00000001, 0x6,
 0x3ffc00841a0, 0x3ff00000000, 0x3ffc00960c0) ../../../../../src/usr/
 ccs/lib/DECthreads/COMMON/cma_vp.c:1588
#3  0x3ff8055ae7c in cma__dispatch(0x6, 0x3ffc00841a0, 0x3ff00000000,
  0x3ffc00960c0, 0x3ff80554354, 0x3ffc0080468) ../../../../../src/usr/ccs/
```

(continued on next page)

**Example 14–2 (Cont.) Debugging a Multithreaded Kernel Core File**

```
 lib/DECthreads/COMMON/cma_dispatch.c:998
#4  0x3ff80554354 in cma__int_wait(0x11ffff428, 0x4ac18, 0x3ffc01de9f8,
 0x70000003, 0x3ff80485192, 0x3ff80089931) ../../../../../src/usr/ccs/
 lib/DECthreads/COMMON/cma_condition.c:2651
#5  0x3ff8056d704 in cma_thread_join(0x11ffff848, 0x11ffffbf0,
 0x11ffffbe8, 0x3ff8056e110, 0x0, 0x3ffc01e3de0) ../../../../../src/
 usr/ccs/lib/DECthreads/COMMON/cma_thread.c:955
#6  0x3ff80563a2c in pthread_join(0x47c18, 0x400000000000002,
 0x11ffffc68, 0x3ffc01de9f8, 0x120013d18, 0x5) ../../../../../src/usr/
 ccs/lib/DECthreads/COMMON/cma_pthread.c:2270
#7  0x120013d70 in main() thread_core.c:92
(ladebug) where thread all8
Stack trace for thread 1
#0  0x3ff805065c8 in __kill(0xb, 0x2, 0x3, 0x47c18, 0x4, 0x6) ../../../../
 ../src/usr/ccs/lib/libc/alpha/kill.s:41
#1  0x3ff8055c560 in exc_raise(0x3ffc01e7e78, 0x0, 0x1, 0x0, 0x309a9a07,
  0xb97c60)
../../../../../src/usr/ccs/lib/DECthreads/COMMON/exc_handling.c:649
#2  0x3ff8055c764 in exc_pop_ctx(0x3ffc01e7bd0, 0x0, 0x0, 0x3ff80567b40,
 0x0, 0x0) ../../../../../src/usr/ccs/lib/DECthreads/COMMON/
 exc_handling.c:805
#3  0x3ff8056e374 in cma__thread_base(0x0, 0x0, 0x0, 0x0, 0x0,
 0x1200136e0) ../../../../../src/usr/ccs/lib/DECthreads/
 COMMON/cma_thread.c:1623
Stack trace for thread 2
>0  0x3ff8051df04 in msg_receive_trap(0xe56b0, 0x0, 0x3ff804b500c,
 0x100000000000, 0x3ff804b5058, 0x18157f0d0d) /usr/sde/osf1/build/
 goldminos.bld/export/alpha/usr/include/mach/syscall_sw.h:74
#1  0x3ff80514764 in msg_receive(0xe0df0, 0x0, 0x0, 0x240000000000000,
 0x61746164702e, 0x3ff800b9fe0) ../../../../../src/usr/ccs/lib/libmach/
 msg.c:95
#2  0x3ff80574b00 in cma__vp_sleep(0x2801000000, 0x3ff00000001, 0x6,
 0x3ffc00841a0, 0x3ff00000000, 0x3ffc00960c0) ../../../../../src/usr/
 ccs/lib/DECthreads/COMMON/cma_vp.c:1588
#3  0x3ff8055ae7c in cma__dispatch(0x6, 0x3ffc00841a0, 0x3ff00000000,
 0x3ffc00960c0, 0x3ff80554354, 0x3ffc0080468) ../../../../../src/usr/
 ccs/lib/DECthreads/COMMON/cma_dispatch.c:998
#4  0x3ff80554354 in cma__int_wait(0x11ffff428, 0x4ac18, 0x3ffc01de9f8,
 0x70000003, 0x3ff80485192, 0x3ff80089931) ../../../../../src/usr/ccs/
 lib/DECthreads/COMMON/cma_condition.c:2651
#5  0x3ff8056d704 in cma_thread_join(0x11ffff848, 0x11ffffbf0,
 0x11ffffbe8, 0x3ff8056e110, 0x0, 0x3ffc01e3de0)
../../../../../src/usr/ccs/lib
#6  0x3ff80563a2c in pthread_join(0x47c18, 0x400000000000002,
 0x11ffffc68, 0x3ffc01de9f8, 0x120013d18, 0x5) ../../../../../src/usr/
#7  0x120013d70 in main() thread_core.c:92
```

(continued on next page)

**Example 14–2 (Cont.) Debugging a Multithreaded Kernel Core File**

```
(ladebug) thread 19
Thread 0x1
(ladebug) printregs10
$r0  [$v0]  = 0                         $r1  [$t0]  = 0
$r2  [$t1]  = 4396974517024             $r3  [$t2]  = 320024
$r4  [$t3]  = 4396974517024             $r5  [$t4]  = 0
$r6  [$t5]  = 0                         $r7  [$t6]  = 130
$r8  [$t7]  = 0                         $r9  [$s0]  = 2
$r10 [$s1]  = 2                         $r11 [$s2]  = 0
$r12 [$s3]  = 0                         $r13 [$s4]  = 0
$r14 [$s5]  = 0                         $r15 [$s6]  = 0
$r16 [$a0]  = 28613                     $r17 [$a1]  = 11
$r18 [$a2]  = 4396974766704             $r19 [$a3]  = 0
$r20 [$a4]  = 0                         $r21 [$a5]  = 4
$r22 [$t8]  = 3                         $r23 [$t9]  = 0
$r24 [$t10] = 3                         $r25 [$t11] = 0
$r26 [$ra]  = 4395904648544             $r27 [$t12] = 4395904749552
$r28 [$at]  = -1                        $r29 [$gp]  = 4396974751200
$r30 [$sp]  = 4396974766800             $r31 [$zero]= 0
$f0         = 0                         $f1         = 0
$f2         = 0                         $f3         = 0
$f4         = 0                         $f5         = 0
$f6         = 0                         $f7         = 0
$f8         = 0                         $f9         = 0
$f10        = 0                         $f11        = 0
$f12        = 0                         $f13        = 0
$f14        = 0                         $f15        = 0
$f16        = 0                         $f17        = 0
$f18        = 0                         $f19        = 0
$f20        = 0                         $f21        = 0
$f22        = 0                         $f23        = 0
$f24        = 0                         $f25        = 0
$f26        = 0                         $f27        = 0
$f28        = 0                         $f29        = 0
$f30        = 0                         $f31        = 0
$pc         = 0x3ff805065c8
(ladebug) thread 2
Thread 0x2
(ladebug) printregs
$r0  [$v0]  = -207                      $r1  [$t0]  = 0
$r2  [$t1]  = 0                         $r3  [$t2]  = 0
$r4  [$t3]  = 0                         $r5  [$t4]  = 0
$r6  [$t5]  = 1                         $r7  [$t6]  = 1
$r8  [$t7]  = 0                         $r9  [$s0]  = 4831834496
$r10 [$s1]  = 0                         $r11 [$s2]  = 4396974730048
$r12 [$s3]  = 0                         $r13 [$s4]  = 0
```

(continued on next page)

**Example 14–2 (Cont.) Debugging a Multithreaded Kernel Core File**

```
$r14 [$s5]  = 162129586585337856       $r15 [$s6]  = 0
$r16 [$a0]  = 4831834496               $r17 [$a1]  = 0
$r18 [$a2]  = 40                       $r19 [$a3]  = 6
$r20 [$a4]  = 0                        $r21 [$a5]  = 6
$r22 [$t8]  = 4                        $r23 [$t9]  = 2
$r24 [$t10] = 1                        $r25 [$t11] = 0
$r26 [$ra]  = 4395904354148            $r27 [$t12] = 4395904050272
$r28 [$at]  = 293938                   $r29 [$gp]  = 4396974651024
$r30 [$sp]  = 4831834336               $r31 [$zero]= 0
$f0         = 0                        $f1         = 0
$f2         = 0                        $f3         = 0
$f4         = 0                        $f5         = 0
$f6         = 0                        $f7         = 0
$f8         = 0                        $f9         = 0
$f10        = 3.237908616585193e-319   $f11        = 65536
$f12        = 0                        $f13        = 0
$f14        = 1.016984725399622e-319   $f15        = 20584
$f16        = 0                        $f17        = 0
$f18        = 0                        $f19        = 0
$f20        = 0                        $f21        = 0
$f22        = 3.237908616585193e-319   $f23        = 65536
$f24        = 0                        $f25        = 0
$f26        = 0                        $f27        = 0
$f28        = 0                        $f29        = 0
$f30        = 0                        $f31        = 0
$pc         = 0x3ff8051df04
(ladebug)
```

**1** When debugging a multithreaded core file, Ladebug shows the kernel thread ID of the of the thread that incurred the fault.

**Note**

For a single thread core file, Ladebug will not show the kernel thread ID.

**2** The `$threadlevel` is set to `native` and you can begin core file kernel thread debugging.

**3** Ladebug provides a list of threads.

**4** Ladebug displays the current thread context.

**5** Ladebug provides a stack trace of the current thread.

**6** Ladebug sets the thread context to thread #2.

7 Ladebug provides the stack trace of the new current thread.

8 When specifying the `where thread all`, Ladebug shows the stack trace for all threads. Notice that it labels the stack trace for each thread.

9 When specifying the `thread` command with the thread ID, Ladebug changes the thread context to that thread (in this case, thread 1).

10 The `printregs` command asks Ladebug to show the register contents within the current thread context.

# 15 Using Debugger Scripts

This chapter describes the scripting features of the Digital Ladebug debugger. Using debugger script commands, you can specify a file to input as debugger commands, send debugger output to a file, or send a transcript of an entire debugging session to a file.

## 15.1 The Debugger Initialization File

When you start the debugger, it first searches for a file named `.dbxinit` in the current directory; if it is not there, it then searches for it in your home directory. If a `.dbxinit` file exists in either directory, the debugger interprets the contents of the file as a string of debugger commands. You can customize your debugger environment by adding commands to your `.dbxinit` file that set debugger aliases (using the `alias` command), debugger variables (using the `set` command), and source-code search paths (using the `use` command).

Ladebug processes the `.dbxinit` file using the language of the main program being debugged. Depending on the language of that program, there are some situations where the behavior of some `.dbxinit` commands may vary. If this is an issue, you can put a `set $lang` command in the file to ensure the expected behavior.

Example 15–1 shows a sample `.dbxinit` file.

**Example 15–1 A Sample .dbxinit File**

```
set $listwindow = 45
alias ls "sh ls -l"
stop in main; run
```

**Example 15–2 Debugger Startup Using a .dbxinit File**

```
% ladebug sample
Welcome to the Ladebug Debugger Version 4.0
------------------
object file name: sample
Reading symbolic information ...done
[#1: stop in main ]
[1] stopped at [main:4 0x1200001180]
      4     for (i=1 ; i<3 ; i++) {
(ladebug)
```

When you invoke the debugger, it executes the commands in the `.dbxinit` file without echoing the commands to the screen. Example 15–2 shows this process for the sample `.dbxinit` file.

## 15.2 Recording Debugger Sessions

You can record both your debugger commands as well as the debugger's responses into a single file by using the `record io` command. The syntax for this command is as follows:

**record io** filename

The debugger records all input and output starting with the command following the `record io` command.

**Note**

The `record io` command does not record the output from the program you are debugging.

To stop recording, you must exit the debugger. If a file already exists with the file name you specify, the debugger appends to the file.

You can record debugger responses to your commands using the `record output` command. The syntax for this command is as follows:

**record output** filename

Use the `record input` command to create a file containing the commands you enter at the debugger prompt. The syntax for this command is as follows:

**record input** filename

Use the `record input` command to create a debugger script. After creating a file containing a list of debugger commands, you can request that the debugger execute the commands in the file using the `source` command. You can also use an editor (such as `vi` or `emacs`) to edit the commands listed in the file. Example 15–3 shows how to use the `record input` command to record a series of debugger commands in a file named `myscript`.

**Example 15–3 Recording a Debugger Script**

```
(ladebug) record input myscript
(ladebug) stop in main
[#1: stop in main ]
(ladebug) run
[1] stopped at [main:4 0x120000b14]
      4     for (i=1 ; i<3 ; i++) {
(ladebug) next
stopped at [main:5 0x120000b1c]
      5         f = factorial(i);
(ladebug) step
stopped at [factorial:13 0x120000bb8]
     13     if (i<=1)
(ladebug) where
>0  0x120000bb8 in factorial(i=1) sample.c:13
#1  0x120000b28 in main() sample.c:5
(ladebug) cont
1! = 1
2! = 2
Thread has finished executing
(ladebug) quit
% cat myscript
stop in main
run
next
step
where
cont
quit
%
```

## 15.3 Playing Back a Command Script

With the `source` or `playback input` commands, you can read in and execute a file containing debugger commands. You can create such a file manually using an editor or automatically using the `record input` command. The syntax of the command to read in a file of debugger commands is as follows:

**source** filename

**playback input** filename

The file corresponding to the *filename* argument must be in the current directory or be preceded by a path specification. When the debugger executes a script file, the `$pimode` debugger variable determines whether the commands are echoed to the display as they are executed, as follows:

- If `$pimode` is set to 0 (the default), commands read in from a script file are not echoed to the display.
- If `$pimode` is set to 1, the commands are echoed.

Example 15–4 shows how to execute a debugger script. Because `$pimode` is set to 0, only the command output, and not the commands themselves, echoes to the display.

**Example 15–4 Executing a Debugger Script**

```
(ladebug) source myscript
[#1: stop in main ]
[1] stopped at [main:4 0x120000b14]
      4     for (i=1 ; i<3 ; i++) {
stopped at [main:5 0x4001d4]
      5         f = factorial(i);
stopped at [factorial:13 0x120000b1c]
     13     if (i<=1)
>0  0x120000bb8 in factorial(i=1) sample.c:13
#1  0x120000b28 in main() sample.c:5
1! = 1
2! = 2
Thread has finished executing
%
```

# 16 Debugging Shared Libraries

You can debug shared libraries if the library was compiled and linked with the option that makes symbol table information available to the debugger. For more information about shared libraries and linking programs to shared libraries, and compiling and linking programs and libraries for debugging, see your compiler documentation.

Loadable drivers can be considered a form of shared libraries, and can be debugged. For information, see Chapter 22 on kernel debugging.

You can debug shared libraries using the same techniques you use to debug any program function. The scope and visibility rules for debugging programs apply to debugging shared libraries.

You can call functions in shared libraries that do not have debugging information available.

## 16.1 Controlling the Reading of Symbols for Shared Libraries

In general, build your images with symbol tables. You cannot set symbolic breakpoints to stop execution in or see the source code of shared libraries that do not have symbol table information available for the debugger.

By default, symbol table information is read into the debugger for all the shared libraries that are loaded, whether loaded at startup of the application or loaded dynamically by the application at run time.

For less bulky images, you can strip out symbol table information with the UNIX command `ostrip` (for information, see the `ostrip(1)` reference page). You can also use debugger commands to control symbols, as follows:

**-nosharedobjs**

The `-nosharedobjs` flag, used on the `ladebug` command, directs the debugger not to read symbol table information for any of the shared objects. (Later, you can use the the `readsharedobj` command at the `(ladebug)` prompt to read in the symbol table information for a specified shared object.)

**listobj**

The `listobj` command lists all the objects (the main image and all shared libraries) that are currently used by the debuggee process. For each object, it lists the full object name (with pathname), the starting address for the text, the size of the text region, and whether the symbol table information for this object is read by the debugger.

The pathnames listed are the ones actually used by the run-time loader when loading the shared libraries.

**readsharedobj** objectname

The `readsharedobj` command directs the debugger to read in the symbol table information for `objectname`, which must be a shared library. The command can only be used when a debuggee program is specified (that is, either Ladebug was invoked with it or the debuggee was loaded by the `load` command).

The symbol table information of `objectname` will be read into the debugger, provided that `objectname` is specified as either of the following:

- An absolute pathname that matches exactly one of the object file names listed in the `listobj` command
- A simple name that matches exactly one and only one of the simple names listed in the `listobj` command

If no match is found, no symbol table information is read into the debugger. If no unique match (for a simple name) is found, no symbol table information is read, and an error message is issued. If the symbols for the specified object have already been read, no symbol table information is read.

The `readsharedobj` command currently only reads in the symbol table information of a shared object that has already been dynamically loaded when the program executes.

**delsharedobj** objectname

The `delsharedobj` command directs the debugger to remove the symbols for `objectname`, which must be a shared object.

If `objectname` is a complete file specification, for example, /usr/shlib/libc.so, then the specified name is used as provided.

The symbol table information of `objectname` will be removed from the debugger, provided that `objectname` is specified as either of the following:

- An absolute pathname that matches exactly one of the object file names listed in the `listobj` command
- A simple name that matches exactly one and only one of the simple names listed in the `listobj` command

If no match is found, no symbol table information is removed from the debugger. If no unique match (for a simple name) is found, no symbol table information is removed, and an error message is issued.

If the last modification time and/or size of the binary file or any of the shared objects used by the binary file has changed since the last `run` or `rerun` command was issued, Ladebug automatically rereads the symbol table information when you execute the program again.

## 16.2 Listing the Shared Library Source Code

Provided your library was compiled with `-g`, to list the source code for the shared library, set the file context to the file containing the source and enter the `list` command. See Example 16–1.

**Example 16–1 Listing the Shared Library Source Code**

```
(ladebug) file func.c
(ladebug) list 10
     10
     11 int funcD(i) {
     12
     13     int a;
>    14     a = i;
     15     return a;
     16 }
```

## 16.3 Setting Breakpoints in a Shared Library

Provided your library was compiled with `-g`, set breakpoints or tracepoints in a shared library by setting a breakpoint on a specific line of source code or function contained in the library, as shown in Example 16–2.

**Example 16–2 Setting Breakpoints in a Shared Library**

```
(ladebug) stop in main;r
[#1: stop in main ]
[1] stopped at [main:28 0x120001280]
     28   a = 1;
(ladebug) stop in funcC
[#2: stop in funcC ]
(ladebug) file func.c
(ladebug) stop at 14
[#3: stop at "func.c":14 ]
(ladebug) status
#1 PC==0x120001280 in main "file.c":28 { break }
#2 PC==0x3ffbf800b98 in funcC "func.c":6 { break }
#3 PC==0x3ffbf800bd8 in funcD "func.c":14 { break }
(ladebug) c
having fun yet?!
[2] stopped at [funcC:6 0x3ffbf800b98]
      6     a = m = n = 100;
(ladebug) c
[3] stopped at [funcD:14 0x3ffbf800bd8]
     14     a = i;
(ladebug)
```

## 16.4 Printing and Modifying Shared Library Variable Values

Provided your library was compiled with `-g`, you can access variables in shared libraries if the variable is visible and in scope according to the rules of the program language. Print and modify the variable the same way you print or modify any program variable, as shown in Example 16–3.

**Example 16–3 Printing and Modifying Shared Library Variable Values**

```
(ladebug) print a
1
(ladebug) which a
"func.c"‘funcD.a
(ladebug) assign a = 2
```

(continued on next page)

**Example 16–3 (Cont.) Printing and Modifying Shared Library Variable Values**

```
(ladebug) print a
2
(ladebug)
```

## 16.5 Stepping into Shared Library Functions

When you are executing the program using the `step` command, the debugger will step into a function in the shared library the same way the debugger steps into any other program function, as shown in Example 16–4.

**Example 16–4 Stepping into Shared Library Functions**

```
(ladebug) list
     29   b = 2;
     30   c = 3;
     31
     32   printf ("having fun yet?!\en");
     33   printf ("%d  %d %d %d\en", funcA(a), funcB(a), funcC(a), funcD(a));
     34   printf ("NOT\en");
     35 }
     36
     37

(ladebug) step
stopped at [main:32 0x120001298]
     32   printf ("having fun yet?!\en");
(ladebug) step
having fun yet?!
stopped at [main:33 0x1200012b0]
     33   printf ("%d  %d %d %d\en", funcA(a), funcB(a), funcC(a), funcD(a));
(ladebug) step
stopped at [funcA:11 0x120001200]
     11     x = 1;
(ladebug) step
stopped at [funcA:12 0x120001208]
     12     a = i + g1 + x;
(ladebug)
```

## 16.6 Calling a Shared Library

Call a shared library function explicitly using the `call` command. You can also embed a call to a function contained in a shared library as shown in Example 16–5.

**Example 16–5 Calling a Shared Library**

```
(ladebug) call funcB(2)
(ladebug) print funcB(2)
2
(ladebug)
```

You can also nest calls to functions in shared libraries as shown in Example 16–6.

**Example 16–6 Nesting Calls to Shared Libraries**

```
(ladebug) rerun
[1] stopped at [main:28 0x120001280]
     28   a = 1;
(ladebug) call funcD(3)
[3] stopped at [funcD:14 0x3ffbf800bd8]
     14     a = i;
(ladebug) where
>0  0x3ffbf800bd8 in funcD(i=3) func.c:14
#1  0x1200011d0 in _mcount()
(ladebug) print funcB(3)
3
(ladebug)
```

Although it is very time-consuming and tedious to debug shared libraries that do not have symbolic information available for the debugger, you can call functions in the shared libraries. Example 16–7 shows, for example, that if your program links the `stdio` shared library, you can call the `strlen` function.

**Example 16–7 Calling a System Library Function**

```
(ladebug) list 1:5
      1
      2 #include <stdio.h>
      3
      4 int gfi = 100;
      5
(ladebug) print strlen("abc")
3
(ladebug)
```

## 16.7 Accessing Shared Libraries on the Stack Trace

If a shared library is active on the stack trace when you enter the `where` command, the shared library will be displayed. You can use the `up`, `down`, and `func` commands to change the func context, as shown in Example 16–8.

**Example 16–8 Accessing Shared Libraries on the Stack Trace**

```
(ladebug) where
>0  0x3ffbf800b98 in funcC(i=1) func.c:6
#1  0x1200012ec in main() file.c:33
(ladebug) up
>0  0x1200012ec in main() file.c:33
     33   printf ("%d  %d %d %d\en", funcA(a), funcB(a), funcC(a), funcD(a));
(ladebug) where
#0  0x3ffbf800b98 in funcC(i=1) func.c:6
>1  0x1200012ec in main() file.c:33
(ladebug) func funcC
funcC in func.c line No. 6:
      6    a = m = n = 100;
(ladebug) where
>0  0x3ffbf800b98 in funcC(i=1) func.c:6
#1  0x1200012ec in main() file.c:33
(ladebug)
```

## 16.8 Disassembling a Memory Address in a Shared Library

Disassemble and modify shared library values contained in memory the same way you disassemble other program function values. In Example 16–9, a range of addresses is disassembled.

**Example 16–9 Disassembling a Memory Address in a Shared Library**

```
(ladebug) 0x3ffbf800b98 /2i
*[funcC:6, 0x3ffbf800b98]      addq    zero, 0x64, t0

 [funcC:6, 0x3ffbf800b9c]      stl     t0, 8(sp)

(ladebug)
```

# 17 Working with Limited Debugging Information

Depending on the options you use when compiling and linking your program, the debugging information available in your program's executable file may range from full to nonexistent. Programs that include shared libraries or other code modules may contain limited debugging information regardless of the compile options you use. Ladebug supports the debugging of programs that do not contain complete debugging information.

This chapter describes how to use Ladebug to debug a program containing limited debugging information. It provides examples and discusses some of the limitations you may experience. There are many scenarios under which a program can be compiled and linked—discussing each is beyond the scope of this chapter.

## 17.1 How Ladebug Works with Limited Debugging Information

Some compilers provide variants of the debug flag that provide different levels of debugging information and optimization. Depending on the options you use when compiling and linking your program, the debugging information available in the program's executable file may range from full to nonexistent. Programs that include shared libraries or other code modules may contain limited debugging information regardless of the compile options you use. Ladebug uses whatever information is available during a debugging session.

For example, with full debugging information, Ladebug can set breakpoints on procedures and functions; it recognizes routine names and knows parameters and values; it can display source code, knows the source file name, and can provide line numbers.

When encountering limited debugging information, Ladebug attempts to set breakpoints by making assumptions from the available information. See Section 17.2 for sample sessions in which you debug programs with limited information.

If no debugging information is available in the program's executable file, Ladebug allows for machine-level debugging. (See Chapter 18 for information on machine-level debugging.)

The following are some examples of compile and link options that are likely to produce limited debugging information:

- Compiled with no `-g` flag
- Compiled with `-g0`, `-g1`, or `-g`
- Linked with the `-x` flag. (The `-r` linker flag is usually used with the `-x` flag, except in the final link.)
- Linked with the `-s` flag
- Linked with a combination of flags on compiles or previous links
- Partially or fully stripped by the `ostrip` command
- Stripped using the `strip` command

**Note**

For the most up-to-date information on compiler and linker options and defaults, see the manual reference page for your particular language compiler and operating system release.

## 17.2 Example Debugging Sessions

The following sections compare Ladebug's ability to debug programs containing full or limited debugging information in the program's symbol table. Each example is provided in two forms:

1. Compiled and linked with `-g2` (full debugging information)
2. Compiled and linked in such a way to produce limited debugging information

As a sample program is analyzed, you will first see the output based on full debugging information, which is then compared to the output where some of the debugging information is missing.

The examples provided are for illustration only and do *not* cover all of the conditions you may encounter. Use these example to help understand what may be happening when you debug your own programs and modules that contain less than full debugging information.

**Note**

If you encounter difficulty debugging your program, the best solution is to recompile and relink with the `-g2` flag.

---

Sample sessions are presented as follows:

- Section 17.2.1 presents a C++ program linked with `-x`. The operating system is Digital UNIX 4.0.
- Section 17.2.2 presents a C program linked with `-x`. The operating system is Digital UNIX 3.2.
- Section 17.2.3 presents a C++ program containing two files, one of which has been linked with `-x -r`. The operating system is Digital UNIX 3.2.
- Section 17.2.4 presents the same program as in Section 17.2.3, except that it was linked with a different series of `-x` and `-r` flags over both files.

### 17.2.1 Example C++ Program Linked with -x

This section presents a sample debugging session on a C++ program containing full symbolic debugging information, then the same program compiled with `-g2` and linked with `-x`. In the second form of the example program, `-x` strips all symbolic debugging information except for procedure and file information. The operating system is Digital UNIX 4.0.

#### 17.2.1.1 Setting Breakpoints

If your executable file contains full symbolic debugging information, Ladebug can set a breakpoint at various levels, as shown in Example 17–1.

**Example 17–1 Setting Breakpoints in a C++ Program Compiled and Linked with -g2**

```
(ladebug) stop in Thing::Thing
Select an overloaded function1
----------------------------------------------------
     1 Thing::Thing(char* const)
     2 Thing::Thing(const int)
     3 Thing::Thing(void)
     4 None of the above
----------------------------------------------------
3
[#1: stop in Thing::Thing(void) ]
(ladebug) stop in Thing::Thing(const int)
[#2: stop in Thing::Thing(const int) ]

(ladebug) stop in dump
Symbol dump not visible in current scope.
dump has no valid breakpoint address
Warning: Breakpoint not set
(ladebug) stop in Thing::dump
[#3: stop in void Thing::dump(const char* const) ]2
```

1 Ladebug can display all the constructors. Note also the need for the `Thing::` scope qualification; `Thing::` is part of the name of the routine.

2 `Thing::` is part of the name for the function `dump` .

If your executable file contains limited symbolic debugging information, Ladebug lacks information to set breakpoints as shown in Example 17–2.

**Example 17–2 Setting Breakpoints in a C++ Program Compiled with -g2 and Linked with -x**

```
(ladebug) stop in Thing::Thing
Thing is not a valid breakpoint address
Warning: Breakpoint not set1
(ladebug) stop in Thing2
Thing is not a valid breakpoint address
Warning: Breakpoint not set
(ladebug) stop in Thing::~Thing
stop in Thing::~Thing
Unable to parse input as legal command or C++ expression.
(ladebug) stop in ~Thing
stop in ~Thing
Unable to parse input as legal command or C++ expression.
(ladebug) stop in dump
[#1: stop in dump ]3
```

1 Even after resetting the language, Ladebug still cannot set the breakpoint.

2 Repeated attempts to set breakpoints in constructors fail.

3 Ladebug can still set a breakpoint on this function because `dump` is unique and not dependent upon the classname `Thing::` to distinguish it.

#### 17.2.1.2 Listing the Source Code

Example 17–3 shows how Ladebug can list the source code corresponding to the position of the program counter if your executable file contains full debugging information.

**Example 17–3 Listing the Source Code of a C++ Program Compiled and Linked with -g2**

```
(ladebug) run
[1] stopped at [Thing::Thing(void):58 0x120002170]
(Cannot find source file classdefinition.C)
(ladebug) use ./src
Directory search path for source files:
 . ./bin /usr/users/debug/ladebug ./src
(ladebug) list
     59     dump ("Thing constructor, no arguments");
     60 }
     61
     62
     63 Thing::Thing (const Thing1 t1)
     64     : thisThing1 (t1),
     65       thisThing2 (bogusThing2),
     66       thisSideEffect (1)
     67 {
     68     sideEffect++;
     69     dump ("Thing constructor, Thing1 argument");
     70 }
     71
     72 #ifndef CHANGE_ORDER
     73 Thing::Thing (const Thing2 t2)
     74     : thisThing1 (bogusThing1),
     75       thisThing2 (t2),
     76       thisSideEffect (1)
     77 {
     78     sideEffect++;
     79     dump ("Thing constructor, Thing2 argument");
```

Without full symbolic debugging information, Ladebug does not know name of the source file. It can display some information for the routine `dump` because it is unique and not dependent upon the `Thing::` class.

#### 17.2.1.3 Displaying the Stack Trace

If your executable file contains full debugging information, Ladebug can display the stack trace of currently active functions. In Example 17–4, note the detailed call stack with class types and values in them.

**Example 17–4 Displaying the Stack Trace of a C++ Program Compiled and Linked with -g2**

```
(ladebug) where
>0  0x120002170 in ((Thing*)0x11ffff7c8)->Thing() classdefinition.C:58
#1  0x120002574 in main() classdefinition.C:106

(ladebug) cont
[3] stopped at [void Thing::dump(const char* const):93 0x1200023f0]
     93     sideEffect++;

(ladebug) where
>0  0x1200023f0 in ((Thing*)0x11ffff7c8)->dump(header=0x120001b80="Thing
constructor, no arguments") classdefinition.C:93
#1  0x120002198 in ((Thing*)0x11ffff7c8)->Thing() classdefinition.C:59
#2  0x120002574 in main() classdefinition.C:106
(ladebug) quit
```

If your executable file contains limited symbolic debugging information, Ladebug can show that you are in a routine called `Thing` but can't differentiate which one. Example 17–5 illustrates this.

**Example 17–5 Displaying the Stack Trace of a C++ Program Compiled with -g2 and Linked with -x**

```
(ladebug) where
>0  0x1200023e8 in dump(0x3ff80894608, 0x12000294f, 0x11ffff7c8, 0xc70,
 0x3ff808941cc, 0x120002950) DebugInformationStrippedFromFile0:???
#1  0x120002198 in Thing(0x11ffff7c8, 0xc70, 0x3ff808941cc, 0x120002950,
 0xd10, 0x3ffc0819100) DebugInformationStrippedFromFile0:???
#2  0x120002574 in main(0x3ffc0002078, 0xffffffff, 0x120001b70, 0x1,
 0x140000060, 0x0) DebugInformationStrippedFromFile0:???
(ladebug) quit
```

### 17.2.2 Example C Program Linked with -x

This section presents a sample debugging session on a C program, first containing full symbolic debugging information, then compiled in the same way but linked with the `-x` flag. The operating system is Digital UNIX 3.2.

In the second form of the program, `-x` strips all symbolic debugging information except for procedure and file information.

#### 17.2.2.1 Setting Breakpoints on Routines

Example 17–6 invokes the user program with full debugging information and sets breakpoints on three routines: `main`, `buildLocalList`, and `createNewElement`.

**Example 17–6 Setting Breakpoints on Routines in a C Program Compiled and Linked with -g2**

```
csh> $LADEBUG ../bin/c_gflags001-g
Welcome to the Ladebug Debugger Version 4.0-9
------------------
object file name: ../bin/c_gflags001-g
Reading symbolic information ...done
(ladebug) stop in main
[#1: stop in main ]
(ladebug) stop in buildLocalList
[#2: stop in buildLocalList ]
(ladebug) stop in createNewElement
[#3: stop in createNewElement ]
(ladebug) run
[1] stopped at [void main(void):157 0x1200014b8]1
    157  mainList = buildLocalList (5);
(ladebug)
```

1 With full debugging information, when you stop at a breakpoint, Ladebug can determine the routine name, the line number, and the address for the routine.

Example 17–7 shows a program compiled with `-g2` and linked with `-x`. In this case, the user program has all symbolic debugging information stripped, except for procedure and file information. The local (nonglobal) symbols are not preserved in the debugging information and Ladebug has no information to work with.

**Example 17–7 Setting Breakpoints on Routines in a C Program Compiled with -g2 and Linked with -x**

```
csh> $LADEBUG ../bin/c_gflags001-x
Welcome to the Ladebug Debugger Version 4.0-9
------------------
object file name: ../bin/c_gflags001-x
Reading symbolic information ...done
(ladebug) stop in main
[#1: stop in main ]
(ladebug) stop in buildLocalList
[#2: stop in buildLocalList ]
(ladebug) stop in createNewElement
[#3: stop in createNewElement ]
(ladebug) run
[1] stopped at [main:??? 0x1200014b0]1
```

1 In this example, there is no line number information and the start address is slightly different (0x1200014b0).

Ladebug normally starts on the first line of the source code. It skips over initialization and other bookkeeping information in the prologue when you enter a routine, so that the stack and parameters are in the expected state. In this example, the information about the prologue is not available, so Ladebug does its best to stop in the routine.

### 17.2.2.2 Listing the Source Code

Example 17–8 shows how Ladebug can list the source code corresponding to the position of the program counter if your executable contains full debugging information,

Ladebug knows about the current program counter and displays lines from line 157.

**Example 17–8 Listing the Source Code of a C Program Compiled and Linked with -g2**

```
(ladebug) use ../src
Directory search path for source files:
 . ../bin ../src

 (ladebug) list
    158  dumpLocalList ("Local list");
    159
    160  buildDuplicateList ();
    161  dumpDuplicateList ("Duplicate list");
    162
    163  return;
    164 }  /* main */
    165
```

In Example 17–9, the name of the source file has been stripped out completely from the symbol table. Ladebug makes up a name for this file: DebugInformationStrippedFromFile0.

**Example 17–9 Listing the Source Code of a C Program Compiled with -g2 and Linked with -x**

```
(ladebug) use ../src
Directory search path for source files:
 . ../bin ../src

 (ladebug) list
 (Can't find file DebugInformationStrippedFromFile0)
```

The name Ladebug creates is of some value. In some cases, this may be the only way of discriminating between two different instances of an overloaded function name. For example, there may be two different functions named `buildList` that can only be resolved by using one of these generated source file names.

The file name "DebugInformationStrippedFromFile0" has the symbol table file number in it. For more complex programs, you could use `odump(1)` and `stdump(1)` to identify where you are in your code.

#### 17.2.2.3 Displaying the Stack Trace

If your executable file contains full debugging information, Ladebug can display the stack trace of currently active functions with detailed information as shown in Example 17–10.

**Example 17–10 Displaying the Stack Trace of a C Program Compiled and Linked with -g2**

```
(ladebug) cont
[2] stopped at [buildLocalList:65 0x1200013ac]
     65  firstElement = NULL_LIST;1
(ladebug) where2
>0  0x1200013ac in buildLocalList(lengthOfList=5) gflags001a.c:65
#1  0x1200014c4 in main() gflags001a.c:157
(ladebug) c
[3] stopped at [createNewElement:44 0x120001348]
     44  newElement = (ListElementHandle) malloc (sizeof (ListElement));
(ladebug) where3
>0  0x120001348 in createNewElement(dataValue=0, useValue=0)gflags001a.c:44
#1  0x1200013d8 in buildLocalList(lengthOfList=5) gflags001a.c:71
#2  0x1200014c4 in main() gflags001a.c:157
(ladebug) quit
```

1 Because Ladebug knows the name of the file, it can list the source line.

2 Note the parameter details. Ladebug knows the name of the parameter (`lengthOfList`), and its type and value. It also knows that there is only one parameter for `buildLocalList` and none for `main`.

3 More call stack information is available. Ladebug knows the names and types of parameters and their values.

With limited debugging information, Ladebug must approximate the call stack information, as in Example 17–11.

**Example 17–11 Displaying the Stack Trace of a C Program Compiled with -g2 and Linked with -x**

```
(ladebug) c
[2] stopped at [buildLocalList: ??? 0x120001398]1
(ladebug) where
>0  0x120001398 in buildLocalList(0x1200012b4, 0x0, 0x0, 0x0, 0x1, 0x11ffffbf8)
DebugInformationStrippedFromFile0:???2
#1  0x1200014c4 in main(0x1, 0x20000000, 0x120001278, 0x120001200, 0x1200012b4,
DebugInformationStrippedFromFile0:???
(ladebug) c
[3] stopped at [createNewElement: ??? 0x120001328]
(ladebug) where3
>0  0x120001328 in createNewElement(0x3ff80016b18, 0x0, 0x1200014c4, 0x0, 0x1,
DebugInformationStrippedFromFile0:???
#1  0x1200013d8 in buildLocalList(0x1200014c4, 0x0, 0x1, 0x0, 0x100000000,0x10
DebugInformationStrippedFromFile0:???
#2  0x1200014c4 in main(0x1, 0x0, 0x100000000, 0x100000005, 0x1200012b4, 0x0)
DebugInformationStrippedFromFile0:???
(ladebug) quit
```

**1** Ladebug does not have line number information. There is also a difference in the breakpoint location.

**2** Note the significant differences in the call stack parameter information. Because the type and parameter name information has been stripped out, Ladebug provides the first six register parameters per the compiler calling convention.

**3** Ladebug again approximates the call stack information.

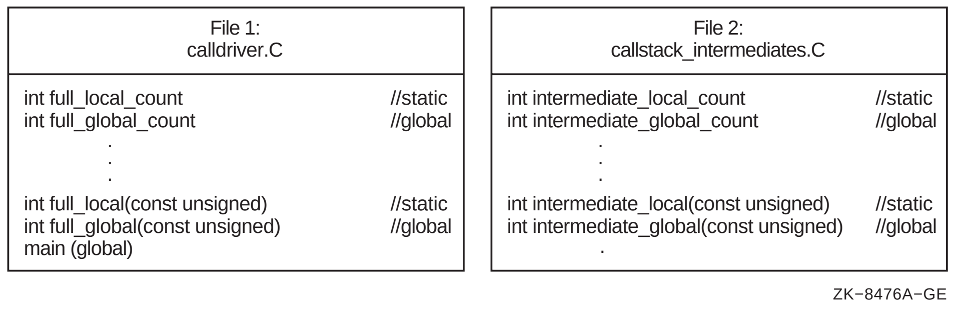
### 17.2.3 Example C++ Program Linked with -x -r

This example program contains two program files, calldriver.C and callstack_intermediates.C, that contain various static and global routines. The static routines are known only within the file while the global (external) routines are known to both files.

In the first form of this program, both calldriver.C and callstack_intermediates.C have full symbolic debugging information. The second form of the program contains limited debugging information; calldriver.C is compiled with `-g2` and linked with the `-x -r` options, stripping out much of the symbol table, so only part of this program has reasonable symbols.

Figure 17–1 shows the structure of the example program.

**Figure 17–1 Example C++ Program Linked with -x -r**


File 1:
calldriver.C
int full_local_count //static
int full_global_count //global
int full_local(const unsigned) //static
int full_global(const unsigned) //global
main (global)
File 2:
callstack_intermediates.C
int intermediate_local_count //static
int intermediate_global_count //global
int intermediate_local(const unsigned) //static
int intermediate_global(const unsigned) //global
ZK−8476A−GE

### 17.2.3.1 Setting Breakpoints on Static and Global Routines

Example 17–12 invokes the user program with full debugging information and sets breakpoints on four routines.

**Example 17–12 Setting Breakpoints on Static and Global Routines in a C++ Program Compiled and Linked with -g2**

```
csh> $LADEBUG ../bin/x_callstack01-g
Welcome to the Ladebug Debugger Version 4.0-9
------------------
object file name: ../bin/x_callstack01-g
Reading symbolic information ...done
Directory search path for source files:
 . ../bin /usr/users/debug/ladebug
(ladebug) stop in full_local
[#1: stop in int full_local(const unsigned) ]1
(ladebug) stop in full_global
[#2: stop in int full_global(const unsigned) ]
(ladebug) stop in intermediate_global
[#3: stop in int intermediate_global(const unsigned) ]
(ladebug) stop in intermediate_local
[#4: stop in int intermediate_local(const unsigned) ]
(ladebug) run2
[3] stopped at [int intermediate_global(const unsigned):41 0x12000206c]
41 intermediate_global_count++;3
```

**1** Ladebug can set breakpoints on all four routines. In the first breakpoint, Ladebug knows the name of the routine and the number and types of the arguments. Be aware that the routines named "..._local" are static and visible only in their own file.

2 When you enter the `run` command, ladebug can determine the routine name, the line number, and address for the routine.

3 Ladebug also knows the file name.

When calldriver.C is compiled with `-g2` but linked with `-x -r`, you get the result shown in Example 17–13 when you run the debugger:

**Example 17–13 Setting Breakpoints on Static and Global Routines in a C++ Program Compiled with -g2 and Linked with -x -r**

```
csh> $LADEBUG ../bin/x_callstack01-xr
Welcome to the Ladebug Debugger Version 4.0-9
------------------
object file name: ../bin/x_callstack01-xr
Reading symbolic information ...done
Directory search path for source files:
 . ../bin /usr/users/debug/ladebug
(ladebug) stop in full_local
Symbol full_local undefined.1
full_local has no valid breakpoint address
Warning: Breakpoint not set
(ladebug) stop in full_global
[#1: stop in full_global ]2
(ladebug) stop in intermediate_local
[#2: stop in int intermediate_local(const unsigned) ]3
(ladebug) stop in intermediate_global
[#3: stop in int intermediate_global(const unsigned) ]
(ladebug) run4
[3] stopped at [int intermediate_global(const unsigned):41 0x12000206c]
(Cannot find source file callstack_intermediates.C)
```

1 The static routine in calldriver.C is no longer visible and Ladebug can't set a breakpoint.

2 There are no parameters, no types, and no return type of the routine. The information is missing from the symbollic debugging information.

3 The information about these routines is unchanged, since full debugging information is available for callstack_intermediates.C.

4 The name of the source file has been stripped out, so Ladebug creates a new name (DebugInformationStrippedFromFile2). Because DebugInformationStrippedFromFile2 has the symbol table file number (2) in it, you can run `odump( )-Fv ../bin/x_callstack01-xr` and look for more information about this file.

### 17.2.3.2 Listing the Source Code

Example 17–14 shows how Ladebug can list the source code for the file callstack_intermediates.C if your executable file contains full debugging information. For this part of the program, Ladebug knows source lines, routine types, and routine parameters.

**Example 17–14 Listing the Source Code of a C++ Program**

```
(ladebug) use ../src
Directory search path for source files:
 . ../bin /usr/users/debug/ladebug ../src
(ladebug) list
     42
     43 #ifdef DO_IO
     44     cout << "intermediate_global: called with (" << i
     45          << "), count now (" << intermediate_global_count << ")"<<endl;
     46 #endif // DO_IO
     47
     48     const int result = intermediate_local (i);
     49     return result;
     50 }  // intermediate_global
     51
```

When the program is compiled the same way but linked with `-x -r`, the results are the same for callstack_intermediates.C. However, you can't see or list the source for callstack_driver.C.

### 17.2.3.3 Displaying the Stack Trace

Example 17–15 shows how Ladebug displays the stack trace of currently active functions if your executable file contains full debugging information.

**Example 17–15 Displaying the Stack Trace of a C++ Program Compiled and Linked with -g2**

```
(ladebug) where1
>0  0x12000206c in intermediate_global(i=0) callstack_intermediates.C:41
#1  0x120001f6c in main() callstack_driver.C:69
(ladebug) cont2
[4] stopped at [int intermediate_local(const unsigned):27 0x120001fec]
     27     intermediate_local_count++;
(ladebug) cont
[2] stopped at [int full_global(const unsigned):41 0x120001edc]
     41     full_global_count++;
(ladebug) cont
[1] stopped at [int full_local(const unsigned):27 0x120001e70]
     27     full_local_count++;

(ladebug) where3
>0  0x120001e70 in full_local(i=0) callstack_driver.C:27
#1  0x120001efc in full_global(i=0) callstack_driver.C:48
#2  0x12000200c in intermediate_local(i=0) callstack_intermediates.C:34
#3  0x12000208c in intermediate_global(i=0) callstack_intermediates.C:48
#4  0x120001f6c in main() callstack_driver.C:69
```

1 Ladebug is able to show the parameter details: the address, the routine name, the parameter and its value, the source file name, and line number.

2 When you issue the `cont` command, Ladebug provides full information on the next breakpoint. In this case, it shows the static routine `intermediate_local` and the number of arguments and its type (`const unsigned`).

3 Information is available for static and global routines in both files.

When callstack_driver.C is compiled with -g2 and linked with `-x -r`, the results are different, as shown in Example 17–16.

**Example 17–16 Displaying the Stack Trace of a C++ Program Compiled with -g2 and Linked with -x -r**

```
(ladebug) where
>0  0x12000206c in intermediate_global(i=0) callstack_intermediates.C:41
#1  0x120001f6c in ../bin/x_callstack01-xr1
(ladebug) cont
[2] stopped at [int intermediate_local(const unsigned):27 0x120001fec]
     27     intermediate_local_count++;
(ladebug) cont
[1] stopped at [full_global: ??? 0x120001ec8]2
(ladebug) where
>0  0x120001ec8 in full_global(0x0, 0x870, 0x3ff808941cc, 0x1200022d0,
0x8e0, 0x3ffc0819100) DebugInformationStrippedFromFile2:???3
#1  0x12000200c in intermediate_local(i=0) callstack_intermediates.C:34
#2  0x12000208c in intermediate_global(i=0) callstack_intermediates.C:48
#3  0x120001f6c in ../bin/x_callstack01-xr4
```

1 Ladebug cannot associate the program counter with a routine or a file but because the program counter is within a known image, it displays the image name.

2 Ladebug sets a breakpoint on the global routine `full_global` but cannot tell in which file or on what line number.

3 Instead, Ladebug creates a file name and associates it with `full_global`: DebugInformationStrippedFromFile2. Since it does not know the names or values of the parameters associated with `full_global`, it lists the first six register parameters, according to the compiler calling convention

4 Ladebug again cannot associate the program counter with a routine or a file but can display the image name.

#### 17.2.3.4 Printing Static and Local Variables

The results are different when you print static and local variables with full or limited debugging information. Full information produces the results shown in Example 17–17.

**Example 17–17 Printing Variables of a C++ Program Compiled and Linked with -g2**

```
(ladebug) print full_local_count1
0
(ladebug) print full_global_count2
1
(ladebug) print intermediate_local_count
Symbol intermediate_local_count not visible in current scope.
Error: no value for intermediate_local_count3
(ladebug) print intermediate_global_count
1
```

1 Ladebug can print the static variable located in this file.

2 Ladebug can print the global variable.

3 This is a legitimate message; Ladebug cannot print the value of a static variable defined in the other file that is not visible.

Example 17–18 shows the results with limited debugging information for callstack_driver.C.

**Example 17–18 Printing Variables of a C++ Program Compiled with -g2 and Linked with -x -r**

```
(ladebug) print full_local_count
Symbol full_local_count undefined.1
Error: no value for full_local_count
(ladebug) print full_global_count2
0
```

1. Ladebug does not recognize the static variable in this file, even though it would be visible here when compiled under `-g2`.
2. Ladebug can print the global variable. If this were a complicated type, Ladebug would probably show it as an int. Type information is lost in most cases of limited debugging information. Local variable information is also lost in most cases.

## 17.2.4 Example C++ Program Linked with Various -x and -r Options

The sample program in this section is the same as the program used in Section 17.2.3, except that it was linked with a different series of `-x` and `-r` flags. As a result, different parts of the program have symbolic debugging information removed. Both files, calldriver.C and callstack_intermediates.C, contain limited debugging information.

As you go through this example debugging session, compare the results with output of the programs compiled with `-g2` in Section 17.2.3.

### 17.2.4.1 Setting Breakpoints on Static and Global Routines

When you try to set a breakpoint, Ladebug cannot recognize the names of static or global routines. The information about the routines in both calldriver.C and callstack_intermediates.C has been stripped out, as shown in Example 17–19.

**Example 17–19 Setting Breakpoints on Static and Global Routines in a C++ Program with Various -x and -r flags**

```
csh> $LADEBUG ../bin/x_callstack03-xr
Welcome to the Ladebug Debugger Version 4.0-9
------------------
object file name: ../bin/x_callstack03-xr
Reading symbolic information ...done
Directory search path for source files:
 . ../bin /usr/users/debug/ladebug
(ladebug) stop in full_local
Symbol full_local undefined.
full_local has no valid breakpoint address1
Warning: Breakpoint not set
(ladebug) stop in full_global
[#1: stop in full_global ]2
(ladebug) stop in intermediate_local
Symbol intermediate_local undefined.
intermediate_local has no valid breakpoint address3
Warning: Breakpoint not set
(ladebug) stop in intermediate_global
[#2: stop in intermediate_global ]4
(ladebug) run
[2] stopped at [intermediate_global: ??? 0x1200015a8]
```

**1** The static routine is not visible to Ladebug.

**2** Ladebug does not know the return type of the routine, the parameters, or the types.

**3** This static routine is also not visible.

**4** Ladebug does not know the return type of the routine, the parameters, or the types for this global routine in callstack_intermediates.C.

#### 17.2.4.2 Listing the Source Code

With limited debugging information on both files, Ladebug cannot list the source since it does not have the name of the file. It does provide a name for the file (DebugInformationStrippedFromFile7), as shown in Example 17–20, which allows you to distinguish from other information later on.

**Example 17–20 Listing the Source Code of a C++ Program with Various -x and -r Flags**

```
(ladebug) use ../src
Directory search path for source files:
 . ../bin /usr/users/debug/ladebug ../src
(ladebug) list
(Can't find file DebugInformationStrippedFromFile7)
```

### 17.2.4.3 Displaying the Stack Trace

With limited debugging information, Ladebug displays the stack trace as shown in Example 17–21.

**Example 17–21 Displaying the Stack Trace of a C++ Program with Various -x and -r Flags**

```
(ladebug) where
>0  0x1200015a8 in intermediate_global(0x1, 0x0, 0x0, 0x0, 0x1, 0x11ffff9a0)
DebugInformationStrippedFromFile7:???1
#1  0x1200014bc in ../bin/x_callstack03-xr2
(ladebug) cont
[1] stopped at [full_global: ??? 0x120001418]3
(ladebug) where
>0  0x120001418 in full_global(0x0, 0x1400000d8, 0x3ff808941cc,
0x120001810, 0x6d0, 0x3ffc0819100)DebugInformationStrippedFromFile6:???4
#1  0x12000155c in UnknownProcedure0FromFile7(0x0, 0x1400000d8, 0x3ff808941cc,
0x120001810, 0x6d0, 0x3ffc0819100) DebugInformationStrippedFromFile7:???5
.
.
.
```

1. In the first (bottom) frame of the stack, Ladebug recognizes the routine name but cannot associate it with a file. It provides the file name DebugInformationStrippedFromFile7 so you can discriminate this routine. It does not know the parameters for the routine so it lists the first six register parameters according to the compiler calling convention.
2. The filename is not known, so Ladebug displays the image name.
3. Ladebug sets a breakpoint on the global routine `full_global` but cannot tell which file or line number.
4. Ladebug creates a new file name for each file in the program. In this case, it created the name DebugInformationStrippedFromFile6.

5 Ladebug recognizes that this is a routine but does not know the name, which means it is a static routine (`-x -r` strips out the names of static routines but retains some routine information). Ladebug creates a routine name (UnknownProcedure0FromFile7). Because Ladebug recognizes the routine name "UnknownProcedure0FromFile7", you can figure out the name of the static routine Ladebug would use by running `odump -Pv`.

#### 17.2.4.4 Setting a Breakpoint on an Unknown Routine

Ladebug lets you set a breakpoint on the routine name it created in Example 17–10, even though information is lacking for static routines. It can display information on global variables. Information about static variables local to this file is lost, as shown in Example 17–22.

**Example 17–22 Setting a Breakpoint on an Unknown Routine in a C++ Program with Various -x and -r Flags**

```
(ladebug) stop in UnknownProcedure0FromFile7
[#3: stop in UnknownProcedure0FromFile7 ]
(ladebug) print full_local_count
Symbol full_local_count undefined.
Error: no value for full_local_count
(ladebug) print full_global_count
0
(ladebug) print intermediate_local_count
Symbol intermediate_local_count undefined.
Error: no value for intermediate_local_count
(ladebug) print intermediate_global_count
1
(ladebug) quit
```

# 18
# Machine-Level Debugging

The Ladebug debugger lets you debug your programs at the machine-code level as well as at the source-code level. Using debugger commands, you can examine and edit values in memory, print the values of all machine registers, and step through program execution one machine instruction at a time.

Only those users familiar with machine-language programming and executable-file-code structure will find low-level debugging useful.

## 18.1 Examining Memory Addresses

You can examine the value contained at an address in memory as follows:

- The `<examine address>` command prints the value contained at the address in one of a number of formats (decimal, octal, hexadecimal, and so on).
- The `print` command, with the appropriate pointer arithmetic, prints the value contained at the address in decimal.

### 18.1.1 Using the <examine address> Command

The `<examine address>` command has two main syntaxes. The following syntax prints a range of addresses by specifying the beginning and end of the range:

```
start_address, end_address / mode
```

If a symbol precedes the slash (/) in an address expression, you may need to enclose the expression in parentheses. For example:

```
(ladebug) ($pc), ($pc+12) / i
```

The following command syntax prints a range of addresses by specifying the beginning address and the total number of memory locations display:

```
start_address / count mode
```

You can enter memory addresses in decimal or in hexadecimal by preceding the number with `0X`. The *mode* variable determines how the values are displayed. Table 18–1 lists the valid modes.

**Table 18–1 Valid Memory Display Modes**

| Mode | Description |
|---|---|
| `d` | Print a short word in decimal |
| `u` | Print a short word in unsigned decimal |
| `D` | Print a long word in decimal |
| `U` | Print a long word in unsigned decimal |
| `o` | Print a short word in octal |
| `O` | Print a long word in octal |
| `x` | Print a short word in hexadecimal |
| `X` | Print a long word in hexadecimal |
| `b` | Print a byte in hexadecimal |
| `c` | Print a byte as a character |
| `s` | Print a string of characters (a C-style string that ends in null) |
| `f` | Print a single-precision real number |
| `g` | Print a double-precision real number |
| `i` | Disassemble machine instructions |

Example 18–1 shows how to disassemble a range of memory.

**Example 18–1 Disassembling Values Contained in a Range of Addresses**

```
(ladebug) 0x120001180, 0x120001185 / i
 [main:4, 0x120001180]  addq    zero, 0x1, t0

 [main:4, 0x120001184]  stl     t0, 24(sp)
(ladebug) 0x120001180 / 2 i
 [main:4, 0x120001180]  addq    zero, 0x1, t0

 [main:4, 0x120001184]  stl     t0, 24(sp)
(ladebug)
```

In this example, the same range of addresses was accessed using `start_address` command in both the `end_address` syntax and the `/ count` syntax.

### 18.1.2 Using Pointer Arithmetic

You can use C and C++ pointer-type conversions to display the contents of a single address in decimal. Using the `print` command, the syntax is as follows:

**print *(int *)**(address)

Using the same pointer arithmetic, you can use the `assign` command to alter the contents of a single address. Use the following syntax:

**assign *(int *)**(address) = value

Example 18–2 shows how to use pointer arithmetic to examine and change the contents of a single address.

**Example 18–2 Using Pointer Arithmetic to Display and Change Values in Memory**

```
(ladebug) print *(int*)(0x10000000)
4198916
(ladebug) assign *(int*)(0x10000000) = 4194744
(ladebug) print *(int*)(0x10000000)
4194744
(ladebug)
```

## 18.2 Examining Machine-Level Registers

The `printregs` command prints the values of all machine-level registers. The registers displayed by the debugger are machine dependent. The values are in decimal or hexadecimal, depending on the value of the `$hexints` variable (the default is 0, decimal). The register aliases are shown; for example, `$r1 [$t0]`.

Example 18–3 shows Digital UNIX Alpha machine-level registers.

**Example 18–3 Printing Machine Registers on the Digital UNIX Alpha Platform**

```
(ladebug) printregs
$r0  [$v0]  = 10                       $r1  [$t0]  = 1
$r2  [$t1]  = 4831844048               $r3  [$t2]  = 5368719424
$r4  [$t3]  = 0                        $r5  [$t4]  = 0
$r6  [$t5]  = 4396972783304            $r7  [$t6]  = 2
$r8  [$t7]  = 10                       $r9  [$s0]  = 337129856
$r10 [$s1]  = 337127744                $r11 [$s2]  = 4396973344608
$r12 [$s3]  = 0                        $r13 [$s4]  = 5368847640
$r14 [$s5]  = 5368753616               $r15 [$s6]  = 20
$r16 [$a0]  = 1                        $r17 [$a1]  = 4831835496
$r18 [$a2]  = 4831835512               $r19 [$a3]  = 4831835848
$r20 [$a4]  = 4396981193976            $r21 [$a5]  = 5
$r22 [$t8]  = 9                        $r23 [$t9]  = 9
$r24 [$t10] = 4831842472               $r25 [$t11] = 1648
$r26 [$ra]  = 4831842828               $r27 [$t12] = 4831842912
$r28 [$at]  = 4396981208928            $r29 [$gp]  = 5368742064
$r30 [$sp]  = 4831835408               $r31 [$zero]= 4831842928
$f0         = 0.1                      $f1         = 0
$f2         = 0                        $f3         = 0
$f4         = 0                        $f5         = 0
$f6         = 0                        $f7         = 0
$f8         = 0                        $f9         = 0
$f10        = 0                        $f11        = 0
$f12        = 0                        $f13        = 0
$f14        = 2.035550460865936e-320   $f15        = 4120
$f16        = 0                        $f17        = 0
$f18        = 0                        $f19        = 0
$f20        = 0                        $f21        = 0
$f22        = 0                        $f23        = 0
$f24        = 0                        $f25        = 0
$f26        = 0                        $f27        = 0
$f28        = 0                        $f29        = 0
$f30        = 0                        $f31        = 0
$pc         = 0x120001270
(ladebug)
```

## 18.3 Stepping at the Machine Level

The `stepi` and `nexti` commands let you step through program execution incrementally, like the `step` and `next` commands described in Chapter 9. The `stepi` and `nexti` commands execute one machine instruction at a time, as opposed to one line of source code. Example 18–4 shows stepping at the machine-instruction level.

**Example 18–4 Stepping Through Program Execution One Machine Instruction at a Time**

```
(ladebug) stop in main
[#1: stop in main ]
(ladebug) run
[1] stopped at [main:4 0x120001180]
      4     for (i=1 ; i<3 ; i++) {
(ladebug) stepi
stopped at [main:4 0x120001184] stl     t0, 24(sp)
(ladebug) Return
stopped at [main:5 0x120001188] ldl     a0, 24(sp)
(ladebug) Return
stopped at [main:5 0x12000118c] ldq     t12, -32664(gp)
(ladebug) Return
stopped at [main:5 0x120001190] bsr     ra,
(ladebug) Return
stopped at [factorial:12 0x120001210]   ldah    gp, 8192(t12)
(ladebug)
```

At the machine-instruction level, you can step into, rather than over, a function's prolog. While within a function prolog, you may find that the stack trace, variable scope, and parameter list are not correct. Stepping out of the prolog and into the actual function updates the stack trace and variable information kept by the debugger.

Single-stepping through function prologs that initialize large local variables is slow. As a workaround, use the `next` command.

# 19 Debugging Multithreaded Applications

This chapter explains how to use the Ladebug debugger to debug multithreaded programs that use DECthreads or kernel threads.

## 19.1 Thread Levels (DECthreads and Native Threads)

The debugger supports two levels of threads:

- DECthreads
- Native threads (also known as kernel threads and machine-level threads); these are threads native to the Digital UNIX system where the application is running

The debugger variable `$threadlevel` tells the debugger to view (interpret) the application threads as DECthreads threads or native threads. Depending on the setting of `$threadlevel`, only the DECthreads or the native threads are recognized by the debugger.

(For complete information on using DECthreads, see the *DECthreads Reference Manual*.)

By default, the `$threadlevel` variable is set to `decthreads` if the debuggee application is multithreaded and is using DECthreads. Otherwise, the `$threadlevel` is set to `native`. This happens each time an application is loaded into the debugger via the command line, the `load` command, or the `attach` command.

For core file debugging, the `threadlevel` is always set to `native`.

You can switch the threads debugging mode from `native` to `decthreads` (or the reverse) by setting `$threadlevel` appropriately. Use the debugger command `set $threadlevel`, as follows:

```
set $threadlevel="decthreads"
set $threadlevel="native"
```

## 19.2 Thread Identification

The valid values of thread identifiers for native threads and DECthreads are as follows:

- Native threads — a mach port id (an integer value)
- A DECthreads sequence number (an integer value)

The thread identifier is interpreted according to the `$threadlevel`.

The debugger variable `$curthread` contains the thread identifier of the current thread. The `$curthread` value is updated when program execution stops or completes.

The current thread context can be modified by setting `$curthread` to a valid thread identifier. This is equivalent to issuing the `thread` *`thread_identifier`* command.

When there is no process or program, `$curthread` is set to 0.

The debugger variable `$tid` is the same as `$curthread`. (See Chapter 22.)

## 19.3 Thread Commands

The categories of thread commands are as follows:

- Context
- Control
- Information
- Modification

These commands are discussed in the following sections.

### 19.3.1 Thread Context Commands

You can use the `thread` command to identify or set the current thread context. The syntax is as follows:

**thread** [thread_identifier]

If you supply a thread identifier, the debugger sets the current context to the thread you specify. If you omit the thread identifier, the debugger displays the current thread context. See Section 19.2 for a list of valid thread identifier values.

The debugger interprets the thread identifier as a DECthreads or kernel thread identifier depending on the value of the debugger variable `$threadlevel`.

The thread context can also be modified by setting the debugger variables `$curthread` or `$tid.` (See Section 19.2.)

### 19.3.2 Thread Control Commands

These commands control the execution of one or more threads in a process. You can stop and resume execution of some or all of the threads in the application.

#### 19.3.2.1 Setting Breakpoints in Multithreaded Applications

Use one of the following `stop` commands to set breakpoints in specific threads:

l)**stop** [variable] [**thread** thread_identifier_list] [**at** line_number] [**if** expression]
**stop** [variable] [**thread** thread_identifier_list] [**in** function] [**if** expression]

The *thread_identifier_list* parameter identifies one or more threads of the current debugging level.

A thread identifier is treated as one of the conditions on the breakpoint. When the `$threadlevel` mode is changed, this condition may no longer be true.

If you list one or more thread identifiers, the debugger sets a breakpoint only in those threads that you specify. If you omit the thread identifier specification, the breakpoint is set at the process level.

#### 19.3.2.2 Setting Tracepoints in Multithreaded Applications

Use one of the following `trace` or `when` command syntaxes to set tracepoints in specific threads:

**trace** [variable] [**thread** thread_identifier_list] [**at** line_number] [**if** expression]
**trace** [variable] [**thread** thread_identifier_list] [**in** function] [**if** expression]
**when** [variable] [**thread** thread_identifier_list] [**at** line_number] [**if** expression] {command [; . . . ]}
**when** [variable] [**thread** thread_identifier_list] [**in** function] [**if** expression] {command [; . . . ]}

If you list one or more thread identifiers, the debugger sets a tracepoint only in those threads that you specify. If you omit the thread identifier specification, the debugger sets a tracepoint at the process level.

#### 19.3.2.3 Stepping Individual Threads

Use one of the following commands to step while putting all other threads on hold:

**step**
**stepi**
**next**
**nexti**

The debugger steps only the current thread.

#### 19.3.2.4 Resuming Thread Execution

You can use the `cont` command to resume execution of the current thread that was put on hold (for example, at a breakpoint). As the current thread resumes, all other threads continue by default. The syntax is as follows:

**cont** [signal]

If you specify a signal, the program continues execution with that signal. The signal value can be either a signal number or a string name (for example, SIGSEGV). The default is 0, which allows the program to continue execution without specifying a signal.

## 19.3.3 Thread Information Commands

Thread information commands let you view information available from the debugger about the threads in your application. The information displayed may vary depending on whether the `$threadlevel` variable is set to `decthreads` or `native`.

To obtain the maximum detail, set the `$verbose` debugger variable to a value of 1. For more information about `$verbose`, see Section 10.13.

#### 19.3.3.1 Thread Queries

You can use the `show thread` command to list all the threads known to the debugger. The syntax is as follows:

**show thread** [thread_identifier_list]

If you specify one or more thread identifiers, the debugger displays information about those threads. If you do not specify any thread identifiers, the debugger displays information for all known threads. For example:

```
(ladebug)  print $threadlevel
‘‘decthreads’’
(ladebug)  show thread
Id   State      Substate   Policy  Prior Name
 ---  -------   ---------- --------- ---- ------------
  1   running              throughput 19  default thread
  3   running              throughput 19  <pthread user@0x1400005d8
* 4   stopped              throughput 19  <pthread user@0x1400005e8
  5   running              throughput 19  <pthread user@0x1400005f8
  6   running              throughput 19  <pthread user@0x140000608
  2   terminated  exited   throughput 19  <pthread user@0x1400005c8
(ladebug)  set $threadlevel="native"
(ladebug)  print $threadlevel
‘‘native’’
(ladebug)  show thread
         Id           State
     -----------      ------
> 0xffffffff81ad6f00  running
  0xffffffff81ad72c0  running
* 0xffffffff81b3f2c0  stopped  at 0x12001410 main(....) : 24
  0xffffffff81ad7a40  running
  0xffffffff81b16f00  running
  0xffffffff81b50000  running
```

Use the `show thread with state` command to list threads in a specific state, such as threads that are currently blocked. The possible states depend on whether you have DECthreads or native threads.

The following state values apply to DECthreads (see the DECthreads documentation for the meaning of the states):

```
ready
blocked
running
terminated
detached
```

The following state values apply to native threads:

```
stopped
running
terminated
```

The syntax is as follows:

**show thread** [thread_identifier_list] **with state = = ready**
**show thread** [thread_identifier_list] **with state = = blocked**
**show thread** [thread_identifier_list] **with state = = running**
**show thread** [thread_identifier_list] **with state = = terminated**
**show thread** [thread_identifier_list] **with state = = detached**

**show thread** [thread_identifier_list] **with state == stopped**

You can use the `where` command to display the stack trace of current threads. You can specify one or more threads or all threads. The syntax is as follows:

**where** [number] [**thread** [thread_identifier_list | **all** |*****]]

The `where` command displays the stack trace of currently active functions, for the current thread.

The `where thread thread_identifier_list` command displays the stack trace(s) of the specified thread(s).

The `where thread all` and the `where thread *` commands are equivalent; they display the stack traces of all threads.

Include the optional `number` argument to see a specified number of levels from the top of the stack. (Each active function is designated by a number that can be used as an argument to the `func` command. The top level on the stack is 0; so if you enter the command `where 3`, you will see levels 0, 1, and 2.) If you do not specify the `number` argument, you will see all levels.

The `print` command evaluates an optional expression in the context of the current thread and displays the result. The syntax is as follows:

**print** expression [ . . . ]

The `call` command evalutes an expression in the context of the current thread and makes the call in the context of the current thread. The syntax is as follows:

**call** expression

The `printregs` command prints the registers for the current thread. The syntax is as follows:

**printregs**

#### 19.3.3.2 Condition Variable Queries (DECthreads Only)

If your `$threadlevel` is `decthreads,` you can use the `show condition` command to list information about currently available condition variables. The syntax is as follows:

**show condition** [condition_identifier_list] [**with state == wait**]

If you supply one or more condition variable identifiers, the debugger displays information about those condition variables that you specify, provided that the list matches the identity of currently available condition variables. If you omit the condition variable identifier specification, the debugger displays information about all condition variables currently available.

Example 19–1 shows the output from a simple `show condition` command.

**Example 19–1 Displaying Condition Variable Information**

```
(ladebug) show condition
Condition variable thread 1 join 1 (0x14001dcf0)
Condition variable thread 1 wait 2 (0x14001de40)
Condition variable Last thread CV 3 (0x14001ffb8)
Condition variable thread 2 join 6 (0x140020af0)
Condition variable thread 2 wait 7 (0x140020c40)
Condition variable thread 3 join 9 (0x140021108)
Condition variable thread 3 wait 10 (0x140021258)
(ladebug)
```

If you specify `with state == wait`, the debugger displays information exclusively for the condition identifiers that have one or more threads waiting on them.

If `$verbose` is set to 1, Ladebug also displays the sequence number of the threads waiting on the condition variables.

If the debuggee application has no DECthreads or the `$theadlevel` is set to `native`, an appropriate message is issued.

#### 19.3.3.3 Mutex Queries for DECthreads

If your `$threadlevel` is `decthreads`, you can use the `show mutex` command to list information about currently available mutexes. The syntax is as follows:

**show mutex** [mutex_identifier_list] [**with state == locked**]

If you supply one or more mutex identifiers, the debugger displays information about only those mutexes that you specify, provided that the list matches the identity of currently available mutexes. If you omit the mutex identifier specification, the debugger displays information about all mutexes currently available.

You can specify `with state == locked` to display information exclusively for locked mutexes.

If `$verbose` is set to 1, Ladebug also displays the sequence number of the threads locking the mutex.

Example 19–2 shows the output from a simple `show mutex` command.

**Example 19–2 Displaying Mutex Information**

```
(ladebug) show mutex
Mutex thread 1 lock 14 (0x14001dc48), type fast, unlocked
Mutex thread 1 wait 15 (0x14001dd98), type fast, unlocked
Mutex thread 2 lock 69 (0x140020a48), type fast, unlocked
Mutex thread 2 wait 70 (0x140020b98), type fast, unlocked
(ladebug)
```

If the debuggee application has no DECthreads or the `$theadlevel` is set to `native`, an appropriate message is issued.

## 19.4 An Example of Debugging a Multithreaded Program

The following example shows the use of Ladebug commands to debug a multithreaded program. For more information on this test program, `prime_numbers`, see the *DECthreads Reference Manual*.

```
Welcome to the Ladebug Debugger Version 3.0
------------------
object file name: thread_prime_numbers
Reading symbolic information ...done
(ladebug) record io out1
(ladebug) stop at 432
[#1: stop at "thread_prime_numbers.c":43 ]
(ladebug) run3
[1] stopped at [prime_search:43 0x120001bb4]
     43     while (not_done)
(ladebug)thread4
Thread 2 (running) "<pthread user@0x140000758>"
  Scheduling: throughput policy at priority 19
  Stack: 0x3ef80; base is 0x40000, guard area at 0x39fff
  General cancelability enabled, asynch cancelability disabled
(ladebug)print my_number5
0
(ladebug)show thread with state == running6
Thread 2 (running) "<pthread user@0x140000758>"
  Scheduling: throughput policy at priority 19
  Stack: 0x3ef80; base is 0x40000, guard area at 0x39fff
  General cancelability enabled, asynch cancelability disabled
Thread 3 (running) "<pthread user@0x140000768>"
  Scheduling: throughput policy at priority 19
  Stack: 0x2e000; base is 0x2e000, guard area at 0x27fff
  General cancelability enabled, asynch cancelability disabled
Thread 4 (running) "<pthread user@0x140000778>"
  Scheduling: throughput policy at priority 19
  Stack: 0x1e000; base is 0x1e000, guard area at 0x17fff
  General cancelability enabled, asynch cancelability disabled
Thread 5 (running) "<pthread user@0x140000788>"
  Scheduling: throughput policy at priority 19
  Stack: 0x8a000; base is 0x8a000, guard area at 0x83fff
  General cancelability enabled, asynch cancelability disabled
Thread 6 (running) "<pthread user@0x140000798>"
  Scheduling: throughput policy at priority 19
  Stack: 0x78000; base is 0x78000, guard area at 0x71fff
  General cancelability enabled, asynch cancelability disabled
(ladebug)show thread with state == blocked7
Thread 1 (blocked) "default thread"
  Waiting on condition variable 4 using mutex 81
  Scheduling: throughput policy at priority 19
  Stack: 0x0 (default stack)
  !! Thread is not on stack !!
  General cancelability enabled, asynch cancelability disabled
(ladebug) where8
>0  0x120001bb4 in prime_search(arg=0x0) thread_prime_numbers.c:43
#1  0x3ff80c5cdf0 in cma__thread_base() ../../../../../src/usr/ccs/lib
/DECthreads/COMMON/cma_thread.c:1547
(ladebug) thread 49
Thread 4 (running) "<pthread user@0x140000778>"
  Scheduling: throughput policy at priority 19
  Stack: 0x1e000; base is 0x1e000, guard area at 0x17fff
  General cancelability enabled, asynch cancelability disabled
(ladebug)where10
>0  0x3ff80c5cd10 in cma__thread_base() ../../../../../src/usr/ccs/lib
/DECthreads/COMMON/cma_thread.c:1498
(ladebug) thread 111
Thread 1 (blocked) "default thread"
  Waiting on condition variable 4 using mutex 81
  Scheduling: throughput policy at priority 19
  Stack: 0x0 (default stack)
  !! Thread is not on stack !!
  General cancelability enabled, asynch cancelability disabled
(ladebug) where12
>0  0x3ff808edf14 in msg_receive_trap() /usr/build/osf1/goldos.bld/export
/alpha/usr/include/mach/syscall_sw.h:74
#1  0x3ff808e4764 in msg_receive() ../../../../../src/usr/ccs/lib/libmach
/msg.c:95
#2  0x3ff80c63d60 in cma__vp_sleep() ../../../../../src/usr/ccs/lib/DECthreads
/COMMON/cma_vp.c:1569
#3  0x3ff80c4c5ec in cma__dispatch() ../../../../../src/usr/ccs/lib/DECthreads
/COMMON/cma_dispatch.c:994
#4  0x3ff80c43e00 in cma__int_wait() ../../../../../src/usr/ccs/lib/DECthreads
/COMMON/cma_condition.c:2531
#5  0x3ff80c5c41c in cma_thread_join() ../../../../../src/usr/ccs/lib
/DECthreads/COMMON/cma_thread.c:926
#6  0x3ff80c5485c in pthread_join() ../../../../../src/usr/ccs/lib/DECthreads
/COMMON/cma_pthread.c:2294
#7  0x120002104 in main() thread_prime_numbers.c:121
```

```
(ladebug) thread 2 13
Thread 2 (running) "<pthread user@0x140000758>"
  Scheduling: throughput policy at priority 19
  Stack: 0x3ef80; base is 0x40000, guard area at 0x39fff
  General cancelability enabled, asynch cancelability disabled
(ladebug) step 14
stopped at [prime_search:45 0x120001bbc]
     45  pthread_testcancel();
(ladebug) cont 15
[1] stopped at [prime_search:43 0x120001bb4]
     43     while (not_done)
(ladebug) thread 16
Thread 3 (running) "<pthread user@0x140000768>"
  Scheduling: throughput policy at priority 19
  Stack: 0x2cf80; base is 0x2e000, guard area at 0x27fff
  General cancelability enabled, asynch cancelability disabled
(ladebug) where 17
>0  0x120001bb4 in prime_search(arg=0x1) thread_prime_numbers.c:43
#1  0x3ff80c5cdf0 in cma__thread_base() ../../../../../src/usr/ccs/lib
/DECthreads/COMMON/cma_thread.c:1547
(ladebug) show mutex (1, 2, 3) 18
Mutex 1 (fast) "default attrs mutex" is not locked
Mutex 2 (fast) "known attr list" is not locked
Mutex 3 (fast) "known mutex list" is not locked
(ladebug) show condition (5, 12, 15) 19
Condition variable 5, "thread 2 wait" has no waiters
Condition variable 12, "thread 6 join" has no waiters
(ladebug) delete 1 20
(ladebug) cont 21
Thread 0 terminated
Thread 1 terminated normally
Thread 2 terminated
Thread 3 terminated
Thread 4 terminated
The list of 110 primes follows:
1, 3, 5, 7, 11, 13, 17, 19, 23, 29,
 31, 37, 41, 43, 47, 53, 59, 61, 67,
 71, 73, 79, 83, 89, 97, 101, 103, 107,
 109, 113, 127, 131, 137, 139, 149, 151, 157,
 163, 167, 173, 179, 181, 191, 193, 197, 199,
 211, 223, 227, 229, 233, 239, 241, 251, 257,
 263, 269, 271, 277, 281, 283, 293, 307, 311,
 313, 317, 331, 337, 347, 349, 353, 359, 367,
 373, 379, 383, 389, 397, 401, 409, 419, 421,
 431, 433, 439, 443, 449, 457, 461, 463, 467,
 479, 487, 491, 499, 503, 509, 521, 523, 541,
 547, 557, 563, 569, 571, 577, 587, 593, 599,
 601
Thread has finished executing
(ladebug) quit
```

1 The `record io` command saves both debugger input and output to a file named `out` in this example.

2 The `stop at` command suspends program execution at the specified line (43 in the example) for any thread.

3 The `run` command starts program execution.

4 The `thread` command identifies or sets the current thread context. Because the thread identification is omitted, the debugger displays the current thread context.

5 The `print` command displays the value of the object `my_number` for the thread that stopped.

6 The `show thread with state == running` command displays threads that are running.

7 Only the blocked threads are displayed.

8 The `where` command displays the stack trace of currently active functions for the current thread (2).

9 The `thread` command sets the current thread context to thread 4.

10 The stack trace of thread 4 is displayed.

11 The `thread` command sets the current thread context to 1.

12 The stack trace of thread 1 is displayed.

13 The `thread` command sets the current thread context to 2.

14 The `step` command executes one line of source code for thread 2 only.

15 The `cont` command resumes program execution until a breakpoint, signal, error, or end of the program is encountered.

16 The `thread` command identifies or sets the current thread context. Because the thread identification is omitted, the debugger determines that the current thread is 3. Note that it reached line 43 for thread 3.

17 The `where` command displays the stack trace of currently active functions in thread 3.

18 The `show mutex (1, 2, 3)` command shows that none of the specified mutexes is locked.

19 The `show condition (5, 12, 15)` command shows that variables 5 and 12 have no threads waiting. Condition 15 does not exist; therefore, no information is shown.

20 The `delete 1` command removes the specified breakpoint or trace.

21 The `cont` command resumes program execution. When the program finishes execution, the threads are terminated and the result is displayed.

# 20 Debugging Multiprocess Applications

With Ladebug, you can debug more than one program or process. This chapter explains how to:

- Bring a process under debugger control
- Set the current process context
- Display a list of processes
- Load images and core files
- Remove process information from the debugger
- Debug programs that fork a child process and/or exec a program

In addition, it presents a sample multiprocess debugging session.

For information about multiprocess debugging from the window interface, see Chapter 5.

## 20.1 Bringing a Process Under Debugger Control

You can bring a process into debugger control in the following ways:

- From the command line, using various Ladebug command line options and arguments (for example, specifying a core file for core file debugging or the `-k` flag for local kernel debugging).
- From the Ladebug prompt, you can load a program using the `load` command or attach to a running process using the `attach` command.
- When the Ladebug variable `$catchforks` is set to 1, Ladebug keeps track of newly forked processes.

For more information on the various ways to invoke Ladebug, see Section 1.4.

## 20.2 Displaying a List of Processes

Use the `show process` command to display the currently executing process or all processes in your application. The syntax is as follows:

```
show process
show process *
show process all
```

If you specify the `show process` command without a qualifier, Ladebug displays information for the current process only. Using the asterisk (*) or the `all` option displays information for all processes. For each process listed, Ladebug shows the process ID, image file, the number of threads, and state.

## 20.3 Setting the Current Process

After bringing a process under debugger control, you can switch between processes using the `process` command. This sets the *current* process. The syntax is as follows:

```
process
process [process_id | image_file | debugger_variable]
```

Specify a specific process using the process ID number or the name of the image. Ladebug sets the current process context to the process ID or the process that runs the binary image. If there are more than one processes running the same binary image, Ladebug warns you and leaves the process context unchanged.

The debugger variables `$childprocess` and `$parentprocess` can also be specified in place of the process ID. (Ladebug automatically sets these variables when an application forks a child process.)

The `process` command without any argument displays information for the current process only.

## 20.4 Loading Image and Core Files

The `load` command reads the symbol table information of an image file and a core file.

To debug a core file, specify the name of the core file with the image file. After loading a program, specify the `run` command to start execution. The syntax is as follows:

```
load [image_file [core_file]]
```

## 20.5 Removing Process Information from the Debugger

Use the `unload` command to remove the process-related information. The syntax of the `unload` command is as follows:

**unload** [process_id | image_file]

The `unload` command removes the symbol table information if the debuggee process isn't running or stopped. You can specify the `detach` to release control of the running process or the `kill` command to terminate the debugger process if the process is created by Ladebug.

If you do not specify a process ID or image file, Ladebug unloads the current process.

## 20.6 Sample Multiprocess Debugging Session

Example 20–1 demonstrates the use of Ladebug commands to debug a multiprocess application.

In the first part of the program, the `process` command shows the current process. The `load` lets you load an image or core file. Specifying `show process all` displays a list of processes, running or stopped.

**Example 20–1 Debugging a Multiprocess Application - Loading an Image File and Showing Processes**

```
$ ladebug
Welcome to the Ladebug Debugger Version 4.0-9
(ladebug) process
There is no current process.
  You may start one by using the ‘load’ or ‘attach’ commands.
(ladebug) load a.out
Reading symbolic information ...done
(ladebug) process1
Current Process: localhost:18340 (a.out).
(ladebug) show process all2
>localhost:18340 (a.out) Unstarted.
(ladebug) load file-func
Reading symbolic information ...done
(ladebug) process
Current Process: localhost:18551 (file-func).3
(ladebug) show process all4
 localhost:18340 (a.out) Unstarted.
>localhost:18551 (file-func) Unstarted.
(ladebug) process 183405
(ladebug) process
Current Process: localhost:18340 (a.out).
(ladebug) list 16
      1
      2 int main(int argc, char* argv[])
      3 {
      4  int a = sizeof(**argv);
      5  int b = sizeof(+(**argv));
      6  int c = sizeof(-(**argv));
      7  return a+b+c;
      8 }
```

**1** Ladebug sets the current process.

**2** Ladebug shows information for all processes. In this case, there is only one process.

**3** After loading `file-func` , Ladebug displays a new current process (process 18551).

**4** Ladebug now shows two processes. The arrow (>) indicates the current process.

**5** When you specify the process ID, Ladebug switches to that process. In this case, it switches to process 18340.

**6** Process 18340 is now the current process. Ladebug lists the process' source code.

Switching between processes sets the current process context, as shown in Example 20–2.

**Example 20–2 Debugging a Multiprocess Application - Switching Between Processes**

```
(ladebug) process 185511
(ladebug) process
Current Process: localhost:18551 (file-func).
(ladebug) list 12
      1
      2
      3 #include <stdio.h>
      4
      5 int gfi = 100;
      6
      7 int g1 = 10;
      8
      9 int funcA(i) {
     10
     11     int a, x;
     12     x = 1;
     13     a = i + g1 + x;
     14     return a;
     15 }
     16
     17 int funcB(i) {
     18
     19     int a;
     20     a = i;
     21     return a;
```

**1** Ladebug toggles back to process 18551 and makes it the current process.

**2** Ladebug shows the source code for process 18551.

## 20.7 Debugging Programs That Fork and/or Exec

This section describes Ladebug's support for debugging programs that fork a child process and/or execs a program.

### 20.7.1 Predefined Debugger Variables for Fork/Exec Debugging

Ladebug contains the following predefined variables that you set for debugging a program that fork and/or execs. By default, the settings are turned off. When activated, the settings apply to all processes you debug.

- `$catchexecs`— When set to 1, this variable instructs the debugger to notify the user and stop the program when a program execs. The default is 0.
- `$catchforks`— When set to 1, this variable instructs the debugger to notify the user when a program forks a child process. The child process stops and is brought under debugger control. By default, the parent process is not stopped. The default is 0.
- `$stopparentonfork`—When set to 1, this variable instructs the debugger to stop the parent process when a program forks a child process. The default is 0.

When a fork occurs, Ladebug automatically sets the debugger variables `$childprocess` and `$parentprocess` to the new child or parent process ID. All examples in this section assume these variables are set.

### 20.7.2 Debugging Programs That Fork Child Processes

Set `$catchforks` to 1 to instruct Ladebug to to keep track of newly forked processes when a child process is forked. Ladebug stops the child process immediately after the fork occurs. The child process inherits all breakpoints from the parent process. It also inherits the signals list for the `catch/ignore` commands from the parent process.

The child process runs the same image file as the parent process and the process context is unchanged. You can switch to the child process using the `process` command.

When the child process finishes, you cannot rerun the child process. By setting the process context to the parent process, you can rerun the program .

#### 20.7.2.1 Setting the Predefined Variables

Before you debug you application, you can check the setting of the predefined variables using the `set` command, as shown in Example 20–3. The default settings for `$catchforks`, `$catchexecs`, and `$stoponparentfork` are all 0. In Example 20–3, these variables appear highlighted for illustration.

**Example 20–3 Default Settings for Predefined Variables**

```
$ ladebug mp-fork
Welcome to the Ladebug Debugger Version 4.0-10
------------------
object file name: mp-fork
Reading symbolic information ...done
(ladebug) set
$ascii = 1
$beep = 1
$catchexecs = 0
$catchforks = 0
$curevent = 0
$curfile = "mp-fork.c"
.
.
.
$stopparentonfork = 0
$threadlevel = "decthreads"
$verbose = 0
```

If your programs frequently fork or exec, you may want to set these variables in your `.dbxinit` initialization file.

#### 20.7.2.2 Scenario for Debugging a Forked Process with the Parent Process Running

To instruct Ladebug to report when the program forks a child process, set the `$catchforks` predefined variable to 1, as follows:

```
(ladebug) set $catchforks=1
```

In Example 20–4, when you run the program, Ladebug notifies you that the child process has stopped. The parent process continues to run.

**Example 20–4 Debugging a Forked Process - Showing the Child Process**

```
(ladebug) run
Process 200 forked.  The child process is 201.
Process 201 stopped on fork. 1
stopped at [void main(void):14 0x120001248] 2
     14   if ((pid=fork()) == 0)
Process has exited with status 18 3
(ladebug) show process all 4
>localhost:200 (mp-fork) dead.
    \_localhost:201 (mp-fork) stopped.
(ladebug)
```

1 Indicates that the child process has stopped.

2 Tells where the child process stopped.

3 Indicates that the parent process, which was not stopped, has completed execution.

4 Shows that the child process (process 201) has stopped and the parent process has completed execution. The parent process (process 200) remains in the current context, as indicated by the arrow (>).

In Example 20–5, the process context is changed. Listing the source code shows the source for the child process.

**Example 20–5 Debugging a Forked Process - Changing the Process Context**

```
(ladebug) process 201 1
(ladebug) show process all
 localhost:200 (mp-fork) dead.
>  \_localhost:201 (mp-fork) stopped.  2
(ladebug) process
Current Process: localhost:201 (mp-fork).
(ladebug) list 3
     15     {
     16       printf("about to exec\n");
     17       execlp("./c_whatis", "./c_whatis", NULL);
     18       perror(" execve failed.");
     19     }
     20
     21   else if (pid != -1)
     22     {
     23       printf("in parent process\n");
     24     }
     25
     26   else
     27     {
     28       printf("Error in fork!");
     29       exit(0);
     30     }
     31 }
```

**1** The current process context is changed to the child process (process 201).

**2** The arrow now indicates process 201 is the current process.

**3** Ladebug lists the source code for the current process. Notice that it began the listing from the line where the parent process forked.

You can continue to debug the current process (the child process). When the child process finishes, you cannot rerun the child process. By setting the process context to the parent process, you can rerun the program, as shown in Example 20–6.

**Example 20–6 Debugging a Forked Process - Rerunning the Program**

```
(ladebug) next
stopped at [void main(void):16 0x12000125c]
     16       printf("about to exec\n");
(ladebug)  next
about to exec
stopped at [void main(void):17 0x120001274]
     17       execlp("./c_whatis", "./c_whatis", NULL);
(ladebug)  next
result of foo = 5040
result of foo = 5040
result of baz = 720
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(6) = 720
factorial(7) = 5040
factorial(8) = 40320
abcdefghij
abcdefghij
Process has exited with status 0 1
(ladebug) show process all
 localhost:200 (mp-fork) dead.
>  \_localhost:201 (mp-fork) dead.
(ladebug) rerun 2
Error: cannot restart existing process.

(ladebug) process 200 3

(ladebug) rerun 4
Process 200 forked.  The child process is 201.
Process 201 stopped on fork.
stopped at [void main(void):14 0x120001248]
     14   if ((pid=fork()) == 0)
in parent process
Process has exited with status 18
(ladebug)
```

**1** Indicates that the child process has finished executing.

**2** You cannot rerun the child process.

**3** Setting back to the parent process (process 200), you can now specify `rerun`.

**4** The program reruns; a new child process is created.

#### 20.7.2.3 Scenario for Debugging a Forked Process with the Parent Process Stopped

In this scenario, you set the predefined variable `$catchforks` and `$stopparen-tonfork` to 1. Setting `$catchforks` to 1 tells Ladebug to notify the user when the program forks and stop the child process. By setting `$stopparentonfork` to 1, the parent process also stops when the program forks a child process. The variable `$stopparentonfork` has no effect when `$catchforks` is set to 0.

To instruct Ladebug to report when the program forks a child process and stop the parent and child processes, set the variables `$catchforks` and `$stopparentonfork` to 1, as follows:

```
(ladebug) set $catchforks=1
(ladebug) set $stopparentonfork=1
```

In Example 20–7, Ladebug stops the parent process when it forks the child process. The current context is the parent process. You can change the process context to the child process using the `process` command.

**Example 20–7 Debugging a Forked Process with Parent and Child Processes Stopped**

```
(ladebug) run
Process 200 forked.  The child process is 201.
Process 201 stopped on fork.
stopped at [void main(void):14 0x120001248]
     14   if ((pid=fork()) == 0)
Process 200 stopped on fork.
stopped at [void main(void):14 0x120001248] 1
     14   if ((pid=fork()) == 0)
(ladebug) show process all
>localhost:200 (mp-fork) stopped.
   \_localhost:201 (mp-fork) stopped.
(ladebug) process 201 2
(ladebug) show process all
 localhost:200 (mp-fork) stopped.
>  \_localhost:201 (mp-fork) stopped.
(ladebug) list 3
     15     {
     16       printf("about to exec\n");
     17       execlp("./c_whatis", "./c_whatis", NULL);
     18       perror(" execve failed.");
     19     }
     20
     21   else if (pid != -1)
     22     {
     23       printf("in parent process\n");
     24     }
     25
     26   else
     27     {
     28       printf("Error in fork!");
     29       exit(0);
     30     }
     31 }
```

**1** Shows that the parent process has stopped at line 14.

**2** Changes the current process context to the child process (process 201).

**3** Lists the source code for the current process (the child process).

In Example 20–8, you continue to debug the current process (the child process). When the child process finishes, you can switch to the parent process and continue debugging.

**Example 20–8 Debugging a Forked Process - Switching to the Parent Process**

```
(ladebug) next 1
stopped at [void main(void):16 0x12000125c]
   LDBGD$:[SLOVENKAI.LADEBUG.MANUAL]LADEBUG_CH_MULTIPROC.SDML;19
(ladebug) next
about to exec
sstopped at [void main(void):17 0x120001274]
     17       execlp("./c_whatis", "./c_whatis", NULL);
(ladebug) next
result of foo = 5040
result of foo = 5040
result of baz = 720
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(6) = 720
factorial(7) = 5040
factorial(8) = 40320
abcdefghij
abcdefghij
Process has exited with status 0 2
(ladebug) show process all
 localhost:200 (mp-fork) stopped.
>  \_localhost:201 (mp-fork) dead.
(ladebug) process 200
(ladebug) list 3
     15     {
     16       printf("about to exec\n");
     17       execlp("./c_whatis", "./c_whatis", NULL);
     18       perror(" execve failed.");
     19     }
     20
     21   else if (pid != -1)
     22     {
     23       printf("in parent process\n");
     24     }
     25
     26   else
```

(continued on next page)

**Example 20–8 (Cont.) Debugging a Forked Process - Switching to the Parent Process**

```
     27     {
     28       printf("Error in fork!");
     29       exit(0);
     30     }
     31 }
(ladebug) next
stopped at [void main(void):21 0x1200012b4]
     21   else if (pid != -1)
(ladebug) next
stopped at [void main(void):23 0x1200012c0]
     23       printf("in parent process\n");
(ladebug) cont
in parent process
Process has exited with status 18 4
```

1 Continues execution in the child process.

2 The child process has finished execution.

3 After switching to the parent process, you can now list its source code.

4 The parent process terminates.

### 20.7.3 Debugging a Process That Execs

Set `$catchexecs` to 1 to instruct Ladebug to notify the user when the program execs. The program stops before executing any user program code or static initializations that are passed to the exec system call. You can debug the newly exec'd program using basic debugging techniques. Ladebug keeps a history of the progression of the exec'd binary files.

In the following scenario, you set the predefined variable `$catchforks` and `$catchexecs` to 1. Ladebug will notify you when an exec occurs. Because `$catchforks` was set, you will be notified of any execs in the child process.

To instruct Ladebug to report when a program execs on the current process context, set the variables `$catchexecs` and `$catchforks` to 1, as follows:

```
(ladebug) set $catchexecs=1
(ladebug) set $catchforks=1
```

When you run the program in Example 20–9, Ladebug notifies you that an exec occurred on the current context and that the child process has stopped on the runtime-loader entry point.

**Example 20–9 Debugging a Process That Execs**

```
(ladebug) run
Process 200 forked.  The child process is 201.
Process 201 stopped on fork.
stopped at [void main(void):14 0x120001248]
     14   if ((pid=fork()) == 0)
in parent process
Process has exited with status 18
(ladebug) show process all
>localhost:200 (mp-fork) dead.
   \_localhost:201 (mp-fork) stopped.

(ladebug) process 201 1
(ladebug) list
     15     {
     16       printf("about to exec\n");
     17       execlp("./c_whatis", "./c_whatis", NULL); 2
     18       perror(" execve failed.");
     19     }
     20
     21   else if (pid != -1)
     22     {
     23       printf("in parent process\n");
     24     }
     25
     26   else
     27     {
     28       printf("Error in fork!");
     29       exit(0);
     30     }
     31 }
(ladebug) next
stopped at [void main(void):16 0x12000125c]
     16       printf("about to exec\n");
(ladebug) next
about to exec
stopped at [void main(void):17 0x120001274]
     17       execlp("./c_whatis", "./c_whatis", NULL);
(ladebug) next
The process 201 has execed the image ‘./c_whatis’. 3
stopped at [???:62 0x3ff8001c3b8] 4
     62     i = 1;
```

1 Ladebug sets the current process context to the child process.

2 Listing the source code, you can see the process is about to exec on line 17.

3 Ladebug notifies you when the exec executes.

**4** The child process is stopped on the runtime-loader entry point. The source display shows the code in the `main` routine.

In Example 20–10, you can set breakpoints in the current process (child process). Ladebug shows the current process and the current executing program.

### Example 20–10 Debugging a Process That Execs - Setting Breakpoints

```
(ladebug) stop in main 1
[#1: stop in int main(void) ]
(ladebug) c
[1] stopped at [int main(void):62 0x1200013a4]
     62     i = 1;
(ladebug) list
     63     foo();
     64     baz(x,3,x+1);
     65
     66     i = 1;
     67     printf("factorial(%d) = %d\n", i, factorial(i));
     68     i = 2; printf("factorial(%d) = %d\n", i, factorial(i));
     69     i = 3; printf("factorial(%d) = %d\n", i,
     70                   factorial(
     71                       i));
     72
     73     i
     74       =
     75         4;
     76     printf(
     77         "factorial(%d) = %d\n",
     78         i,
     79         factorial(i));
     80
     81     if (i < 5)
     82        i = 5;
     83     else
(ladebug) process 2
Current Process: localhost:201 (./c_whatis).
(ladebug) show process all
 localhost:200 (mp-fork) dead.
>  \_localhost:201 (mp-fork->./c_whatis) stopped. 3
(ladebug) next
stopped at [int main(void):63 0x1200013ac]
     63     foo();
(ladebug) next
result of foo = 5040
```

(continued on next page)

**Example 20–10 (Cont.) Debugging a Process That Execs - Setting Breakpoints**

```
stopped at [int main(void):64 0x1200013bc]
     64     baz(x,3,x+1);
(ladebug) next
result of foo = 5040
result of baz = 720
stopped at [int main(void):66 0x1200013dc]
     66     i = 1;
(ladebug) next
stopped at [int main(void):67 0x1200013e4]
     67     printf("factorial(%d) = %d\n", i, factorial(i));
(ladebug) next
factorial(1) = 1
stopped at [int main(void):68 0x12000141c]
     68     i = 2; printf("factorial(%d) = %d\n", i, factorial(i));
(ladebug) next
factorial(2) = 2
stopped at [int main(void):69 0x12000145c]
     69     i = 3; printf("factorial(%d) = %d\n", i,
(ladebug) step
stopped at [int factorial(int):8 0x1200011e8]
      8     switch (n) {
(ladebug) where
>0  0x1200011e8 in factorial(n=3) c_whatis.c:8
#1  0x120001470 in main() c_whatis.c:69
(ladebug) cont
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(6) = 720
factorial(7) = 5040
factorial(8) = 40320
abcdefghij
abcdefghij
Process has exited with status 0
(ladebug) show process all
 localhost:200 (mp-fork) dead.
>  \_localhost:201 (mp-fork) dead.
(ladebug)
```

1 You can set a breakpoint on the current process.

2 Shows the current process and the current executing program.

3 Shows the image file history as a progression of images.

# 21 Remote Debugging

This chapter describes debugging programs running on remote systems. A remote debugger consists of a server running on the target system and a client (the debugger) running on the host system. Once connected to the target system, you use Ladebug to debug your program in the same way you debug your programs running locally.

For a detailed description of writing a remote debugger server, see Appendix B.

## 21.1 Remote Debugging Environment

The remote debugging environment consists the following components that interact through the remote debugger protocol:

- Ladebug debugger, acting as the debugger client running on the host system
- A remote debugger server, running on the target system
- The program you want to debug

The functionality available in a remote debugging session depends in part on which debugger server you are using. For Digital UNIX Version 4.0-6, this server is the server for the EB64 Alpha Evaluation Board (see Section 21.6) or Digital UNIX, or a server that you write for your own Alpha environment (see Appendix B).

The Ladebug server provided with Digital UNIX Version 4.0-6 (or later) is not compatible with versions of the debugger earlier than Version 4.0-6. (This incompatibility affects you only if you use remote debugging.) The incompatibility stems from a security enhancement introduced in Version 4.0-6: The server checks that a connect request is from a privileged port.

## 21.2 Reasons for Remote Debugging

There are several cases in which you would use a remote debugger:

1. The target system does not have (or cannot run) a local debugger.

   The target system may be an embedded system that cannot support a debugger. You also may be debugging a part of the target's software that has to work before you can support a local debugger. A remote debugger server (that is, the software) requires less and different support from the target's operating system (if one exists) than the support required by a local debugger.

2. The target system does not have the resources to run a local debugger.

   The target system may not be able to run a local debugger and simultaneously run the program being debugged. The remote debugger server uses less resources (particularly memory) than a resident Ladebug debugger.

3. The source files for the program being debugged are not accessible from the target system.

   These sources are accessible from a host system from which you can remotely debug the target system.

4. The target system's screen or keyboard cannot be used for debugging.

   A screen or keyboard interface may not exist because they are used by the program being debugged. Another possibility is that the target system's physical location is different from the user's.

5. A local debugger may interfere with window-interface applications.

   On many interactive systems, it may be best to run a remote debugger. Alternatively, you may be able to run a local debugger that directs its input/output (I/O) to a screen on the remote system.

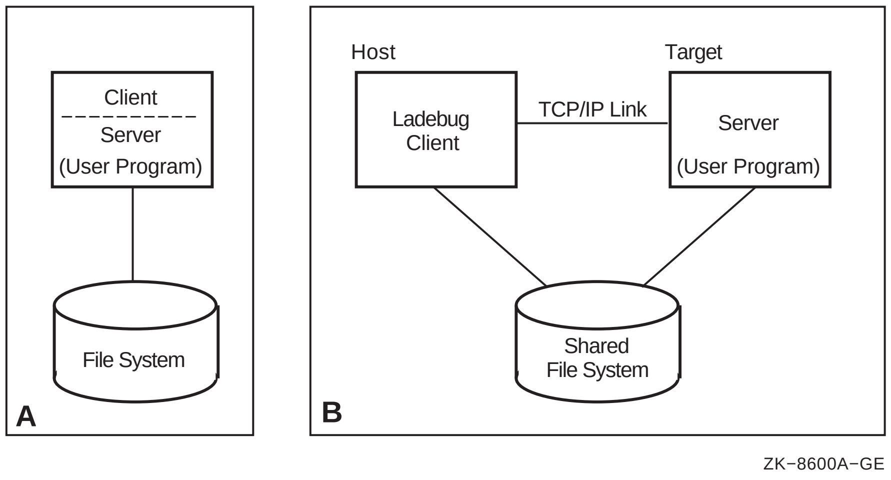
6. You may want to debug a process running on another machine.

## 21.3 Client/Server Model for Remote Debugging

Remote debugging with Ladebug uses a client/server model. In this model, the host system, or client, initiates a connection to the target hardware and server software, which processes client requests.

Ladebug supports remote debugging in various client/server configurations. Figure 21–1 shows two configurations that use a single file system.

**Figure 21–1 Client/Server Model with a Single or a Shared File System**



In configuration A, the client and server are implemented on a single machine which connects to a file system.

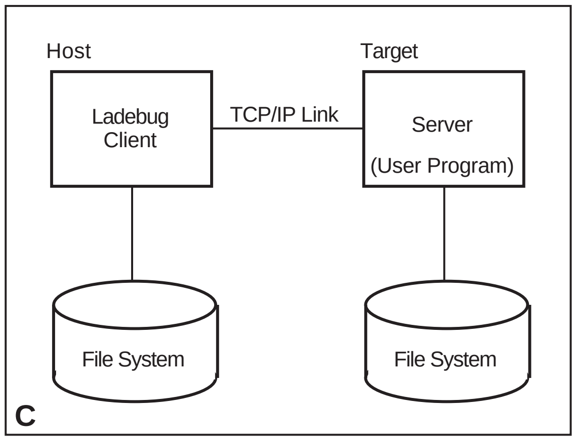
In configuration B, a host system client is connected to a remote server through TCP/IP. Both host and target systems share the same file system.

A host system client can also be connected to a remote server in a configuration that employs separate file systems, as shown in Figure 21–2.

**Note**

In this case, you must specify the `-rfn` option with the remote file name and have a locally accessible binary with debugger information.

**Figure 21–2 Client/Server Model with Separate File Systems**



ZK−8601A−GE

In configurations A, B, and C in Figure 21–1 and Figure 21–2, the user program resides on the target system. The host machine interacts with the target system in the following way:

1. After starting the server deamon, a remote session is established when the host system connects to the target system.
2. The Ladebug client on the host system interacts with the server on the target system to debug the user program. A new server is started for each user program being debugged.
3. The debugger terminates the remote server when you end the debugging session.

Table 21–1 describes the client/server concepts for remote debugging. Section 21.4 describes the tasks for remote debugging.

**Table 21–1 Client/Server Concepts for Remote Debugging**

| Client | Server |
|---|---|
| Is Ladebug debugger. | Is remote debugger server. |
| Runs on host system. | Runs on target hardware (for example, EB64 or Digital UNIX Alpha. ) |
| Makes requests to server. | Controls the process being debugged. |
| Is responsible for all access to source files and symbol table. | Is not responsible. |
| Uses debug protocol; sends protocol commands to the server. | Uses debug protocol; receives protocol commands and sends responses. |
| Contains information about the process being debugged. | Contains information about the processes' environment. |
| Does not control processes. | Server controls a single process; server deamon controls multiple processes (messages to the server containing a server ID). |

## 21.4 Tasks for Remote Debugging

This section describes general tasks to debug programs running on remote systems. The tasks include:

1. Starting the server daemon
2. Starting the debugger, user program, and connecting to the server
3. Debugging the user program
4. Exiting the debugger and disconnecting from the server

### 21.4.1 Starting the Server Daemon

The server daemon must be running on the target system before you can remotely debug programs on that target. You can start the server daemon either from a system startup file or interactively from the command line.

**Note**

Under certain conditions, running the server daemon negatively affects the security of your system. If you are running an old server, there may be a security problem (see the documentation for that server). An individual user ID can be protected by prohibiting remote access from a particular host (or from all hosts) in the `.rhosts` and the `hosts.equiv` files. On Digital UNIX machines, `.rhosts` must have `rw` privileges only for the owner, with no privileges for the group and others.

---

For example, to start the server daemon interactively and output system messages to a log file, log in as superuser then start the server daemon as follows:

```
$ /usr/bin/ladebug-server > ladebug-server.log &
Ladebug remote debug server deamon starting
/usr/bin/ladebug-server : server is servdb.ptl.dss.com (11.18.49.164)
...
```

### 21.4.2 Starting Ladebug

When you start Ladebug you also start the user program and connect to the server.

Use the `-rn` command-line option, which specifies the IP name or address of the machine on which the server deamon is running and on which you want your user program to run. Specifying this command-line option is the only difference between starting Ladebug to debug a remote application and starting Ladebug to debug a local one.

If you start Ladebug without specifying the process ID (`-pid`) , it starts a debuggee process in the remote node running the indicated image file. If you do not specify the user name on the remote node (`-ru`) , it uses the local user name.

For example, if you are connecting to the target system `servdb` to debug the user program `~/work/test/hello` and the file system is shared:

```
$  ladebug -rn servdb ~/work/test/hello
Welcome to the Ladebug Debugger Version 3.0
------------------
object file name: /usr/users/dss/work/test/hello
machine name: servdb
Reading symbolic information ...done
```

If you start Ladebug and specify a process ID, Ladebug connects to the process in the remote node running the process. If the specified process ID does not exist, the server returns an error and refuses connection.

**Note**

If you are connecting to a server that does not share the same file system, specify the `-rfn` option with the `-rn` option. See Section 21.5

for detailed descriptions of all the remote debugging command-line options.

---

### 21.4.3 Debugging the User Program

You debug a user program running on the target system the same way as you would a local program. Note the following differences:

- When debugging an already running process, the `run` and `rerun` commands are disabled since you do not need to start the process.
- When you specify the `quit` command to a remote debugger session, it does not terminate the process running on the target system. To do this, you must use the `kill` command.
- The `cont` command with a signal number is not supported.
- Ladebug provides less information when a process stops than when a local process stops.
- Additional messages may be displayed if problems are encountered communicating with the server.

The following example shows the result of running user program `~/work/test/hello` on remote node `servdb`:

```
(ladebug)  stop in main; run
[#1: stop in main ]
[1] stopped at [main:6 0x120001fa0]
      6   (void) printf("Hello, world !\n");
(ladebug)  cont
Thread has finished executing
(ladebug)
```

Note that the output of the program is not displayed after the `cont` command. With remote debugging, the program output is displayed on the target system. You can also redirect the output of the application to a log file.

The same program run locally would look like this:

```
$  ladebug ~/work/test/hello
Welcome to the Ladebug Debugger Version 3.0
------------------
object file name: /usr/users/dss/work/test/hello
Reading symbolic information ...done
(ladebug) stop in main; run
[#1: stop in main ]
s[1] stopped at [main:6 0x120001fa0]
      6   (void) printf("Hello, world !\n");
(ladebug) cont
Hello, world !
Thread has finished executing
(ladebug)
```

### 21.4.4 Exiting the Debugger and Disconnecting from the Server

The Ladebug `quit` command ends the remote debugger session and automatically disconnects from the server.

**Note**

The `quit` command does not terminate the process running on the target system. Use the `kill` command to terminate a running process and end the remote debugger session.

## 21.5 Command-Line Options for Remote Debugging

Table 21–2 lists the Ladebug command-line options that support remote debugging.

**Table 21–2 Command Line Options for Remote Debugging**

| Option/Qualifier | Meaning/Conditions |
|---|---|
| `-rn`[1] `node_or_address [,udp_port`[2]`]` | Specifies the internet node name or IP address of the machine on which the remote debugger server is running (that is, the node running the program to be debugged); optionally specifies the UDP port on which to connect the server. Either the node name or IP address is required; there is no default. |
| `-pid process_id` | Specifies the process ID of the process to be debugged. When you specify this option, Ladebug debugs a running process rather than loading a new process. |
| `-rfn`[1] `arbitrary_string` | Specifies the file name (or other identifier) of the image to be loaded on a remote system. This option defaults to the local image file name and it is passed to the remote system uninterpreted. Use only with `-rn`; do not combine with `-pid`. |
| `-rinsist` | Connects to a running remote process using the connect insist protocol message instead of the connect protocol message. This option functions as a request to the server to connect to the client, even if another client is already connected. (The previously connected client is disconnected.) Use only with with `-rn` and `-pid`. |
| `-ru username` | Specifies the user name to be used on the remote system. The default is the local user name. |

[1]Depending on your shell, it may be necessary to enclose this option in quotes to prevent the shell from interpreting the punctuation characters in the parameter.

[2]Current remote debugger servers use UDP port 410 (the default); older releases might use UDP port 21511.

The following examples show how to use the remote debugger command-line options.

### Example 1

```
$  ladebug -rn 1.2.3.4 -ru brown program1
```

Connects to the server on the node with IP address `1.2.3.4` and asks the server to load a process called `program1`. The local copy of the object file is also called `program1`. The user name on the remote node is `brown`.

**Example 2**

```
$  ladebug -rn EB64 -rfn '**process name A**' program3
```

Connects to the server on the node with IP name `EB64` and asks it to load the process called `'**process name A**'`. The local object file is called `program3`.

## 21.6 Example Remote Debugger Session Using the Evaluation Board Server

This section describes the steps you need to perform to debug remote processes running on the Evaluation Board Server [1]. The procedures described in Section 21.6.1 through Section 21.6.3 describe the simplest procedure for using Ladebug with the Evaluation Board Server. Since Ladebug can be started at any monitor breakpoint (not only when the program is at its initial entry point), many variations are possible.

### 21.6.1 Building an Executable File

To build the executable file for remote debugging:

1. Compile your source files using the `-g` option to save symbolic information.
2. Link the source files with the `-N` and `-Tx` options, where `x` is the load address for the executable file.
3. Use the `cstrip` utility to strip the `coff` header from the executable file. Keep the unstripped executable file.

### 21.6.2 Loading the Executable File and Starting the Server

To load the executable file and start the Ladebug server:

1. Set up the host Digital UNIX Alpha machine as the `bootp` server for the Evaluation Board Server (for example, the EB64).
2. Start the server.
3. Use the `bootadr` command on the monitor to set the boot address to the load address that was used when linking the executable file.
4. Use the `netload` command to load the stripped executable file across the ethernet.
5. Set a breakpoint at the entry point to the program with the `stop` command.
6. Start the program with the `go` command.

[1] The server includes the EB64, EB64+, and EB66 evaluation boards.

7. When the program reaches the breakpoint, enter the `LADBX` command to start the Ladebug server.

### 21.6.3 Starting Ladebug on the Host System

After starting the server, to start the Ladebug debugger on the host system:

1. Enter the following command:

   ```
   ladebug -rn <EB64> -pid 0 <exe>
   ```

   In this command, <exe> is the file name of the unstripped executable file and <EB64> is the internet name or address of the Evaluation Board Server.

2. After you finish debugging or want to return to the monitor's local command interface, enter the `quit` command. The `quit` command closes Ladebug and returns the monitor to its command prompt. The state of the program you are debugging is not affected by this action.

### 21.6.4 Special Conditions

The following conditions may arise when remote debugging using the Evaluation Board Server:

- If the program under server control disables interrupts for long periods of time, Ladebug may lose communication with the Evaluation Board Server.
- The Evaluation Board Server does not support the loading of programs under Ladebug control. As such, a program must be loaded by the monitor and then connected to Ladebug, as described in Section 21.6.2.

# 22 Kernel Debugging

Ladebug supports kernel debugging, which is a task normally performed by systems engineers or system administrators. A systems engineer might debug a kernel space program, which is built as part of the kernel and which references kernel data structures. A systems administrator might debug a kernel when a process is hung, or kernel parameters need to be examined or modified, or the operating system hangs, panics, or crashes. Kernel debugging aids in analyzing crash dumps.

### Security

You may need to be the superuser (`root` login) to examine either the running system or crash dumps. Whether or not you need to be the superuser depends on the directory and file protections for the files you attempt to examine.

### Compiling a Kernel for Debugging

Compilation of a kernel should be done without full optimization and without stripping the kernel of its symbol table information. Otherwise, your ability to debug the kernel is greatly reduced.

By default, compilation does not strip the symbol table information. By default, optimization is only partial. If you do not change these defaults, there should not be a problem.

**Adding or Deleting Symbol Table Information** From within the debugger, you can selectively add or delete symbol table information for a kernel image, with the `addstb` or `delstb` commands. These commands can be useful because symbol table information can impact debugger performance and take up considerable disk space. The syntax is as follows:

addstb kernel_image

delstb kernel_image

## Patching a Disk File

From within the debugger, you can use the `patch` command to correct bad data or instructions in an executable disk file. The text, initialized data, or read-only data areas can be patched. The `bss` segment cannot be patched because it does not exist in disk files.

The syntax is as follows:

patch expression1 = expression2

For example,

```
(ladebug)  patch @foo = 20
```

## Setting the Thread Context

The debugger variable `$tid` contains the thread identifier of the current thread. The `$tid` value is updated implicitly by the debugger when program execution stops or completes.

You can modify the current thread context by setting `$tid` to a valid thread identifier.

When there is no process or program, `$tid` is set to 0.

The debugger variable `$tid` is the same as `$curthread` except that `$tid` is used for kernel debugging.

## Summary and Additional Information

The remainder of this chapter briefly describes the use of Ladebug to

- Perform local kernel debugging
- Analyze crash dumps
- Perform remote kernel debugging with the `kdebug` debugger

You can find additional information on kernel debugging in

- The `kdbx(8)` reference page
- The `kdebug(8)` reference page
- The *Digital UNIX Kernel Debugging* manual

The kernel debugging functionality supported by Ladebug is very similar to the functionality described in the above-listed `dbx` sources, with the substitution of the term `ladebug` for the term `dbx` .

## 22.1 Local Kernel Debugging

When you have a problem with a process, you can debug the running kernel or examine the values assigned to system parameters. (It is generally recommended that you avoid modifying the value of the parameters, which can cause problems with the kernel.)

Invoke the debugger with the following command:

```
#  ladebug -k /vmunix /dev/mem
```

The `-k` flag maps virtual to physical addresses to enable local kernel debugging. The `/vmunix` and `/dev/mem` parameters cause the debugger to operate on the running kernel.

Now you can use Ladebug commands to display the current process identification numbers (process IDs), and trace the execution of processes. The following example shows the use of the command `kps` (which is the alias of the `show process` command) to display the process IDs:

```
(ladebug) kps

  PID     COMM
00000     kernel idle
00001     init
00014     kloadsrv
00016     update
  .
  .
  .
```

The Ladebug commands `cont`, `next`, `rerun`, `run`, `setting registers`, `step`, and `stop` are not available when you do local kernel debugging. (Stopping the kernel would also stop the debugger.)

If you want to examine the stack of, for example, the `kloadsrv` daemon, you set the `$pid` symbol to its process ID (14) and enter the `where` command, as in the following example (the spacing in the example has been altered to fit the page):

```
(ladebug) set $pid = 14
(ladebug) where

>  0 thread_block()
["/usr/sde/osf1/build/goldos.nightly1/src/kernel/kern/sched_prim.c":1623, 0xfffffc000043d77c]
   1 mpsleep(0xffffffff92586f00, 0x11a, 0xfffffc0000279cf4, 0x0, 0x0)
["/usr/sde/osf1/build/goldos.nightly1/src/kernel/bsd/kern_synch.c":411, 0xfffffc000040adc0]
   2 sosleep(0xffffffff92586f00, 0x1, 0xfffffc000000011a, 0x0, 0xffffffff81274210)
["/usr/sde/osf1/build/goldos.nightly1/src/kernel/bsd/uipc_socket2.c":654, 0xfffffc0000254ff8]
   3 sosbwait(0xffffffff92586f60, 0xffffffff92586f00, 0x0, 0xffffffff92586f00, 0x10180)
["/usr/sde/osf1/build/goldos.nightly1/src/kernel/bsd/uipc_socket2.c":630, 0xfffffc0000254f64]
   4 soreceive(0x0, 0xffffffff9a64f658, 0xffffffff9a64f680, 0x8000004300000000, 0x0)
["/usr/sde/osf1/build/goldos.nightly1/src/kernel/bsd/uipc_socket.c":1297, 0xfffffc0000253338]
   5 recvit(0xfffffc0000456fe8, 0xffffffff9a64f718, 0x14000c6d8, 0xffffffff9a64f8b8, 0xfffffc000043d724)
["/usr/sde/osf1/build/goldos.nightly1/src/kernel/bsd/uipc_syscalls.c":1002, 0xfffffc00002574f0]
   6 recvfrom(0xffffffff81274210, 0xffffffff9a64f8c8, 0xffffffff9a64f8b8, 0xffffffff9a64f8c8,
0xfffffc0000457570) ["/usr/sde/osf1/build/goldos.nightly1/src/kernel/bsd/uipc_syscalls.c":860,
0xfffffc000025712c]
   7 orecvfrom(0xffffffff9a64f8b8, 0xffffffff9a64f8c8, 0xfffffc0000457570, 0x1, 0xfffffc0000456fe8)
["/usr/sde/osf1/build/goldos.nightly1/src/kernel/bsd/uipc_syscalls.c":825, 0xfffffc000025708c]
   8 syscall(0x120024078, 0xffffffffffffffff, 0xffffffffffffffff, 0x21, 0x7d)
["/usr/sde/osf1/build/goldos.nightly1/src/kernel/arch/alpha/syscall_trap.c":515, 0xfffffc0000456fe4]
   9 _Xsyscall(0x8, 0x12001acb8, 0x14000eed0, 0x4, 0x1400109d0)
["/usr/sde/osf1/build/goldos.nightly1/src/kernel/arch/alpha/locore.s":1046, 0xfffffc00004486e4]
```

Examining the stack trace may reveal the problem. Then you can modify parameters, restart daemons, or take other corrective actions.

## The kdbx Interface

The `kdbx` interface is a crash analysis and kernel debugging tool. It serves as a front end to the Ladebug debugger. The `kdbx` interface is extensible, customizable, and insensitive to changes to offsets and sizes of fields in structures. The only dependencies on kernel header files are for bit definitions in flag fields.

The `kdbx` interface has facilities for interpreting various symbols and kernel data structures. It can format and display these symbols and data structures in the following ways:

- In a predefined form as specified in the source code modules that currently accompany the `kdbx` interface
- As defined in user-written source code modules according to a standardized format for the contents of the `kdbx` modules

The Ladebug commands (except signals such as Ctrl/P) are available when you use the `kdbx` interface. (Many of these commands have aliases that match `dbx` commands, for the convenience of users who are accustomed to debugging kernels with the `dbx` debugger.) In general, `kdbx` assumes hexadecimal addresses for commands that perform input and output.

The sections that follow explain using `kdbx` to debug kernel programs.

## Beginning a kdbx Session

Using the `kdbx` interface, you can examine either the running kernel or dump files created by the `savecore` utility. In either case, you examine an object file and a core file. For running systems, these files are usually /vmunix and /dev/mem, respectively. The `savecore` utility saves dump files it creates in the directory specified by the /sbin/init.d/savecore script. By default, the `savecore` utility saves dump files in the /var/adm/crash directory.

To examine a running system, enter the `kdbx` command with the following parameters:

```
    # kdbx -k /vmunix /dev/mem
```

When you begin a debugging session, `kdbx` reads and executes the commands in the system initialization file /var/`kdbx` /system.`kdbx` rc. The initialization file contains setup commands and alias definitions. (For a list of `kdbx` aliases, see the `kdbx(8)` reference page.) You can further customize the `kdbx` environment by adding commands and aliases to one of the following initialization files:

- /var/`kdbx` /site.`kdbx` rc, which contains customized commands and alias definitions for a particular system
- ~/.`kdbx` rc, which contains customized commands and alias definitions for a specific user
- ./.`kdbx` rc, which contains customized commands and alias definitions for a specific project (this file must reside in the current working directory when `kdbx` is invoked)

## The kdbx Interface Commands

The `kdbx` interface provides the following commands:

`alias [name ] [command-string ]`

Sets or displays aliases. If you omit all arguments, alias displays all aliases. If you specify the variable name, alias displays the alias for name, if one exists. If you specify name and command-string, alias establishes name as an alias for command-string.

`context proc | user`

Sets context to the user's aliases or the extension's aliases. This command is used only by the extensions.

`coredata start_address end_address`

Dumps, in hexadecimal, the contents of the core file starting at start_ address and ending before end_address.

`ladebug` `command-string`

Passes the variable command-string to Ladebug. Specifying Ladebug is optional; if the command is not recognized by `kdbx` , it is passed automatically.

`help [-long ] [args ]`

Prints help text.

`proc [flags ] [extension ] [arguments ]`

Executes an extension and gives it control of the `kdbx` session until it quits. The variable extension specifies the named extension file and passes arguments to it as specified by the variable arguments. Valid flags are as follows:

`-debug`

Causes input to and output from the extension to be displayed on the screen.

`-pipe in_pipe out_pipe`

Used in conjunction with the dbx debugger for debugging extensions.

`-print_output`

Causes the output of the extension to be sent to the invoker of the extension without interpretation as `kdbx` commands.

`-redirect_output`

Used by extensions that execute other extensions to receive the output from the called extensions. Otherwise, the user receives the output.

`-tty`

Causes `kdbx` to communicate with the subprocess through a terminal line instead of pipes. If you specify the -pipe flag, proc ignores it.

`print string`

Displays string on the terminal. If this command is used by an extension, the terminal receives no output.

`quit`

Exits the `kdbx` interface.

`source [-x ] [file(s) ]`

Reads and interprets files as `kdbx` commands in the context of the current aliases. If the you specify the -x flag, the debugger displays commands as they are executed.

`unalias name`

Removes the alias, if any, from name.

The `kdbx` interface contains many predefined aliases, which are defined in the `kdbx` startup file /var/`kdbx` /system.`kdbx` rc.

#### Using kdbx Extensions

In addition to its commands, the `kdbx` interface provides extensions. You execute extensions using the `kdbx` command `proc` . For example, to execute the `arp` extension, you enter the following command:

```
kdbx> proc arp
```

You can create your own `kdbx` extensions.

For more information on the extensions, see the *Digital UNIX Kernel Debugging* manual.

## 22.2 Crash Dump Analysis

If your system panics or crashes, you can often find the cause by using either Ladebug or `kdbx` to analyze a crash dump.

The operating system can crash in the following ways:

- Hardware trap — A hardware problem often results in the kernel trap( ) function being invoked.
- Software panic — A software panic, resulting from a software failure, calls the kernel panic( ) function.
- Hung system — When the system hangs, you can force the creation of dump files.

If the system crashes because of a hardware fault or an unrecoverable software state, a dump function is invoked. The dump function copies the core memory into the primary default swap disk area as specified by the `/etc/fstab` file structure table and the `/sbin/swapdefault` file. At system reboot time, the information is copied into a file, called a crash dump file.

You can analyze the crash dump file to determine what caused the crash. For example, if a hardware trap occurred, you can examine variables, such as `savedefp`, the program counter (pc), and the stack pointer (sp), to help you determine why the crash occurred. If a software panic caused the crash, you can use the Ladebug debugger to examine the crash dump and the `uerf` utility to examine the error log. Using these tools, you can determine what function called the panic( ) routine.

Crash dump files, such as `vmunix.n` and `vmcore.n`, usually reside in the `/var/adm/crash` directory. The version number (*n* in `vmunix.n` and `vmcore.n`) must match for the two files.

For example, you might use the following command to examine dump files:

```
# ladebug -k vmunix.1 vmcore.1
```

### Examining the Exception Frame

When you debug your code by working with a crash dump file, you can examine the exception frame using Ladebug. The variable `savedefp` contains the location of the exception frame. (No exception frames are created when you force a system to dump.) Refer to the header file `/usr/include/machine/reg.h` to determine where registers are stored in the exception frame. The following example shows an exception frame:

```
(ladebug) print savedefp/33X

ffffffff9618d940:   0000000000000000 fffffc000046f888
ffffffff9618d950:   ffffffff86329ed0 0000000079cd612f
      .
      .
      .
ffffffff9618da30:   0000000000901402 0000000000001001
ffffffff9618da40:   0000000000002000
```

### Extracting the Character Message Buffer

You can use Ladebug to extract the preserved message buffer from a running system or dump files to display system messages logged by the kernel. For example:

```
(ladebug) **print *pmsgbuf**

struct {
       msg_magic = 405601
       msg_bufx = 1181
       msg_bufr = 1181
       msg_bufc = "Alpha boot: memory from 0x68a000 to 0x6000000
DEC OSF/1 T1.2-2   (Rev. 5); Thu Dec 03 11:20:36 EST 1992
physical memory = 94.00 megabytes.
available memory = 83.63 megabytes.
using 360 buffers containing 2.81 megabytes of memory
tc0 at nexus
scc0 at tc0 slot 7
asc0 at tc0 slot 6
rz1 at asc0 bus 0 target 1 lun 0 (DEC        RZ25        (C) DEC 0700)
rz2 at asc0 bus 0 target 2 lun 0 (DEC        RZ25        (C) DEC 0700)
rz3 at asc0 bus 0 target 3 lun 0 (DEC        RZ26        (C) DEC T384)
rz4 at asc0 bus 0 target 4 lun 0 (DEC        RRD42     (C) DEC   4.5d)
tz5 at asc0 bus 0 target 5 lun 0 (DEC        TLZ06        (C)DEC 0374)
asc1 at tc0 slot 6
fb0 at tc0 slot 8
  1280X1024
ln0: DEC LANCE Module Name: PMAD-BA
ln0 at tc0 slot 7
ln0: DEC LANCE Ethernet Interface, hardware address: 08:00:2b:2c:f6:9f
DEC3000 - M500 system
Firmware revision: 1.1
PALcode: OSF version 1.14
lvm0: configured.
lvm1: configured.
setconf: bootdevice_parser translated 'SCSI 0 6 0 0 300 0 FLAMG-IO' to 'rz3' " }
(ladebug)
```

### The crashdc Utility

The crashdc utility collects critical data from operating system crash dump files or from a running kernel. You can use the data it collects to analyze the cause of a system crash. The crashdc utility uses existing system tools and utilities to extract information from crash dumps. The information garnered from crash dump files or from the running kernel includes the hardware and software configuration, current processes, the panic string (if any), and swap information.

The crashdc utility is invoked each time the system is booted. If it finds a current crash dump, crashdc creates a data collection file with the same numerical file name extension as the crash dump.

You can also invoke crashdc manually. The syntax of the command for invoking the data collection script is as follows:

```
/bin/crashdc  vmunix. n /vmcore. n
```

See the *Digital UNIX Kernel Debugging* manual for an example of the output from the crashdc command.

### Managing Crash Dump File Creation

To ensure that you are able to analyze crash dump files following a system crash, you must understand the crash dump file creation process. This process requires that you reserve space on the system for crash dump files. The amount of space you save depends upon your system configuration and the type of crash dump file you want the system to create.

### Saving Dumps to a File System

When the system reboots, it attempts to save a crash dump from the crash dump partition to a file system. The savecore utility (/sbin/savecore), which is invoked during system startup before the dump partition is accessed, checks to see if the system crashed or was rebooted. If the system crashed within the last three days, the savecore utility performs the following tasks as the system reboots:

- Checks to see if a dump has been made within the last three days and that there is enough space to save it.
- Saves the dump file and kernel image into a specified directory. The default files for the kernel image and the dump file are vmunix.n and vmcore.n, respectively.

  The variable n gives the number of the crash. The number of the crash is recorded in the bounds file. After the first crash, the bounds file is created in the crash dump directory and the value one is stored in it. That value is incremented for each succeeding crash.
- Logs a reboot message using the facility LOG_CRIT, which logs critical conditions. For more information, refer to the syslog(3) reference page.
- Logs the panic string in both the ASCII and binary error log files, if the system crashed as a result of a panic.
- Attempts to save the kernel syslog message buffer from the dump files. The msgbuf.err entry in /etc/syslog.conf file specifies the file name and location for the msgbuf dump file. The default /etc/syslog.conf file specification is as follows:

  ```
  msgbuf.err                    /var/adm/crash/msgbuf.savecore
  ```

  If the msgbuf.err entry is not specified in the /etc/syslog.conf file, the msgbuf dump file is not saved. The msgbuf dump file cannot be forwarded to any system.

When the syslogd daemon is later initialized, it checks for the msgbuf dump file. If a msgbuf dump file is found, syslogd processes the file and then deletes it.

- Creates the file binlogdumpfile.n in the /var/adm/crash directory. The variable n is determined by the value of the bounds file.

You can modify the system default for the location of dump files by using the rcmgr command to specify another directory path for the /sbin/savecore utility:

```
# /usr/sbin/rcmgr set SAVECORE_DIR </newpath>
```

The /sbin/init.d/savecore script invokes the /sbin/savecore utility.

### Crash Dump Files

Crash dump files are either partial (the default) or full. The following sections describe each type of file and explains allocating the proper amount of space in the crash dump partition and file system.

### Partial Crash Dump Files

Unlike full crash dumps, the size of a partial crash dump file is proportional to the amount of system activity at the time of the crash: the higher the level of system activity and the larger the amount of memory in use at the time of a crash, the larger the partial crash dump files will be. For example, when a system with 96 megabytes (MB) of memory crashes, it creates a vmcore.n file with 10 to 96 MB of memory (depending upon system activity) and a vmunix.n file with approximately six MB of memory.

**Note**

If you compress a core dump file from a partial crash dump, you must use care in decompressing it. Using the uncompress command with no options results in a core file equal to the size of memory. To ensure that the decompressed core file remains at its partial dump size, you need to use the uncompress command with the -c option and the dd command with the conv=sparse option. For example, to decompress a core file named vmunix.0.Z, enter the following command:

```
# uncompress -c vmcore.0.Z | dd of=vmcore.0 conv=sparse
262144+0 records in
262144+0 records out
```

### Full Crash Dump Files

Full crash dump files can be very large because the vmunix.n file is a copy of the running kernel and the size of the vmcore.n file is slightly larger than the amount of physical memory on the system that crashed. For example, when a system with 96 MB of memory crashes, it creates a vmcore.n file with approximately 96 MB of memory and a vmunix.n file with approximately six MB of memory.

### Selecting a Crash Dump Type

The default is to use partial crash dumps. If you want to use full dumps, you can modify the default behavior in the following ways:

- By specifying the d flag to the boot_osflags console environment variable.
- By modifying the kernel's partial_dump variable to 0 using the Ladebug debugger (discussed in Chapter 8) as follows:

  ```
  (ladebug) a partial_dump = 0
  ```

  A partial_dump value of 1 indicates that partial dumps are to be generated.

### Determining Crash Dump Partition Size

If you intend to save full crash dumps, you need to reserve disk space equal to the size of memory, plus one additional block for the dump header. For example, if your system has 128 MB of memory, you need a crash dump partition of at least 128 MB, plus one block (512 bytes).

If you intend to save partial crash dumps, the size of the disk partition may vary, depending upon system activity. For example, for a system with 128 MB of memory, if peak system activity is low (never using more than 60 MB of memory), the size of the crash dump partition can be 60 MB. If peak system activity is high (using all of memory), 128 MB of disk space is needed.

If full dumps are turned on and there is not enough disk space to create dump files for a full dump, partial dumps are automatically invoked.

### Determining File System Space for Saving Crash Dumps

The size of the file system needed for saving crash dumps depends on the size and the number of crash dumps you want to retain. A general guideline is to reserve, at a minimum, the size of your crash dump partition, plus 10 MB. If necessary, you can increase this amount later.

If your system cannot save a crash dump due to insufficient disk space, it returns to single user mode. This return to single user mode prevents system swapping from corrupting the dump file. Space can then be made available in the crash dump directory, or the changed directory, before continuing to multiuser mode. You can override this option using the following command:

```
# /usr/sbin/rcmgr set SAVECORE_FLAGS M
```

This command causes the system to always boot to multiuser mode even if it cannot save a dump.

### Procedures for Creating Dumps of a Hung System

If necessary, you can force the system to create dump files when the system hangs. The method for forcing crash dumps varies according to the hardware platform. The methods are described in the *Digital UNIX Kernel Debugging* manual.

### Guidelines for Examining Crash Dump Files

In examining crash dump files, there is no one way to determine the cause of a system crash. However, the following guidelines should assist you in identifying the events that led to the crash:

- Gather some facts about the system (for example, operating system type, version number, revision level, and hardware configuration).
- Look at the panic string, if one exists. This string is contained in the preserved message buffer, `pmsgbuf,` and in the `panicstr` global variable.
- Locate the thread executing at the time of the crash. (Use the `where` command.) Most likely, this thread will contain the events that led to the panic.
- Determine whether you can fix the problem. If the system crashed because of lack of resources (for example, swap space), you can probably eliminate the problem by adding more of that resource.
- If the problem is with the software, you may need to file a report with your local Digital Customer Support Center.

For more information, and for examples, see the *Digital UNIX Kernel Debugging* manual. This manual contains detailed information on the following topics related to crash dump analysis:

- The `crashdc` utility
- Managing crash dump file creation
- Saving dumps to a file system
- Selecting full or partial crash dump files
- Determining crash dump partition size and file system space

- Procedures (according to hardware platform) for creating dumps of a hung system

**Note**

Crash dump analysis is possible only with local, not remote, kernel debugging.

---

## 22.3 Remote Kernel Debugging with the kdebug Debugger

For remote kernel debugging, Ladebug is used in conjunction with the `kdebug` debugger,† which is a tool for executing, testing, and debugging test kernels. The `kdebug` code runs inside the kernel to be debugged on a test system, while Ladebug runs on a remote system and communicates with `kdebug` over a serial line or a gateway system.

You use Ladebug commands to start and stop kernel execution, examine variable and register values, and perform other debugging tasks, just as you would when debugging user space programs. The `kdebug` debugger, not Ladebug, performs the actual reads and writes to registers, memory, and the image itself (for example, when breakpoints are set).

### Connections Needed

The kernel code to be debugged runs on a test system. Ladebug runs on a remote build system and communicates with the kernel code over a serial communication line or through a gateway system.

You use a gateway system when you cannot physically connect the test and build systems. The build system is connected to the gateway system by a network line. The gateway system is connected to the test system by a serial communication line.

The following diagram shows the physical connection of the test and build systems (with no gateway):

```
  Build system         Serial line        Test system
(with Ladebug) <----------------------> (kernel code here)
```

† Used alone, `kdebug` has its own syntax and commands, and allows local nonsymbolic debugging of a running kernel across a serial line. See the `kdebug(8)` manpage for information about `kdebug` local kernel debugging.

The following diagram shows the connections when you use a gateway system:

```
  Build system       Network    Gateway      Serial line         Test system
(with Ladebug) <-----------> system <---------------------> (kernel code here)
                                with
                                kdebug
                                daemon
```

### System Requirements

The test, build, and (if used) gateway systems must meet the following requirements for `kdebug`:

- Test system

  Must be running Version 2.0 or higher of the Digital UNIX operating system, must have the Kernel Debugging Tools subset loaded, and must have the Kernel Breakpoint Debugger kernel option configured.

- Build system

  Must be running Version 3.2 or higher of the Digital UNIX operating system. Also, this system must contain a copy of the kernel code you are testing and, preferably, the source used to build that kernel code.

- Gateway system

  Must be running Version 2.0 or higher of the Digital UNIX operating system, and must have the Kernel Debugging Tools subset loaded.

### Getting Ready to Use the kdebug Debugger

To use the `kdebug` debugger, first do the following:

1. Attach the test system and the build system or test system and gateway system. See your hardware documentation for information about connecting systems to serial lines and networks.
2. Configure the kernel to be debugged with the configuration file option OPTIONS KDEBUG. If you are debugging the installed kernel, you can do this by selecting KERNEL BREAKPOINT DEBUGGING from the kernel options menu.
3. Recompile kernel files, if necessary. By default, the kernel is compiled with only partial debugging information, occasionally causing Ladebug to display erroneous arguments or mismatched source lines. To correct this, recompile selected source files specifying the CDEBUGOPTS=-g argument.
4. Copy the kernel to be tested to `/vmunix` on the test system. Retain an exact copy of this image on the build system.

5. Install the Product Authorization Key (PAK) for the Developer's kit (OSF-DEV), if it is not already installed. For information about installing PAKs, see the Installation Guide.

6. Determine the debugger variable settings or command-line options you will use, as follows:

   **Debugger variables:** On the build system, add the following lines to your `.dbxinit` file if you need to override the default values (and you choose not to use the corresponding options, described below). Alternatively, you can use these lines within the debugger session, at the `(ladebug)` prompt:

   ```
   set $kdebug_host="gateway_system"
   set $kdebug_line="serial_line"
   set $kdebug_dbgtty="tty"
   ```

   `$kdebug_host` specifies the node or address of the gateway system. By default, `$kdebug_host` is set to `localhost,` for when a gateway system is not used.

   `$kdebug_line` specifies the serial line to use as defined in the `/etc/remote` file of the build system (or the gateway system, if one is being used). By default, `$kdebug_line` is set to `kdebug` .

   `$kdebug_dbgtty` sets the terminal on the gateway system to display the communication between the build and test systems, which is useful in debugging your setup. To determine the terminal name to supply to the `$kdebug_dbgtty` variable, enter the `tty command` in the desired window on the gateway system. By default, `$kdebug_dbgtty` is null.

   **Options:** Instead of using debugger variables, you can specify any of the following options on the `ladebug` command line:

   - The `-rn` option specifies the node or address of the gateway system, and can be used instead of `$kdebug_host.`
   - The `-line` option specifies the serial line, and can be used instead of `$kdebug_line.`
   - The `-tty` option specifies the terminal name, and can be used instead of `$kdebug_dbgtty.`

   The above three options require the `-remote` option or its alternative, the `-rp kdebug` option.

   The variables you set in your `.dbxinit` file will override any options you use on the `ladebug` command line. In your debugging session, you can still override the `.dbxinit` variable settings by using the `set` command at the `(ladebug)` prompt, prior to issuing the `run` command.

7. If you are debugging on an SMP system, set the `lockmode` system attribute to four, as shown:

   ```
   #  sysconfig -r lockmode = 4
   ```

   Setting this system attribute makes debugging on an SMP system easier.

### Invoking the Debugger

When the setup is complete, start up the debugger as follows:

1. Invoke the Ladebug debugger on the build system, supplying the pathname of the copy of the test kernel that resides on the build system. Set a breakpoint and start running Ladebug as follows (assuming that `vmunix` resides in the `/usr/test` directory):

   ```
   # ladebug -remote /usr/test/vmunix

       .
       .
       .

   (ladebug)  stop in hard_clock
   [2] stop in hard_clock
   (ladebug)  run
   ```

   Because Ctrl/C cannot be used as an interrupt, you should set at least one breakpoint if you wish the debugger to gain control of kernel execution. You can set a breakpoint anytime after the execution of the `kdebug_bootstrap( )` routine. Setting a breakpoint prior to the execution of this routine can result in unpredictable behavior.

   **Note**

   Pressing Ctrl/C causes the remote debugger to exit, not interrupt as it does during local debugging.

2. Halt the test system and, at the console prompt, set the `boot_osflags` console variable to contain the `k` option, and then boot the system. For example:

   ```
   >>>  set boot_osflags k
   >>>  boot
   ```

   Alternatively, you can enter:

   ```
   >>>  boot -A k
   ```

Once you boot the kernel, it begins executing. The Ladebug debugger halts execution at the breakpoint you specified, and you can begin issuing Ladebug debugging commands. All Ladebug commands are available, except `kps`, `attach`, and `detach`. See Part V, Command Reference for information on Ladebug debugging commands.)

### Breakpoint Behavior on SMP Systems

If you set breakpoints in code that is executed on an SMP system, the breakpoints are handled serially. When a breakpoint is encountered on a particular CPU, the state of all the other processors in the system is saved and those processors spin, similarly to how execution stops when a simple lock is obtained on a particular CPU.

When the breakpoint is dismissed (for example, because you entered a `step` or `cont` command to the debugger), processing resumes on all processors.

### Troubleshooting Tips

If you have completed the `kdebug` setup and it fails to work, refer to the following list for help:

- Be sure the serial line is attached properly. Use the `tip` command to test the connection: Log onto the build system (or the gateway system if one is being used) as root and enter the following command:

  ```
  # tip kdebug
  ```

  If the command does not return the message "connected," another process, such as a print daemon, might be using the serial line port that you have dedicated to the `kdebug` debugger. To remedy this condition, do the following:

  – Check the `/etc/inittab` file to see if any processes are using that line. If so, disable these lines until you finish with the `kdebug` session. See the `inittab(4)` reference page for information on disabling lines.

  – Use the `ps` command to see if any processes are using the line. For example, if you are using the `/dev/tty00` serial port for your `kdebug` session, check for other processes using the serial line as follows:

  ```
  # ps agxt00
  ```

  If a process is using `tty00,` kill that process.

  – Determine whether any unused `kdebugd` gateway daemons are running:

  ```
  # ps agx | grep kdebugd
  ```

  If one is running, kill the process.

- If the test system boots to single user or beyond, then kdebug has not been configured into the kernel as specified in the section Getting Ready to Use the kdebug Debugger. Ensure that the boot_osflags console environment variable specifies the k flag and try booting the system again:

  ```
  >>>  set boot_osflags k
  >>>  boot
  ```

- Be sure you defined the Ladebug variables in your .dbxinit file correctly, or specify them correctly on the command line.

  Determine which pseudoterminal line you ran tip from by issuing the /usr/bin/tty command. For example:

  ```
  # /usr/bin/tty
  /dev/ttyp2
  ```

  The example shows that you are using pseudoterminal /dev/ttyp2. Edit your $HOME/.dbxinit file on the build system as follows:

  1. Set the $kdebug_dbgtty variable to /dev/ttyp2 with this command:

     ```
     set $kdebug_dbgtty="/dev/ttyp2"
     ```

  2. Set the variable $kdebug_host to the host name of the system from which you entered the tip command. For example, if the host name is decosf, the entry in the .dbxinit file should be:

     ```
     set $kdebug_host="decosf"
     ```

  3. Remove any settings of the $kdebug_line variable:

     ```
     set $kdebug_line=""
     ```

- Start Ladebug on the build system. You should see informational messages on the pseudoterminal line, /dev/ttyp2, which kdebug is starting.
- If you are using a gateway system, ensure that the inetd daemon is running on the gateway system. Also, check the TCP/IP connection between the build and gateway system using one of the following commands: rlogin, rsh, or rcp.

### 22.3.1 Analyzing a Crash Dump

When the operating system crashes, the dump function copies core memory into swap partitions. Then at system reboot time, this copy of core memory is copied into the crash dump file, which you can analyze.

When the operating system hangs, you may need to force a crash dump.

## 22.4 Debugging Loadable Drivers

The procedure for debugging loadable drivers depends on whether you are doing local or remote kernel debugging.

### Loadable Drivers and Local Kernel Debugging

For local kernel debugging, any loadable drivers already present in the kernel are automatically loaded into the debugger once when the debugger is started. Since the kernel is running, additional drivers can be loaded at any time. If you wish to obtain the most current list of loaded drivers, you can manually load any new driver information with the following command:

```
(ladebug) readsharedobj /driver-directory/driver.mod
```

The list of drivers currently known to ladebug can be displayed as follows:

```
(ladebug) listobj
ObjectName                        Start Addr            Size        Symbols
                                                       (bytes)       Loaded
-----------------------------------------------------------------------------
/vmunix                    0xfffffc0000230000       2442992          Yes
/var/subsys/dna_netman.mod
                           0xffffffff90ce0000         49152          Yes
/var/subsys/dna_dli.mod
                           0xffffffff90cf0000         57344          Yes
/var/subsys/dna_base.mod
                           0xffffffff90d04000        393216          Yes
/var/subsys/dna_xti.mod
                           0xffffffff90b0a000          8192          Yes
```

### Loadable Drivers and Remote Kernel Debugging

For remote kernel debugging, you can debug loadable drivers as follows:

1. On your remote machine, create a directory called (for example)

   `/usr/opt/TMU100`

   Put both the source file and the loadable driver's `.mod` file into this directory. Assuming a loadable driver called `tmux`, get the `tmux.mod` file, and make sure you have permission to read, write, and execute the file.

2. Configure your system as follows:

   Create a /usr/opt/TMU100/stanza.loadable file:

```
tmux:
   Subsystem_Description = TMUX device driver
   Module_Config_Name = tmux
   Module_Config1 = controller tmux0 at *
   Module_Type = Dynamic
   Module_Path = /usr/opt/TMU100/tmux.mod
   Device_Dir = /dev/streams
   Device_Char_Major = Any
   Device_Char_Minor = 0
   Device_Char_Files = tmux0
```

Run the sysconfigdb and sysconfig utilities:

```
sysconfigdb -a -f /usr/opt/TMU100/stanza.loadable tmux
sysconfigdb -s
cp /usr/opt/TMU100/tmux.mod /subsys/tmux.mod
cd /subsys
ln -s device.mth tmux.mth
```

3. Ensure that there is a copy of the driver tmux.mod residing in the *same directory* on the local machine, for example, /subsys/tmux.mod.

4. Start up Ladebug with the "-remote" option, and set a breakpoint in the routine `subsys_preconfigure`. Issue the run command:

   ```
   (ladebug) stop in subsys_preconfigure
   [#1: stop in void subsys_preconfigure(caddr_t) ]
   (ladebug) run
   ```

   `subsys_preconfigure` is provided to assist developers in debugging configuration routines. It is needed because the debugger is not notified when a subsystem is loaded, and a user-defined breakpoint cannot be set until the load has occurred. `subsys_preconfigure` is guaranteed to be called following a subsystem load but prior to a configuration.

   If your kernel has been built with symbolic information, once stopped in `subsys_preconfigure` you can examine the variable `subsys` to see if this event corresponds to your driver being loaded:

   ```
   (ladebug) print subsys
   0xfffffc0000620cd4="generic"
   ```

   Depending on the number of subsystems being automatically loaded, you may stop in `subsys_preconfigure` many times. If your driver is being loaded last, or if you are manually loading your driver (see item #5, below), you may prefer to employ a more useful breakpoint that gets you closer to your desired stopping point, one that brings you closer to the `subsys_preconfigure` call that takes place immediately prior to your driver being loaded.

5. If you are manually loading your driver, you will need to run the remote kernel to the point where you have the hash (#) single-user prompt. Once there, configure the driver as follows:

   ```
   # sysconfig -c tmux
   ```

6. Once your driver has been loaded into the kernel, the debugger will stop at your `subsys_preconfigure` breakpoint. At this time you can issue the following commands:

   ```
   (ladebug) print subsys
   0xfffffc0000620cd4="tmux"
   (ladebug) readsharedobj /subsys/tmux.mod
   ```

   The `readsharedobj` command causes ladebug to retrieve the symbolic information associated with your driver. A lot of information is transferred over a serial line during this operation, so expect it to take several seconds to complete.

7. Now Ladebug knows about your driver, so you can proceed with symbolic debugging as you would with any other program. For example:

```
(ladebug) stop in tmux_configure
[#2: stop in int tmux_configure(cfg_op_t, caddr_t, ulong, caddr_t, ulong) ]
(ladebug) cont
[2] stopped at [int tmux_configure(cfg_op_t, caddr_t, ulong, caddr_t,
  ulong):933 0xffffffff89a14028]
    933     sa.sa_version      = OSF_STREAMS_11;
(ladebug) next
stopped at [int tmux_configure(cfg_op_t, caddr_t, ulong, caddr_t,
  ulong):934 0xffffffff89a14034]
    934     sa.sa_flags        = STR_IS_DEVICE | STR_SYSV4_OPEN;
(ladebug) print sa.sa_version
84107547
```

**Note**

If you use sysconfig to unload and then subsequently reload a driver in a kernel actively being debugged by ladebug, any breakpoints previously present in that driver will be lost. To reestablish those breakpoints in the newly loaded subsystem, issue the following ladebug commands prior to continuing:

```
(ladebug) disable *
(ladebug) enable *
(ladebug) cont
```

# Part V

## Command Reference

This section explains the individual Ladebug commands.

## ladebug—dxladebug

### Name

ladebug — Invokes the command interface of the debugger.

dxladebug — Invokes the graphical user interface of the debugger.

### Syntax (command-line)

**ladebug** [**-I** *directory*] [**-c** *file*] [**-prompt** *string*] [**-nosharedobjs**] [**-pid** *process_id*] [**-rn** *node_or_address* [*,udp_port*]] [**-rfn** *remote_executable_file*] [**-ru** *remote_username*] [**-rinsist**] [**-k**] [**-line** *serial_line*] [**-remote**] [**-rp** *remote_debug_protocol*] [**-tty** *terminal_device*] [*executable_file* [*core_file*]]

### Syntax (window interface)

**dxladebug** [**-iow** ] [**-k** | **-kernel**] [**-P** *program_arguments*]

### Description

Ladebug is a symbolic source code debugger that debugs programs compiled by the DEC C, ACC, DEC C++, DEC Ada, DEC COBOL, DEC Fortran 90 and DEC Fortran 77 compilers. For full source-level debugging, compile the source code with the compiler option that includes the symbol table information in the compiled executable file.

### Options and Parameters

The following table describes the options and parameters for the Digital Ladebug debugger command line debugger:

| | |
|---|---|
| **-I** | Specifies the directory containing the source code for the target program. Use multiple `-I` options to specify more than one directory. The debugger searches directories in the order in which they were specified on the command line. |

| | |
|---|---|
| **-c** | Specifies an initialization command file. The default initialization file is `.dbxinit.` By default, Ladebug searches for this file during startup, first in the current directory; if it is not there, Ladebug searches your home directory for the file. |
| **-prompt** | Specifies a debugger prompt. The default debugger prompt is `(ladebug)` . If the prompt argument contains spaces or special characters, enclose the argument in quotes (" "). |
| **-nosharedobjs** | Prevents the reading of symbol table information for any shared objects that are loaded when the program executes. Later in the debug session, the user can enter the `readsharedobj` command to read in the symbol table information for a specified object. |
| **-pid** | Specifies the process ID of the process to be debugged. Cannot be used with any remote or kernel debugging flags. |
| **-rn** | Specifies the host name or Internet address of the machine on which the remote debugger server is running. Optionally, specifies the UDP port on which to connect the server. Depending on your shell, it may be necessary to use quotation marks to avoid shell command-line interpretation on the local system. With remote kernel debugging, the default is `localhost.` |
| **-rfn** | Specifies the file name (or other identifier) of the executable file to be loaded on the remote system. This option defaults to the local executable file name and is passed uninterpreted to the remote system. When using this option, specify the remote executable file using the syntax of the remote file system; use quotation marks to avoid shell command-line interpretation on the local system. Use only with `-rn;` do not combine with `-pid.` |
| **-ru** | Specifies the user name to be used on the remote target machine. If the `-ru` option is not specified, the default is the local user name. |

| | |
|---|---|
| **-rinsist** | Connects to a running remote process using the insist protocol message instead of the connect protocol message. This option functions as a request to the server to connect to the client, even if another client is already connected. Use only with `-rn` and `-pid`. |
| **-k** or **-kernel** | Enables local kernel debugging. |
| **-line** | For use with remote kernel debugging; specifies the serial line. If used, `-remote` or `-rp kdebug` must also be used. The default is `kdebug`. |
| **-remote** | Enables remote kernel debugging; for use with the `kdebug` kernel debugger. |
| **-rp** | Specifies the remote debug protocol, either `ladebug_preemptive` or `kdebug`. `-rp kdebug` enables remote kernel debugging. |
| **-tty** | For use with remote kernel debugging; specifies the terminal device. If used, `-remote` or `-rp kdebug` must also be used. |
| *executable_file* | Specifies the program executable file. If the file is not in the current directory, specify the pathname. |
| *core_file* | Specifies the core file. If the core file is not in the current directory, specify the pathname. |

The following table describes the options and parameters for the `dxladebug` command:

| | |
|---|---|
| **-iow** | Invokes the window interface to the debugger and displays a separate user program I/O window. |
| **-k** | Enables local kernel debugging. |
| **-P** | Specifies the arguments used by the program you are debugging. |

## General Instructions for Entering Ladebug Commands

Enter Ladebug commands at the debugger prompt `(ladebug)`. You can enter more than one command on the same line by separating each command with a semicolon (;). Commands are executed in the same order in which they are entered in the command line.

Continue command input to another line by entering a backslash (\) at the end of each line. The maximum command-line length is 255 characters.

In debugger commands, the keywords in the following list must be surrounded by parentheses in expressions that use them as variables or type names:

```
thread
in
at
state
if
policy
priority
with
```

For example, the commands `print (thread)` and `print (thread* )t` are valid. The parentheses enable the debugger to distinguish an identifier from a keyword.

## Command Summary

Table REF–1 lists and describes the Ladebug commands, grouped in functionally related sets:

**Table REF–1 Functionally Related Sets of Ladebug Commands**

| | |
|---|---|
| `#` | Enter a comment. |
| `alias`, `unalias` | Define, view, or delete a debugger command alias. |
| `catch`, `ignore` | Examine and change the list of operating system signals trapped by the debugger. |
| `history`, `!` | Repeat and list previously used commands. |
| `file` | Set the file scope. |
| `list`, `use`, `unuse`, `/`, `?` | Select and view program source code. |
| `listobj` | List all loaded objects, including the main image and shared libraries. |
| `readsharedobj`, `delsharedobj` | Read or delete symbol table information for the specified shared object. |

(continued on next page)

**Table REF–1 (Cont.) Functionally Related Sets of Ladebug Commands**

| | |
|---|---|
| `load`, `unload` | Load an image file and, optionally, a core file; remove all related symbol table information associated with the process being debugged. |
| `stopi`, `tracei`, `wheni`, `nexti`, `stepi`, `printregs` | Machine-code level commands. |
| `print`, `printf`, `dump`, `assign`, `whatis`, `which`, `whereis` | Examine program expressions and change their values. |
| `quit`, `help` | Exit and get help about the debugger. |
| `run`, `rerun`, `cont`, `next`, `step`, `return`, `call`, `goto`, `kill` | Execute or terminate a program under debugger control. |
| `sh` | Execute a Bourne shell command. |
| `set`, `unset` | Define, view, or delete a debugger variable. |
| `source`, `playback input`, `record input`, `record output`, `record io` | Read in or save a file containing debugger input data or output data. |
| `stop`, `when`, `trace`, `status`, `delete`, `enable`, `disable` | Set, list, delete, enable, and disable program breakpoints and tracepoints. |
| `stop thread`, `when thread`, `wheni thread`, `trace thread`, `tracei thread` | Control execution of one or more threads in a process. |
| `thread`, `show thread`, `show condition`, `show mutex` | View information available from the debugger about threads in your application. |
| `process`, `show process` | Display information for current process(es) and change the current process. |
| `where`, `where thread`, `up`, `down`, `func` | Examine the stack trace and change the function scope. |
| `setenv`, `export`, `printenv`, `unsetenv` | Manipulate subsequent debuggee environments with environment variables . |
| `class` | Change or display the class scope. |

(continued on next page)

**Table REF–1 (Cont.) Functionally Related Sets of Ladebug Commands**

| | |
|---|---|
| `kps` | Used to list all system processes; for local kernel debugging only. |
| `pop` | Removes execution frame(s) from call stack. |

Table REF–2 gives a quick summary of the function of each individual Ladebug command:

**Table REF–2 Ladebug Commands: Individual Summary**

| Command | Directly Manipulable in Window Interface† | Function |
|---|---|---|
| `#` | | Causes debugger to ignore all input until end of marked line. |
| `!` | | Without argument, repeats previous command line (use `!!` or `!-1`). With argument, repeats specified command line. |
| `/` | | Invokes forward search in source program. |
| `?` | | Invokes backward search in source program. |
| `alias` | | Without argument, displays all aliases and their definitions. With argument, displays definition for specified alias. |
| `assign` | Yes | Changes value of variable, memory address, or expression. |
| `attach` | Yes | Connects to a running process. |
| `call` | | Executes specified function. |

†"Yes" denotes commands with an exact mouse action equivalent in the window interface. To use any other command in the window interface, you must type the command at the command-entry prompt in the communications pane.

(continued on next page)

**Table REF–2 (Cont.) Ladebug Commands: Individual Summary**

| Command | Directly Manipulable in Window Interface† | Function |
|---|---|---|
| `catch` | | Without argument, shows which operating system signals debugger currently traps.<br>With argument, traps specified operating system signal. |
| `catch unaligned` | | Traps program immediately after unaligned data access occurs. |
| `class` | | For C++ only.<br>Without argument, displays current class scope.<br>With argument, changes class scope. |
| `cont` | Yes | Resumes program execution. |
| `delete` | Yes | Removes specified breakpoint or tracepoint. |
| `delsharedobj` | | Removes symbol table information for specified shared object. |
| `detach` | Yes | Detaches from a running process specified from process ID list. |
| `disable` | Yes | Disables specified breakpoint or tracepoint. |
| `down` | Yes | Without argument, changes function one level down stack.<br>With argument, changes function specified number of levels down stack. |
| `dump` | Yes | Without argument, lists parameters and local variables in current function.<br>With argument, lists parameters and local variables in specified active function. |

†"Yes" denotes commands with an exact mouse action equivalent in the window interface. To use any other command in the window interface, you must type the command at the command-entry prompt in the communications pane.

(continued on next page)

**Table REF–2 (Cont.) Ladebug Commands: Individual Summary**

| Command | Directly Manipulable in Window Interface† | Function |
|---|---|---|
| `enable` | Yes | Enables specified breakpoint or tracepoint. |
| `export` | | Synonym for `setenv`. |
| `file` | Yes | Without argument, displays name of current file scope.<br>With argument, changes specified file scope. |
| `func` | Yes | Without argument, displays current function scope.<br>With argument, changes function scope to function currently active on stack. |
| `goto` | Yes | Branches to specified line in function where execution is suspended. |
| `help` | Yes | Without argument, displays list of debugger commands.<br>With argument, displays description of specified command. |
| `history` | | Without argument, displays default number of previously executed commands. (20).<br>With argument, displays specified number of previously executed commands. |
| `ignore` | | With argument, shows which operating system signals debugger currently ignores.<br>With argument, ignores specified operating system signal. |

†"Yes" denotes commands with an exact mouse action equivalent in the window interface. To use any other command in the window interface, you must type the command at the command-entry prompt in the communications pane.

(continued on next page)

**Table REF–2 (Cont.) Ladebug Commands: Individual Summary**

| Command | Directly Manipulable in Window Interface† | Function |
|---|---|---|
| `ignore unaligned` | | Instructs debugger not to stop when unaligned access occurs (default). |
| `kill` | | Terminates program process while leaving debugger running. |
| `kps` | | Lists all system processes (valid for local kernel debugging only). |
| `list` | Yes | Depending on argument, displays source code lines beginning with line corresponding to any of following:<br>• Line designating position of program counter<br>• Last line listed, if multiple list commands are entered<br>• Line number specified as first argument |
| `listobj` | Yes | Lists all loaded objects, including main image and shared libraries. |
| `load` | Yes | Loads image file or core file. |
| `next` | Yes | When next line of code to be executed contains a function call, executes function and returns to line immediately after function call. |
| `nexti` | Yes | When machine instruction contains a function call, executes function being called. |
| `patch` | | Corrects bad data or instructions in executable disk files. |

†"Yes" denotes commands with an exact mouse action equivalent in the window interface. To use any other command in the window interface, you must type the command at the command-entry prompt in the communications pane.

(continued on next page)

**Table REF–2 (Cont.) Ladebug Commands: Individual Summary**

| Command | Directly Manipulable in Window Interface† | Function |
|---|---|---|
| `playback input` | | Executes debugger commands contained within a file. |
| `pop` | | Without argument, removes removes one execution frame from the call stack.<br>With argument, removes the specified number of execution frames from the call stack. |
| `print` | Yes | Displays current value of a variable or expression visible in current context. |
| `printenv` | | Without argument, displays values of all environment variables.<br>With argument, displays value of specified environment variable. |
| `printf` | Yes | Formats and displays a complex structure. |
| `printregs` | Yes | Displays contents of all machine registers. |
| `process` | Yes | Without argument, displays current process.<br>With argument, switches to specified process. |
| `quit` | Yes | Ends debugging session and returns to operating system prompt. |
| `readsharedobj` | | Reads in symbol table information for specified shared library or loadable kernel module. |
| `record input` | | Saves all debugger commands to a file. |
| `record io` | | Saves both debugger input and debugger output to a file. |

†"Yes" denotes commands with an exact mouse action equivalent in the window interface. To use any other command in the window interface, you must type the command at the command-entry prompt in the communications pane.

(continued on next page)

**Table REF–2 (Cont.) Ladebug Commands: Individual Summary**

| Command | Directly Manipulable in Window Interface† | Function |
|---|---|---|
| `record output` | | Saves debugger output to a file. |
| `rerun` | Yes | Restarts program execution. |
| `return` | Yes | Without argument, continues execution of current function until control is returned to caller.<br>With argument, execution continues until control is returned to specified function. |
| `run` | Yes | Starts program execution. |
| `set` | | Without argument, examines definitions of all debugger variables.<br>With argument, defines specified debugger variable. |
| `setenv` | | Without argument, displays values of all environment variables.<br>With argument, sets value of specified environment variable. |
| `sh` | | Executes Bourne shell command. |
| `show condition` | | For DECthreads only.<br>Without argument, displays information about all condition variables currently available.<br>With argument, displays information about condition variables specified. |
| `show mutex` | | Lists information about currently available mutexes. |
| `show process` | Yes | Displays information for current process. |

†"Yes" denotes commands with an exact mouse action equivalent in the window interface. To use any other command in the window interface, you must type the command at the command-entry prompt in the communications pane.

(continued on next page)

**Table REF–2 (Cont.) Ladebug Commands: Individual Summary**

| Command | Directly Manipulable in Window Interface† | Function |
|---|---|---|
| `show thread` | Yes | Without argument, lists all threads known to debugger.<br>With argument, displays information about specified thread. |
| `source` | | Executes debugger commands contained within a file. |
| `status` | Yes | Lists all breakpoints and tracepoints, reference number associated with each, and whether breakpoint is disabled. |
| `step` | Yes | Steps into and executes first line of function. |
| `stepi` | Yes | Steps into and executes next machine instruction. If function call, steps into and executes first instruction in function. |
| `stop` | | Without variable argument, suspends program execution and returns to prompt.<br>With variable argument, suspends program execution when variable changes. |
| `stopi` | | Suspends program execution when specified variable value changes. |
| `thread` | Yes | Identifies or sets current thread context. |

†"Yes" denotes commands with an exact mouse action equivalent in the window interface. To use any other command in the window interface, you must type the command at the command-entry prompt in the communications pane.

(continued on next page)

**Table REF–2 (Cont.) Ladebug Commands: Individual Summary**

| Command | Directly Manipulable in Window Interface† | Function |
|---|---|---|
| `trace` | | Without argument, causes debugger to print message when each function is entered, but does not suspend program execution.<br>With argument, causes debugger to print message when specified variable value changes, but does not suspend program execution. |
| `tracei` | | Without argument, prints message but does not suspend program execution.<br>With argument, prints message when any of following occur, but does not suspend program execution:<br>• Value of specified variable changes.<br>• Specified expression evaluates to true.<br>• Both. |
| `unalias` | | Deletes specified alias. |
| `unload` | | Removes all related symbol table information that debugger associates with process being debugged, specified by either a process ID or image file. |
| `unset` | | Deletes debugger variable. |
| `unsetenv` | | Without argument, removes all environment variables.<br>With argument, removes specified environment variable. |

†"Yes" denotes commands with an exact mouse action equivalent in the window interface. To use any other command in the window interface, you must type the command at the command-entry prompt in the communications pane.

(continued on next page)

**Table REF–2 (Cont.) Ladebug Commands: Individual Summary**

| Command | Directly Manipulable in Window Interface† | Function |
|---|---|---|
| `unuse` | | Without argument, sets search list to default, home directory, current directory, or directory containing executable file.<br>With argument, removes specified directory from search list. |
| `up` | Yes | Without argument, changes function scope one level up stack.<br>With argument, changes function scope specified number of levels up function scope. |
| `use` | | Without argument, lists directories searched for source-code files.<br>With argument, makes source-code files in specified directory available. |
| `whatis` | Yes | Prints type of specified expression. |
| `when` | Yes | Executes specified command. |
| `wheni` | Yes | Executes specified command without suspending program execution. |
| `where` | Yes | Displays stack trace of currently active functions for current thread. |
| `whereis` | | Shows all declarations of expression. |
| `which` | | Shows fully qualified scope information when instance of specified expression occurs within current scope. |

†"Yes" denotes commands with an exact mouse action equivalent in the window interface. To use any other command in the window interface, you must type the command at the command-entry prompt in the communications pane.

## Command Descriptions

startaddress, endaddress / mode
startaddress / count mode

You can display stored values as character strings, machine instructions, or decimal, octal, hexadecimal, or real numbers. Specify the address and the number of words or bytes (count) information in hexadecimal, octal, or decimal. The display mode must be specified along with the address range.

The modes are:

| | |
|---|---|
| `d` | Print a short word in decimal |
| `u` | Print a short word in unsigned decimal |
| `D` | Print a long word in decimal |
| `U` | Print a long word in unsigned decimal |
| `o` | Print a short word in octal |
| `O` | Print a long word in octal |
| `x` | Print a short word in hexadecimal |
| `X` | Print a long word in hexadecimal |
| `b` | Print a byte in hexadecimal |
| `c` | Print a byte as a character |
| `s` | Print a string of characters (a C-style string that ends in null) |
| `f` | Print a single-precision real number |
| `g` | Print a double-precision real number |
| `i` | Disassemble machine instructions |

Note that you must enter a space between "count" and "mode" if the language of the program being debugged is COBOL.

# comment

When the debugger encounters the # command, it ignores all inputs until the end of the line.

**!!**
**!** integer
**!-** integer
**!** text

To repeat the last command line, enter two exclamation points or press the Return key. You can also enter `!-1`.

To repeat a command line entered during the current debugging session, enter an exclamation point followed by the integer associated with the command line. (Use the `history` command to see a list of commands used.) For example, to repeat the seventh command used in the current debugging session, enter `!7`. Enter `!-3` to repeat the third-to-last command.

To repeat the most recent command starting with a string, use the last form of the command. For example, to repeat a command that started with `bp`, enter `!bp`.

**/** string
**?** string

Use the string search commands `/` and `?` to locate text in the source code. The `/` character invokes a forward search; the `?` character invokes a backward search. Enter `/` or `?` without an argument to find the next location of the previously specified text.

The search begins from the current position of the program counter. If no program counter exists for the current source file, the search begins after the last source line displayed by the debugger.

**alias** [aliasname]
**alias** aliasname [(argument)] "[aliasname] string"
**unalias** aliasname

Enter the `alias` command without an argument to display all aliases and their definitions. Specify an alias name to display the definition for that alias.

Use the second form to define a new alias or to redefine an existing alias. The definition can contain:

- The name of another alias, if the nested alias is the first identifier in the definition; for example, you can define a nested alias and invoke the alias as follows:

  ```
  (ladebug) alias begin "stop in main; run"
  (ladebug) alias pv(x) "begin; print(x)"
  (ladebug) pv(i)
  ```

- A string in quotation marks, specified with backslashes before the quotation marks, as in the following two examples:

```
    (ladebug) alias x "set $lang=\"C++\" "
    (ladebug) alias x
    x       set $lang="C++"
    (ladebug)

    (ladebug) alias ada "ignore sigalrm; ignore sigfpe; set $lang=\"Ada\";"
    (ladebug) alias ada
    ada     ignore sigalrm; ignore sigfpe; set $lang="Ada";
    (ladebug)
```

Note that in the first example there is a space between the last two quotation marks. In the second example there is a semicolon. The two quotation marks cannot be together; they must be separated by a space or character(s).

Invoke the alias by entering the alias name, including an argument if the alias definition specified an argument.

Use the `unalias` command to delete an alias.

Alias commands can contain no more than 56 arguments.

The following predefined aliases are included with the debugger:

| | |
|---|---|
| `S` | `next` |
| `Si` | `nexti` |
| `W` | `list $curline - 10:20` |
| `a` | `assign` |
| `b` | `stop at` |
| `bp` | `stop in` |
| `c` | `cont` |
| `d` | `delete` |
| `e` | `file` |
| `f` | `func` |
| `g` | `goto` |
| `h` | `history` |
| `j` | `status` |
| `l` | `list` |
| `li` | `($curpc)/10 i; set $curpc = $curpc + 40` |
| `n` | `next` |
| `ni` | `nexti` |
| `p` | `print` |
| `pd` | `printf "%ld",` |
| `pi` | `playback input` |
| `po` | `printf "0%o",` |
| `pr` | `printregs` |
| `ps` | `printf "%s",` |
| `px` | `printf "0x%lx",` |
| `q` | `quit` |
| `r` | `rerun` |
| `ri` | `record input` |
| `ro` | `record output` |
| `s` | `step` |

```
si      stepi
t       where
tlist   show thread (only defined during kernel debugging)
tset    thread (only defined during kernel debugging)
u       list $curline - 9:10
w       list $curline - 5:10
wi      ($curpc - 20)/10 i
```

**assign** target = expression

Use the `assign` command to change the value of a variable, memory address, or expression that is accessible according to the scope and visibility rules of the language. The expression can be any expression that is valid in the current context.

For C++, use this syntax:

**assign** [classname::]member = ["filename"] ‘expression
**assign** [object.]member = ["filename"] ‘expression

For C++, use the `assign` command to modify static and object data members in a class, and variables declared as reference types, type `const`, or type `static`. The address referred to by a reference type cannot be changed, but the value at that address can be changed.

**attach** process_id image_file

Use the `attach` command to connect to a running process. Supply the process ID number and image file name.

**call** function([parameter])

Use the `call` command to execute a single function. Specify the function as if you were calling it from within the program. If the function has no parameters, specify empty parentheses.

You can nest called functions by setting a breakpoint in a function and executing it using the `call` command. When execution suspends at the breakpoint, use this command to execute the nested function.

For multithreaded applications, the call is made in the context of the current thread.

For C++: When you set the `$overloadmenu` debugger variable to 1 and call an overloaded function, the debugger will list the overloaded functions and call the function you specify.

**catch** [signal]

Enter the `catch` command or the `ignore` command without an argument to see which operating system signals the debugger currently traps or ignores. Use the `catch` command or the `ignore` command followed by an argument to trap or ignore, respectively, that signal.

Only one signal at a time can be added to, or removed from, the list of trapped or ignored signals.

The `catch` and `ignore` commands operate on a per program basis; you must first load a program (using the `load` command) or attach to a process (using the `attach` command).

**catch unaligned**

Enter the `catch unaligned` command to instruct the debugger to stop when unaligned data access occurs in the debuggee process. The debugger stops at the instruction following the instruction where the unaligned access occurs, and issues a message. The default is `ignore unaligned.`

**class** [classname]

For C++ only: Use the `class` command without an argument to display the current class scope. Specify an argument to change the class scope. After the class scope is set, refer to members of the class by omitting the class name prefix.

Setting the class scope nullifies the function scope.

**cont** [signal]

Use the `cont` command without a parameter value to resume program execution until a breakpoint, a signal, an error, or the end of the program is encountered. Specify a parameter value to send an operating system signal to the program continuing execution.

For multithreaded applications, use the `cont` command to resume execution of the current thread that was put on hold. As the current thread resumes, the other threads run freely.

The signal parameter value can be either a signal number or a string name (for example, SIGSEGV). The default is 0, which allows the program to continue execution without specifying a signal. If you specify a signal parameter value, the program continues execution with that signal.

**delete** integer [, . . . ]
**delete all**
**delete ***

Enter the `delete` command followed by the number or numbers associated with a breakpoint or trace (as listed by the `status` command) to remove the specified breakpoint or tracepoint.

Enter the `delete all` command or the `delete *` command to remove all breakpoints and tracepoints.

This command operates on a per program basis; you must first load a program (using the `load` command) or attach to a process (using the `attach` command).

**delsharedobj** shared_object

Use the `delsharedobj` command to remove the symbol table information for the specified shared object from the debugger.

**detach** [process_id_list]

Use the `detach` command to detach from a running process you specify from the process ID list. Specifying a process ID and detaching from the process disables your ability to debug the process. If you do not specify a process ID from the list, Ladebug detaches from the current process.

**disable** integer [, . . . ]
**disable all**
**disable ***

Enter the `disable` command followed by the number or numbers associated with a breakpoint or trace (as listed by the `status` command) to disable the breakpoint or trace. Enter the `disable all` command or the `disable *` command to disable all breakpoints and traces.

The disabled breakpoint is displayed in response to the `status` command but it is ignored during execution. Disabled breakpoints remain disabled until they are explicitly reactivated or deleted.

This command operates on a per program basis; you must first load a program (using the `load` command) or attach to a process (using the `attach` command).

**down** [number]

Use the `up` command or the `down` command without an argument to change the function scope to the function located one level up or down the stack. Enter a number argument to change the function scope to the function up or down the specified number of levels. If the number of levels exceeds the number of active functions on the stack, the function scope moves up or down as many levels as possible and the debugger issues a warning message.

When the function scope changes, the debugger displays the source line corresponding to the last point of execution in that function.

**dump** [function]
**dump.**

Use the `dump` command without an argument to list the parameters and local variables in the current function. To list the parameters and local variables in an active function, specify it as an argument.

Enter the `dump.` command (include the dot) to list the parameters and local variables for all functions active on the stack.

**enable** integer [, . . . ]
**enable all**
**enable ***

Enter the `enable` command followed by the number or numbers associated with a breakpoint or trace (as listed by the `status` command) to enable a breakpoint or trace.

Enter the `enable all` command or the `enable *` command to activate all previously disabled breakpoints and traces.

This command operates on a per program basis; you must first load a program (using the `load` command) or attach to a process (using the `attach` command).

**export** [env_variable [= value]]

Sets the value of the specified environment variable. If no variable is specified, the command displays the values of all environment variables. If a variable is specified but no value is specified, then the variable is set to NULL.

This command is not for the current debuggee's environment, but for the environment of any debuggees subsequently created with `fork(2)` or with subsequent `run` or `rerun` commands.

`export` and `setenv` are synonyms.

See Chapter 9 for more information on commands that manipulate the environment of subsequent debuggees.

**file** filename

Enter the `file` command without an argument to display the name of the current file scope. Include a file name as an argument to change the file scope. Change the file scope to set a breakpoint in a function not in the file currently being executed. To see source code for a function not in the file currently being executed, use the `file` command to set the file scope and the `use` command to direct the search for the sources of that file.

**func** [function]
**func** [integer]

Use the `func` command without an argument to display the current function scope. To change the function scope to a function currently active on the stack, specify either the name of the function or the number corresponding to the stack activation level. (Enter the `where` command to display the stack trace.)

When the function scope is changed, the debugger displays the source line corresponding to the last point of execution in that function.

**goto** linenumber

Use the `goto` command to branch to a line located in the function where execution is suspended. When you branch to a line, the source code between the line where execution suspended and the specified line is not executed.

**help** [command]

Enter the `help` command without an argument to display a list of debugger commands. Include a command argument to display a description of that command.

**history** [integer]

Enter the `history` command without an argument to display previously executed commands. The debugger displays the number of command lines defined for the `$historylines` debugger variable. (The default is 20 lines of user input.) Include an integer argument to display that number of previous commands.

**ignore** [signal]

Enter the `catch` command or the `ignore` command without an argument to see which operating system signals the debugger currently traps or ignores. Use the `catch` command or the `ignore` command followed by an argument to trap or ignore, respectively, that signal.

Only one signal at a time can be added to, or removed from, the list of trapped or ignored signals.

The `catch` and `ignore` commands operate on a per program basis; you must first load a program (using the `load` command or attach to a process (using the `attach` command).

**ignore unaligned**

Enter the `ignore unaligned` command, which is the default, to instruct the debugger not to stop when unaligned access occurs. (Also see the `catch unaligned` command.)

**kill**

Use the `kill` command to terminate the program process leaving the debugger running. When a process terminates, any breakpoints and traces previously set are retained. You can later rerun the program.

**kps**

Use the `kps` command to list all system processes. (This command is valid for local kernel debugging only.)

**list** startline [,endline]
**list** startline[:count]
**list** function

The `list` command displays source-code lines beginning with the source line corresponding to the position of the program counter, the last line listed if multiple list commands are entered, or the line number specified as the first argument to the `list` command. Specify the exact range of source-code lines by including either the endline or the number of lines you want to display. The arguments can be expressions that evaluate to integer values.

To display the source code for a particular function, enter the function as an argument.

**listobj**

Use the `listobj` command to list all loaded objects, including the main image and the shared libraries. For each object, the information listed consists of the full object name (with pathname), the starting address for the text, the size of the text region, and whether the symbol table information has been read by the debugger.

**load** [image_file [core_file]]
**unload** process_id_list or image_file

The `load` command loads an image file and optionally a core file. After loading an image file, enter the `run` command to start program execution.

The `unload` command removes all related symbol table information that the debugger associated with the process being debugged, specified by either a process ID or image file.

**next**

Use the `next` and the `step` commands to execute a line of source code. When the next line to be executed contains a function call, the `next` command executes the function being called and returns to the line immediately after the function call. The `step` command steps into the function and executes only the first line of the function.

For multithreaded applications, use these commands to step the current thread while putting all other threads on hold.

**nexti**

Use the `stepi` command and the `nexti` command to execute a machine instruction. When the instruction contains a function call, the `stepi` command steps into the function being called, and the `nexti` command executes the function being called.

For multithreaded applications, use these commands to step the current thread while putting all other threads on hold.

**patch** expression1 = expression2

Use the `patch` command to correct bad data or instructions in executable disk files. The text, initialized data, or read-only data areas can be patched. The `bss` segment, as well as stack and register locations, cannot be patched because they do not exist on disk files.

Use this command exclusively when you need to change the ondisk binary. Use the `assign` command when you need only to modify debuggee memory.

If the image is executing when you issue the `patch` command, the corresponding location in the debuggee address space is updated as well. (The debuggee is updated regardless of whether the patch to disk succeeded, as long as the source and destination expressions can be processed by the `assign` command.) If the user program is loaded but not yet started, the patch to disk is performed without the corresponding `assign` to memory.

**playback input** filename

Use the `source` command or the `playback input` command to execute debugger commands contained within a file. (Note that you can also execute debugger commands when you invoke Ladebug by creating an initialization file named `.dbxinit.` By default, Ladebug searches for this file during startup, first in the current directory; if it is not there, Ladebug searches your home directory for the file.) Format the commands as if they were entered at the debugger prompt.

When a command file is executed the value of the `$pimode` debugger variable determines whether the commands are echoed. If the `$pimode` variable is set to 1, commands are echoed; if `$pimode` is set to 0 (the default), commands are not echoed. The debugger output resulting from the commands is always echoed.

**pop** [number_of_frames]

The `pop` command removes one or more execution frames from the call stack, undoing the work already done by the removed execution frames. The optional argument is the number of execution frames to remove from the call stack. If you do not specify the argument, one frame is removed. If specified, the number must be a positive integer less than or equal to the number of frames currently on the call stack.

See Chapter 9 for more information.

**print** expression [, . . . ]
**print** @expression

The `print` command displays the current value of a variable or expression that is visible in the current context, as defined by the scope and visibility rules of the program language. The expression can be any expression that is valid in the current context.

The `print @line_number` command displays the address of the specified line number. For example, `print @10` displays the address of line number 10.

For C++:

**print *this**
**print** object
**print** [object.]member
**print *(** derived_class ***)** object

For C++, use the `print` command to display the value of an object, including inherited members and the value of data members in an object. Type casts can be used to interpret a base class object as a derived class object, or to interpret a derived class object as a base class object. To interpret a base class type object as the derived type, use the `print *` command.

**printenv** [env_variable]

Displays the value of the specified environment variable. If none is specified, the command displays the value of all environment variables.

This command is not for the current debuggee's environment, but for the environment of any debuggees subsequently created with `fork(2)` or with subsequent `run` or `rerun` commands.

See Chapter 9 for more information on commands that manipulate the environment of subsequent debuggees.

**printf** [format [,expression , . . . ]]

Use the `printf` command to format and display a complex structure. The *format* argument is a string expression of characters and conversion specifications using the same format specifiers as the `printf` C function.

**printregs**

Use the `printregs` command to display the contents of all machine registers for the current thread. Register values are displayed in decimal or hexadecimal, depending on the value of the `$hexints` variable. The list of registers displayed by the debugger is machine dependent.

**process**
**process** [process_id | image_file | debugger_variable]

Specify a specific process using the process ID number or the name of the image. Ladebug sets the current process context to the process ID or the process that runs the binary image. If there is more than one process running the same binary image, Ladebug warns you and leaves the process context unchanged. The debugger variables `$childprocess` and `$parentprocess` can also be specified in place of the process ID. (Ladebug automatically sets these variables when an application forks a child process.)

**quit**

Use the `quit` command to end the debugging session and return to the operating system prompt.

**readsharedobj** shared_object

Use the `readsharedobj` command to read in the symbol table information for the specified shared object. This object must be a shared library or loadable kernel module. The command can be used only when a debuggee program is specified; that is, either Ladebug has been invoked with it, or the debuggee was loaded by the `load` command.

**record input** filename
**record output** filename
**record io** filename

Use the `record input` command to save all the debugger commands to a file. The commands in the file can be executed by using the `source` command or the `playback input` command.

The `record output` command saves all debugger output to a file. The output is simultaneously echoed to the screen. (The program output is not saved.)

The `record io` command saves both input to and output from the debugger.

To stop recording debugger input or output, exit the debugger.

**return** [function]

Use the `return` command without an argument to continue execution of the current function until it returns to its caller. If you include a function name, execution continues until control is returned to the specified function. The function must be active on the call stack.

**run** [program_arguments] [io_redirection]
**rerun** [program_arguments] [io_redirections]

Use the `run` and `rerun` commands to start program execution. Enter program flags, options, and input and output redirections as arguments. If the `rerun` command is specified without arguments, the arguments entered with the previous `run` command are used.

If the last modification time and/or size of the binary file or any of the shared objects used by the binary file has changed since the last `run` command was issued, Ladebug automatically rereads the symbol table information. If this happens, the old breakpoint settings may no longer be valid after the new symbol table information is read.

**set** [variable = definition]
**unset** variable

To examine the definitions of all debugger variables, enter the `set` command without arguments. (Use the `print` command to display the definition of a single debugger variable.)

To define a debugger variable, enter the `set` command followed by a variable name, an equal sign, and a definition. Enclose string definitions in quotes. The definition can be any expression allowed by the language of the program being debugged.

Use the `unset` command to delete a variable.

If you want to remove an environment variable, or all environment variables, use the `unsetenv` command, not the `unset` command.

The debugger contains many predefined variables to describe the current state of the debugger, and to allow you to customize the debugger environment. You can delete and redefine the predefined debugger variables in the same way you define your own variables. If you delete a predefined debugger variable, the debugger uses the default value for that variable. The settings on the predefined variables apply to all debugging processes.

The debugger has the following predefined variables:

| | |
|---|---|
| `$ascii` | With the default value of 1, allows the `print` command to print character-type data as ASCII characters, only when the bit value is that of a printable 7-bit subset ASCII character. (Other bit values are printed as octal numbers.) With a value of 0, this variable causes all printable ISO Latin-1 characters to be printed as characters. |
| `$beep` | With the default value of 1, causes a beep to sound when a user attempts to perform an illegal action while editing the debugger command line (for example, moving the cursor past the end of the line, or “yanking”—pasting – from an empty cut buffer). |
| `$childprocess` | Can be specified in place of the process ID. (Ladebug automatically sets this variable when an application forks a child process.) |
| `$catchexecs` | When set to 1, instructs the debugger to notify the user and stop the program when a program execs. The default is 0. |
| `$catchforks` | When set to 1, instructs the debugger to notify the user when a program forks a child process. The child process stops and is brought under debugger control. (By default, the parent process is not stopped. See the `$stopparentonfork` description.) The default is 0. |
| `$curevent` | Is set to the event number of the current event at the start of an event, allowing its use within the expression of an event. |

| | |
|---|---|
| `$curfile` | Specifies the current source-code file. |
| `$curline` | Specifies the current line in the source file. |
| `$curpc` | Specifies the current point of program execution. This variable is used by the `wi` and `li` aliases. |
| `$cursrcline` | Specifies the line number of the last line of source code listed plus one. |
| `$curthread` | Indicates the thread ID of the current thread within the current process. You can change to a different thread by setting `$curthread`. |
| `$decints` | When set to 1, all integers printed by the debugger are displayed as decimal numbers. The default is 0. |
| `$editline` | With the default of 1, enables the command-line editing features. The command-line editing features are described in Chapter 7. For backward compatibility you can set this variable to 0. |
| `$eventecho` | When set to 1, echoes events (such as breakpoints) with event numbers when executed. The default is 1. |
| `$hasmeta` | For international users. When set to 1, causes any 8-bit character to be interpreted as the corresponding 7-bit character plus the Meta character (which is the ALT key whose MSB bit represents a Meta modifier). This could be used for binding editing functions to ALT plus key sequences. The default depends on several factors, including the locale and whether the terminal has Meta capability. In the United States, the default is usually 0. |
| `$hexints` | When set to 1, all integers will be displayed as hexadecimal numbers. The default is 0. |
| `$historylines` | Specifies the number of previously input commands listed in response to the `history` command. The default is 20. |
| `$indent` | When set to 1, specifies that structures will be printed with added indentation to render them more readable. The default value is 1. |

| | |
|---|---|
| `$lang` | Specifies the programming language used in the program being debugged. For mixed-language programs, `$lang` is set to the language corresponding to the current frame. The variable is updated when the program execution stops. |
| `$listwindow` | Specifies how many lines the `list` command displays. The default is 20. |
| `$main` | Specifies the name of the function that the debugger enters first. The default is `main( )` , but this can be set to any function. This variable is useful when debugging languages whose primary function is called something other than `main( )` . |
| `$maxstrlen` | Specifies the maximum number of characters to print when the value of a string is printed. The default is 128. |
| `$octints` | When set to 1, the debugger prints all integers as octal numbers. The default is 0. |
| `$overloadmenu` | When debugging C++ programs only, if this variable is set to 1, a menu of overloaded functions is displayed so you can select the desired function from the list of type signatures of the function. When set to 0, no menu is displayed and you must disambiguate overloaded functions by providing the full type signature. The default is 1. |
| `$parentprocess` | Can be specified in place of the process ID. (Ladebug automatically sets this variable when an application forks a child process.) |
| `$pid` | Indicates the process ID of the current process. Only for use in kernel debugging (either local or remote). |
| `$pimode` | Specifies whether the `playback input` command echoes input. If set to 1, commands from the script file are echoed. If set to 0, commands are not echoed. The default is 0. |
| `$prompt` | Specifies the debugger prompt. The default is `(ladebug)`. |
| `$repeatmode` | When set to 1, causes the debugger to repeat the previous command if you press the Return key at the `(ladebug)` prompt. The default is 1. |

| | |
|---|---|
| `$stackargs` | When set to 1, causes the values of arguments to be included in the output of the `where`, `up`, `down` , and `dump` commands. When large or complex values are passed by value, the output can be voluminous. You can set `$stackargs` to 0 to suppress the output of argument values. The default is 1. |
| `$stepg0` | When set to 0, the debugger steps over calls to routines that are compiled without symbol information. When set to 1, the debugger steps into these calls. The default is 0. |
| `$stoponattach` | When set to 1, causes the debugger to stop a running process right after attaching to it. When set to 0, causes the debugger to allow the process to run to completion; in this case, to interrupt the process, enter Ctrl/C. The default is 0. |
| `$stopparentonfork` | When set to 1, instructs the debugger to stop the parent process when a program forks a child process. (Also see the `$catchforks` description.) The default is 0. |
| `$threadlevel` | Enables Ladebug to determine whether you are working with DECthreads or native threads. The default is `decthreads` if the application is multithreaded and is using DECthreads. Otherwise, the default is `native.` You can switch from one mode to the other by setting `$threadlevel.` In kernel mode, `$threadlevel` is always `native.` |
| `$tid` | Indicates the thread ID of the current thread within the current process. You can change to a different thread by setting `$tid.` Only for use in kernel debugging (either local or remote). |
| `$verbose` | When set to 1, specifies whether debugger output should include all possible program information, including base pointers and virtual function pointers (for C++ programs only). The default is 0. |

**setenv** [env_variable [value]]

Sets the value of the specified environment variable. If no variable is specified, the command displays the values of all environment variables. If a variable is specified but no value is specified, then the variable is set to NULL.

This command is not for the current debuggee's environment, but for the environment of any debuggees subsequently created with `fork(2)` or with subsequent `run` or `rerun` commands.

`export` and `setenv` are synonyms.

See Chapter 9 for more information on commands that manipulate the environment of subsequent debuggees.

**sh** command [argument . . . ]

Use the `sh` command to execute a Bourne shell command. Do not enclose the shell command and its arguments in quotations.

**Example:**

```
(ladebug) sh ls -l sample.c
```

**show condition** [condition_identifier_list]
**show condition** [condition_identifier_list] with state = = wait

For DECthreads only. Use the `show condition` command to list information about currently available DECthreads condition variables. If you supply one or more condition identifiers, the debugger displays information about those condition variables that you specify, provided that the list matches the identity of currently available condition variables. If you omit the condition variable specification, the debugger displays information about all condition variables currently available.

Use the `show condition with state == wait` command to display information only for condition variables that have one or more threads waiting. If `$verbose` is set to 1, the sequence numbers of the threads waiting on the condition variables are displayed.

**show mutex** [mutex_identifier_list]
**show mutex** [mutex_identifier_list] with state = = locked

For DECthreads only. Use the `show mutex` command to list information about currently available mutexes. If you specify one or more mutex identifiers, the debugger displays information about only those mutexes that you specify, provided that the list matches the identity of currently available mutexes. If you omit the mutex identifier specification, the debugger displays information about all mutexes currently available.

Use the `show mutex with state == locked` command to display information exclusively for locked mutexes. If `$verbose` is set to 1, the sequence numbers of the threads locking the mutexes are displayed.

**show process**
**show process ***
**show process all**

Use the `show process` command to display information for the current process. The `show process *` and `show process all` commands display information for all processes.

**show thread** [thread_identifier_list]
**show thread** [thread_identifier_list] with state = = ready
**show thread** [thread_identifier_list] with state = = blocked
**show thread** [thread_identifier_list] with state = = running
**show thread** [thread_identifier_list] with state = = terminated
**show thread** [thread_identifier_list] with state = =detached
**show thread** [thread_identifier_list] with state = =stopped

Use the `show thread` command to list all the threads known to the debugger. If you specify one or more thread identifiers, the debugger displays information about the threads that you specify, if the thread matches what you specified in the list. If you omit a thread specification, the debugger displays information for all threads.

Use the `show thread with state` command to list only the threads with a specific state (characteristic).

The valid state values for DECthreads are `ready`, `blocked`, `running`, `terminated`, and `detached`.

The valid state values for native theads are `stopped`, `running`, and `terminated`.

**source** filename

Use the `source` command or the `playback input` command to execute debugger commands contained within a file. (Note that you can also execute debugger commands when you invoke Ladebug by creating an initialization file named `.dbxinit`. By default, Ladebug searches for this file during startup, first in the current directory; if it is not there, Ladebug searches your home directory for the file.) Format the commands as if they were entered at the debugger prompt.

When a command file is executed, the value of the `$pimode` debugger variable determines whether the commands are echoed. If the `$pimode` variable is set to 1, commands are echoed; if `$pimode` is set to 0 (the default), commands are not echoed. The debugger output resulting from the commands is always echoed.

**status**

The `status` command lists all breakpoints and tracepoints, the reference number associated with each, and whether the breakpoint is disabled. All breakpoint settings are on a per process basis.

**step**

Use the `next` and the `step` commands to execute a line of source code. When the next line to be executed contains a function call, the `next` command executes the function being called and returns to the line immediately after the function call. The `step` command steps into the function and executes only the first line of the function.

For multithreaded applications, use these commands to step the current thread while putting all other threads on hold.

**stepi**

Use the `stepi` command and the `nexti` command to execute a machine instruction. When the instruction contains a function call, the `stepi` command steps into the function being called, and the `nexti` command executes the function being called.

For multithreaded applications, use these commands to step the current thread while putting all other threads on hold.

**stop** variable
**stop** [variable] **if** expression
**stop** [variable] **at** linenumber [**if** expression]
**stop** [variable] "filename":linenumber
**stop** [variable] **in** function [**if** expression]
**stop** [variable] "filename" ‘function [**if** expression]
**stop** [variable] [**thread** thread_identifier_list] [**at** linenumber] [**if** expression]
**stop** [variable] [**thread** thread_identifier_list] [**in** function] [**if** expression]

Enter the `stop` command without a variable argument to suspend program execution and return to the prompt. (All breakpoint settings are on a per process basis.)

Enter the `stop` command with a variable argument to suspend program execution when the variable changes.

Specify `if` with an expression to suspend execution when the expression evaluates to true. When you specify both an expression and a variable, execution suspends only if the specified expression evaluates to true and the variable has changed.

To suspend execution when a line or function is encountered, use the third, fourth, or fifth syntax form. If you specify a variable, execution suspends only if the variable has changed when the line or function is encountered.

If you specify an expression, execution suspends only if the expression evaluates to true when the line or function is encountered.

If you specify both a variable and an expression, execution suspends only if the variable has changed and the expression evaluates to true when the line or function is encountered.

Specify a filename and function to instruct Ladebug to stop in a particular function in the specified file, thus eliminating ambiguity.

Use the seventh and eighth forms of the syntax for multithreaded applications. The `thread_identifier_list` parameter identifies one or more threads of the current debugging level (native or DECthreads). If you specify one or more thread identifiers, the breakpoint is set only in those threads. If you do not specify any thread identifier, the breakpoint is set on all threads or at the process level.

The following example shows how to suspend program execution when line number 10 is encountered and the variable `f` equals 2:

```
(ladebug) stop at 10 if f==2
```

For C++:

**stop in** [classname::]function
**stop in** object.function
**stop in** objectptr->function
**stop in** object.function **if** (& object = = this)
**stop in** [classname::]classname (type_signature)
**stop in** [classname::]~classname
**stop in** [classname::] function [(type_signature)] [(void)]
**stop in all** function

For C++ only.

The first form of the `stop` command sets a breakpoint in a member function using the static class type information. This form presumes that run-time information from an object is needed to determine the address of the function at which to set the breakpoint.

If you need run-time information to determine the correct virtual function, use the second or third syntax form to qualify the function name with the object when you set the breakpoint. This way of setting the breakpoint causes the debugger to stop at the member function in all objects declared with the same class type as the specified object.

To set a breakpoint that stops only in the member function for this specific object and not all instances of the same class type, use the fourth form of the `stop` command.

The fifth and sixth syntax forms set breakpoints in a constructor and destructor, respectively.

To set a breakpoint in a specific version of an overloaded function, either set `$overloadmenu` to 1, enter the command `stop in` *`function`*, and choose the appropriate function from the menu, or specify the function and the type signature as arguments to the command. If the overloaded function has no parameters, `void` must be explicitly specified.

Use the last form to set a breakpoint in all versions of an overloaded function.

**Example:**

```
(ladebug) stop at 10 if f==2
```

**stopi** variable
**stopi** [variable] **if** expression
**stopi** [variable] **at** address [**if** expression]

Enter the `stopi` command with a variable to suspend execution when the variable value changes.

Specify `if` with an expression to suspend execution when the expression evaluates to true. When you specify both a variable and an expression, execution suspends only if the specified expression evaluates to true and the variable has changed.

To suspend execution when an address is encountered, use the third syntax form. If you specify a variable, execution suspends only if the variable has changed when the address is encountered. If you specify an expression, execution suspends only if the expression evaluates to true when the address is encountered. If you specify both a variable and an expression, execution suspends only if the variable has changed and the expression evaluates to true when the address is encountered.

The `stopi` command is different from the `stop` command because the debugger checks the breakpoint set with the `stopi` command after executing each machine instruction. Thus, the debugger performance is affected when you use the `stopi` command.

As of Version 4.0 of the debugger, the `stopi in` command is no longer valid, and results in an error message. Replace `stopi in` in your code with `stopi at` for an address or `stop in` for a routine.

**thread** [thread_identifier]

Use the `thread` command to identify or set the current thread context. If you supply a thread identifier, the debugger sets the current context to the thread you specify. If you omit the thread identifier, the debugger displays the current thread context.

The debugger interprets the thread identifier as a DECthreads or kernel thread identifier depending on the value of the debugger variable `$threadlevel`.

**trace** variable [**if** expression]
**trace** [variable] **at** line_number [**if** expression]
**trace** [variable] **in** function [**if** expression]
**trace** [variable] [**thread** thread_identifier_list] [**at** line_number] [**if** expression]
**trace** [variable] [**thread** thread_identifier_list] [**in** function] [**if** expression]

When you use the `trace` command without an argument, the debugger prints a message but does not suspend program execution when each function is entered. Specify a variable to print a message when the variable value changes. Specify `if` with an expression to print a message when an expression evaluates to true. When you specify both a variable and an expression, a message is printed only if the expression evaluates to true and the variable has changed.

To print a message when a line or function is encountered, use the second or third syntax form.

If you specify a variable, a message is printed only if the variable has changed when the line or function is encountered.

If you specify an expression, a message is printed only if the expression evaluates to true when the line or function is encountered.

If you specify both a variable and an expression, a message is printed only if the variable has changed and the expression evaluates to true when the line or function is encountered.

The following example traces the variable `f` when the program is executing the function `main`:

```
(ladebug) trace f in main
```

Use the `trace thread` command to set tracepoints in specific threads. If you list one or more thread identifiers, the debugger sets a tracepoint only in those threads you specify. If you omit the thread identifier specification, the debugger sets a tracepoint in all the threads of the application.

**tracei** [variable] [**if** expression]
**tracei** [variable] **at** address [**if** expression]
**tracei** [variable] **in** function [**if** expression]
**tracei** [variable] [**thread** thread_identifier_list] [**at** line_number] [**if** expression]
**tracei** [variable] [**thread** thread_identifier_list] [**in** function] [**if** expression]

When you use the `tracei` command the debugger prints a message, but does not suspend program execution. Specify a variable to print a message when the variable value changes. Specify an expression to print a message when an expression evaluates to true. When you specify both a variable and an expression, a message is printed only if the expression evaluates to true and the variable has changed.

To print a message when an address or function is encountered, use the second or third syntax form.

If you specify a variable, a message is printed only if the variable has changed when the address or function is encountered.

If you specify an expression, a message is printed only if the expression evaluates to true when the address or function is encountered.

If you specify both a variable and an expression, a message is printed only if the variable has changed and the expression evaluates to true when the address or function is encountered.

The `tracei` command differs from the `trace` command in that the debugger evaluates the tracepoint set with the `tracei` command after the debugger executes each machine instruction. Thus, when you use the `tracei` command, the debugger performance is affected.

In the following example, a breakpoint is set to print a message every time the function `factorial` is entered:

```
(ladebug) tracei factorial
```

Use the `tracei thread` command to set tracepoints in specific threads. If you list one or more thread identifiers, the debugger sets a tracepoint only in those threads you specify. If you omit the thread identifier specification, the debugger sets a tracepoint in all the threads of the application.

**unsetenv** [env_variable]

Removes the specified environment variable. If no variable is specified, all environment variables are removed.

This command is not for the current debuggee's environment, but for the environment of any debuggees subsequently created with `fork(2)` or with subsequent `run` or `rerun` commands.

See Chapter 9 for more information on commands that manipulate the environment of subsequent debuggees.

**up** [number]

Use the `up` command or the `down` command without an argument to change the function scope to the function located one level up or down the stack. Enter a number argument to change the function scope to the function up or down the specified number of levels. If the number of levels exceeds the number of active functions on the stack, the function scope moves up or down as many levels as possible and the debugger issues a warning message.

When the function scope changes, the debugger displays the source line corresponding to the last point of execution in that function.

**use** [directory]
**unuse** [directory]
**unuse ***

Enter the `use` command without an argument to list the directories the debugger searches for source-code files. Specify a directory argument to make source-code files in that directory available to the debugger. (You can also use the `ladebug` command `-I` option to specify search directories.)

Enter the `unuse` command without an argument to set the search list to the default, the home directory, the current directory, and the directory containing the executable file. Include the name of a directory to remove it from the search list. The asterisk `*` argument removes all directories from the search list.

**whatis** expression

The `whatis` command prints the type of the specified expression. The expression can be any expression that follows the syntax, scope, and visibility rules of the program language.

For C++:

**whatis** classname
**whatis** [classname::]member
**whatis** [classname::]function

The first syntax form of the `whatis` command for C++ displays the class type. The second form displays the type of a member function or data member. To display all versions of an overloaded function, use the third form.

**when** {command [, . . . ]}
**when if** expression {command [, . . . ]}
**when at** linenumber [**if** expression] {command [, . . . ]}
**when in** function [**if** expression] {command [, . . . ]}
**when** [**thread** thread_identifier_list] [**at** line_number] [**if** expression] {command [; . . . ]}
**when** [**thread** thread_identifier_list] [**in** function] [**if** expression] {command [; . . . ]}

Use the `when` command to execute the specified command. (The `when` command does not suspend program execution.) The debugger command must be enclosed in braces. Separate multiple commands with semicolons.

To execute a command when an expression evaluates to true, use the second syntax form. To execute commands when a line or function is encountered, use the third or fourth syntax form.

If you specify an expression, the command is executed only if the expression evaluates true when the line or function is encountered.

**Example:**

```
(ladebug) when at 5 {list;where}
```

Use the `when thread` command to set tracepoints in specific threads. If you list one or more thread identifiers, the debugger sets a tracepoint only in those threads you specify. If you omit the thread identifier specification, the debugger sets a tracepoint in all the threads of the application.

**wheni** {command [, . . . ]}
**wheni if** expression {command [, . . . ]}
**wheni at** linenumber [**if** expression] {command [, . . . ]}
**wheni in** function [**if** expression] {command [, . . . ]}
**wheni** [**thread** thread_identifier_list] [**at** line_number] [**if** expression] {command [; . . . ]}
**wheni** [**thread** thread_identifier_list] [**in** function] [**if** expression] {command [; . . . ]}

Use the `wheni` command to execute the specified command. (The `wheni` command does not suspend program execution.) The debugger command must be enclosed in braces. Separate multiple commands with semicolons.

To execute a command when an expression evaluates to true, use the second syntax form. To execute a command when an address or function is encountered, use the third or fourth form.

If you specify an expression, the command is executed only if the expression evaluates to true when the address or function is encountered.

The `wheni` command differs from the `when` command in that the debugger evaluates the tracepoint set with the `wheni` command after each machine instruction is executed. Thus, using the `wheni` command affects performance.

For example, the following command stops program execution, lists ten lines of source code and displays the stack trace when the value of the variable `i` is equal to `3` in the function `main`:

```
(ladebug) wheni in main if i == 3 {wi;where}
```

Use the `wheni thread` command to set tracepoints in specific threads. If you list one or more thread identifiers, the debugger sets a tracepoint only in those threads you specify. If you omit the thread identifier specification, the debugger sets a tracepoint in all the threads of the application.

**where** [number]
**where** [number] **thread** thread_identifier_list
**where** [number] **thread all**
**where** [number] **thread ***

The `where` command displays the stack trace of currently active functions, for the current thread. The `where thread thread_identifier_list` command displays the stack trace(s) of the specified thread(s). The `where thread all` and the `where thread *` commands are equivalent; they display the stack traces of all threads.

Include the optional `number` argument to see a specified number of levels at the top of the stack. (Each active function is designated by a number, which can be used as an argument to the `func` command. The top level on the stack is 0; so if you enter the command `where 3`, you will see levels 0, 1, and 2.) If you do not specify the `number` argument, you will see all levels.

**whereis** expression

The `whereis` command shows all declarations of the expression. Each declaration is fully qualified with scope information.

**which** expression

The `which` command shows the fully qualified scope information for the instance of the specified expression in the current scope. If available to the debugger, the name of the source file containing the function in which the expression is declared, the name of the function, and the name of the expression are included. The components of the qualification are separated by period (.) characters.

## Restrictions

The maximum command-line length is 255 characters.
Alias commands can contain no more than 56 arguments.

## Related Information

```
ada(1)
c89(1)
cc(1)
cxx(1)
cobol(1)
f77(1)
f90(1)
printf(1)
signal(3)
```

# A
# Using Ladebug Within emacs

This chapter describes how to invoke and use the Ladebug debugger within the emacs editing environment. You can control your debugger process entirely through the emacs GUD buffer (see Section A.3), which is a variant of Shell mode. All the Ladebug commands are available, and you can use the Shell mode history commands to repeat them.

Ladebug Version 3.0 supports GNU Emacs Version 19.22 and above.

The information in the following sections assumes the user is familiar with emacs and is using the emacs notation for naming keys and key sequences.

## A.1 Loading Ladebug-Specific emacs Lisp Code

For each emacs session, before you can invoke Ladebug, you must load the Ladebug-specific emacs lisp code, as follows:

```
M-x load-file /usr/lib/emacs/lisp/ladebug.el
```

You can also place a `load-file` command in your emacs initialization file (~/.emacs). For example,

```
load-file " . . .  /ladebug.el"
```

## A.2 Invoking Ladebug Within emacs

To start Ladebug within emacs and specify the name of the target program you want to debug, enter:

```
M-x ladebug [target-program]
```

## A.3 emacs Debugging Buffers

When you start Ladebug, `emacs` displays the GUD (Grand Unified Debugger) buffer in which the `(ladebug)` prompt is displayed. The GUD buffer saves all of the commands you enter and the program output for you to edit.

When you issue the Ladebug `run` command in the GUD buffer and hit a breakpoint, `emacs` displays the source of your program in a second buffer (source buffer) and indicates the current execution line with `=>`.

By default, `emacs` sets its current working directory to be the directory containing the target program. Ladebug does not do this when invoked directly, therefore you may need to change the source code search path when using Ladebug from within `emacs`. To set an alternate source code search path, enter the `use` command with a directory argument, for example:

```
(ladebug) use usr/prog/test
```

All `emacs` editing functions and GUD key bindings are available. For example:

- You can execute a `step` command by entering the command in the GUD buffer.
- You can select a line of code in the current source buffer and enter a command to set a breakpoint at that position by typing

  ```
  C-x SPC
  ```

For more information on `emacs` functionality and key bindings, refer to `emacs` documentation.

# B

# Writing a Remote Debugger Server

This appendix describes how to write a Ladebug remote debugger server for an Alpha target (operating system or hardware platform). It describes the functionality required of the server and explains how this functionality is implemented in the Digital UNIX server and in the server included in the debugger monitors for the Alpha server evaluation boards. It also includes information about writing new Ladebug remote debugger servers.

## B.1 Reasons for Using a Remote Debugger

The main reason for using a remote debugger is to debug software on a system that cannot run a fully functional debugger locally. Examples of when you might use a remote debugger are:

- When debugging an embedded system that does not include displays or keyboards.
- When porting an operating system. You cannot usually run a debugger locally on the target system until a substantial part of the operating system has been ported.
- When developing software to run on an operating system that does not support a fully functional local debugger.

Remote debuggers are useful simply for debugging software on systems that are remote from where you are working, although in this case there are often other alternatives (for example logging into the system across a network). Whether it is better to use remote debugging or one of these alternatives will often depend on the precise characteristics of the network and the debugger used.

## B.2 Alternatives to Using a Remote Debugger

In most cases, the alternatives to using a remote debugger are either to put debugging code, such as print statements, in the software being debugged or to develop a simple local debugger for the target system.

Adding debugging code to the software increases the complexity of the software, hence causing additional bugs, and it is often difficult to determine what information will be needed to debug the software. A further problem is the debugging code can itself change the behavior of the software and as such normally has to be removed before the software is released.

Developing a local debugger can itself be a major task. Since such a debugger is normally a one-off development, you cannot normally justify including support for high level features (such as source level debugging). Even if this is possible, attempting to provide such facilities locally the target system will often not have the resources (memory, for example) required to run a high level debugger.

## B.3 The Structure of a Remote Debugger

All remote debuggers consist of two parts:

- A client that runs on the user's local system (the host system)
- A server that runs on the system being debugged (the target system)

These communicate through a remote debugger protocol that runs over some communication mechanism such as a serial line or the Internet.

The client provides the user interface to the debugger and most of the intelligence of the debugger. For example, the client will normally do all translation between addresses and variable or function names.

The server makes available low level functions that allow the client to examine and control software running on the target system. For example, the server provides functions to read and modify the target system's memory. The client requests these functions, and receives responses, using the remote debugger protocol.

In general, a server is much simpler than a client and will require only minimal functionality from the target system on which it is running. This allows servers to be implemented for environments in which the functionality of a full workstation operating system is not available.

## B.4 Types of Targets

Most target systems for remote debugging are of these two types:

- Targets on which the debuggees are applications running on top of an operating system such as Digital UNIX or VxWorks. A remote debugger server for such a target will normally communicate with the client using the operating system's networking interface (for example, sockets in Digital UNIX), and will normally use an operating system debugger interface (for example, the `ptrace( )` function) to implement its debugger functions. The Digital UNIX server described in Section B.6.1 is an example of a server for such a target.
- Targets (typically special purpose embedded hardware) on which the debuggees are either stand alone programs or operating systems directly using the hardware. On such targets the server is typically incorporated in a ROM based debugger monitor that drives the network device directly. Such a server will perform its debugger functions by directly reading and writing memory. The Evaluation Board Server described in Section B.6.2 is an example of such a server.

## B.5 Ladebug as a Remote Debugger

Ladebug is a debugger running on Digital UNIX systems. It supports a wide range of languages including Ada, C, C++, COBOL, and Fortran. Besides providing local debugging on Digital UNIX systems, it supports remote debugging through the Ladebug remote debugger protocol. The same text and windows based interfaces are available for use with both local and remote debugging and almost all the commands that are available for local debugging are also available for remote debugging.

### B.5.1 Target and Programming System Requirements

Ladebug servers must be able to:

- Set and clear breakpoints in the program being debugged (the debuggee).
- Read from and write to the debuggee's memory and registers.
- Find out when the debuggee reaches a breakpoint and read the debuggee's memory and registers when it is at a breakpoint.

- Stop a running debuggee when it is not at a breakpoint.

**Note**

Servers can be implemented on targets without this feature. However, without this feature, the Ladebug Ctrl/C function (that stops a running program) will not work.

- Send and receive UDP packets to and from the host system running Ladebug.
- Respond to messages from the host system without excessive delay even when the debuggee is running and is not at a breakpoint.

For Ladebug to debug a program there must be symbolic information for the program available to Ladebug on the host in a form that it understands. At present, the only form of symbolic information that Ladebug understands for programs running on Alpha processors is extended COFF (ECOFF) for Digital UNIX. The program must follow the register usage, function calling, and other conventions expected of programs that have this form of symbolic information. For example, a program for which the symbolic information is ECOFF must use Digital UNIX register usage and function calling conventions.

### B.5.2 The Protocol

The Ladebug Remote Debugger Protocol is a request/response protocol running over UDP. The debugger client (Ladebug) initiates all transactions sending a request to the server. On receiving the request the server acts upon the request and sends a response. The server never sends any messages except in response to requests received from the client. The requests that the client can send are listed in Table B–1.

**Table B–1 Remote Debugger Protocol Client Requests**

| Request | Action |
|---|---|
| Load Process | Loads a new process for debugging. |
| Connect to Process | Connects to an existing process. |
| Connect to Process Insist | Connects to an existing process even if other debugger sessions are already connected to it. |
| Probe Process | Checks the state of the process being debugged. |
| Disconnect from Process | Disconnects, ending a debugger session. |
| Kill Process | Kills the process; and then disconnects. |
| Stop Process | Stops a running process. |
| Continue Process | Continues running a stopped process. |
| Step | Executes one instruction in the process being debugged. |
| Set Breakpoint | Sets a breakpoint at an address. |
| Clear Breakpoint | Clears a breakpoint at an address. |
| Get Next Breakpoint | Gets the "next" breakpoint that is known to the server. Breakpoints are returned in an arbitrary order but no breakpoint will be returned more than once in a single scan of the list. |
| Get Registers | Gets the contents of all the registers. |
| Set Registers | Sets the contents of all the registers. |
| Read | Reads memory. |
| Write | Writes to memory. |

Section B.11 contains a full description of the protocol.

### B.5.3 Starting a Remote Debugger Session

The protocol provides three alternative requests for starting a debugger session:

- Load Process loads a new process on the target system. The Load Process request contains a file name and various other data that a server may need to load the process.
- Connect to Process connects to an existing process running on target system. It simply sends a 32 bit process ID to the target system to identify the new debuggee.

- Connect to Process Insist is identical in form to Connect to Process. It is intended to be interpreted as a request to the server to disconnect, if necessary, any old debugger sessions from the process before connecting the new debugger session to the process. This is needed because normally only one debugger session can debug any one process at a time.

All servers should implement either Load Process or Connect to Process but a server need not implement both of them. A server that implements Connect to Process can choose any of the following:

- Not to implement Connect to Process Insist
- To implement Connect to Process Insist to mean the same as Connect to Process
- To implement Connect to Process Insist to do more than Connect to Process

To allow a single target machine to run multiple remote debugger sessions at the same time, clients always send Connect and Load requests to a fixed privileged, UDP port on the target. The server is expected to allocate a new unprivileged UDP port before replying. The new port is used on the target as the source and destination of all messages for the remainder of the debugger session.

To allow servers to be run on systems on which security is an issue (for example typical Digital UNIX systems) the Connect and Load requests contain the client and server login names. The server can use these login names, together with the name of the host system, to check that the remote user is authorized to run programs on the target system, as is done when a user runs programs through `rsh`.

### B.5.4 Ending a Remote Debugger Session

There are two different requests that end a remote debugger session:

- The Kill Process request ends a debugger session, killing the debuggee at the same time.
- The Disconnect from Process request ends the remote debugger session without killing the debuggee.

Although there are explicit Ladebug commands that call up each of these requests, Ladebug normally kills processes that it has loaded and disconnects from processes to which it has connected. As such, a server that implements the Load Process function should at least implement the Kill Processes function and a server that implements the Connect to Process function should implement the Disconnect from Process function. The following are optional:

- Implementation of the Kill Process function in servers that only support connecting to existing processes
- Implementation of the Disconnect from Process function in servers that only support Load Process

## B.6 Example Servers

Two example Ladebug servers are available. The source code of these example servers is available from Digital Equipment Corporation for unrestricted reuse on Alpha based platforms (see the copyright notice in the source code for details).

### B.6.1 The Digital UNIX Server

The Digital UNIX server is designed to allow the debugging of user processes running on remote Digital UNIX systems. The version described in this section supports loading new processes using the Load Process request but does not support the Connect to Process or Connect to Process Insist requests.

The server consists of a **server daemon** and **user servers**:

- The server daemon receives load messages, checks that they are valid and creates a user server for each valid load request. It then passes the load request on to the user server.
- The user server implements the remainder of the protocol and the low level debugger functions requested by the protocol.

The remote debugger daemon must be run as a root process. It would normally be started at system start-up. The user server loads the debuggee using Digital UNIX's `fork( )` and `exec( )` functions and uses Digital UNIX's `ptrace( )` interface to implement the low level debugger facilities required. The load server and daemon both use Digital UNIX UDP sockets to communicate with the client.

### B.6.2 Evaluation Board Server

The evaluation board server is included in the evaluation board debug monitor provided with the Alpha evaluation boards (EB64, EB64+, EB66, etc.). It is designed to provide source level debugging of operating system kernels being ported to these boards, and of programs running on these boards without an operating system. The complete monitor (including the remote debugger server) is designed to be easily ported to other Alpha based hardware.

The server only supports starting debugger sessions through the Connect to Process or Connect to Process Insist requests. This server does not support the Load Process request. Since the monitor is not a multiprocessing system, the server ignores the process ID in the Connect requests. It also ignores the login names in Connect requests.

The user is expected to load the test program using the monitor load facilities (LOAD, NETLOAD, etc.) before starting the debugger server. The server interprets all addresses it receives as physical addresses. The server then performs all debugger functions by directly reading and writing memory.

To set breakpoints, the debugger patches a PAL call into the code being debugged. To avoid conflict with the use of the breakpoint PAL call by operating system kernels, this is not the standard breakpoint PAL call (the BPT PAL call) but a special PAL call (DBGSTOP). DBGSTOP exhibits identical behaviour but has its own system entry address. It is implemented in the debugger version of evaluation boards' PAL code. When this PAL call is executed, it results in a call back to the monitor at which point the state of the debuggee is saved and the server is reentered.

The monitor's ethernet software allows server to register to receive packets addressed to particular UDP port and to send packets on any UDP port. The server depends on interrupts to receive packets while the debuggee is running. Upon receiving any interrupt, the monitor polls the ethernet driver for messages. The ethernet software passes any appropriate messages to the server.

A consequence of using interrupts to receive messages is that some care is needed when debugging programs that do their own interrupt handling. To allow such programs to be debugged, the Evaluation Board user library contains a function that polls the ethernet. This function would normally be called by the application every time it receives an interrupt.

### B.6.3 Structure of the Servers

Each server consists of:

- A communicator that receives UDP messages from the clients, checks that they come from valid clients, and passes them on to the protocol handler.

  Once the protocol handler has dealt with a message and built a reply it passes this reply back to the communicator. The communicator then sends this reply to the client. The communicator is target dependent. It makes use of the target's UDP functions to read and write messages. In the Digital UNIX server, the communicator also contains the server's main program and the code of the daemon. As such, it is responsible for creating the user servers.

- A protocol handler that interprets the debugger messages received and builds the replies. The protocol handler calls the breakpoint table handler and the debugger kernel to perform the functions requested by the messages. The protocol handler is target independent.
- A target dependent debugger kernel that performs the actual debugger functions (such as loading a process, setting a breakpoint, or reading memory). These are performed by the kernel:
  - Through an operating system interface (for example `ptrace( )` on Digital UNIX)
  - In servers for embedded monitors, by directly reading from and writing to memory
- A breakpoint table handler that creates and controls a table of the breakpoints that the server has set in the debuggee. This table handler is target independent.

### B.6.4 Creating a Server for a New Target

The simplest way of creating a server for a new target is to base it upon one of the example servers. Normally, if you are developing the server to be part of a monitor program, you should base it upon the evaluation board server.

If, however, you are developing it as an operating system utility you should probably base it upon the Digital UNIX server. You should try to make as few changes as possible to the example servers, since you are likely to have no satisfactory way of debugging software (and hence the servers themselves) until you have successfully ported them to your target system.

## B.7 The Communicators

This section describes

- The communicator interface functions (see Section B.7.1)
- The Digital UNIX communicator (see Section B.7.2)
- The evaluation board monitor (see Section B.7.3)
- How to port communicators to other systems (see Section B.7.4)

## B.7.1 Communicator Interface Functions

The communicator contains the main function to the debugger server, to which the interface is target-dependent. It also contains some functions that the other components of the server can call. These functions are as follows:

- **int a_client_is_connected(void)**—This function returns true if there is a client already connected to the server, and false otherwise.
- **int this_client_is_connected(void)**—This function returns true if the client from whom the communicator last received a message is connected to the server and false otherwise.
- **void set_connected(void)**—This function tells the communicator that the last packet received connected a client to the server.
- **void disconnect_client(void)**—This function tells the communicator that there is no client connected to the server.

## B.7.2 Digital UNIX Communicator

The Digital UNIX communicator is implemented in the C source file server_ main.c. This file contains the daemon's entry point (`main( )`), the main function of the user servers (`user_server_main( )` ), and the interface functions previously described.

When the daemon is started, `main( )` creates a socket and binds it to the Ladebug remote debugger connect port. It then reads packets from this port ignoring any packets that are not load requests. When it receives a load request, it checks that the client user is allowed to run remote debugger sessions on this machine using the server user name he has requested. This it does by calling to the Digital UNIX function `ruserok( )`.

If the request is valid, the daemon creates a child process (by forking). The parent process then simply continues round the packet reading loop. The child process:

1. Creates a new session.
2. Changes its group ID to the primary group ID of the requested server user.
3. Changes its login name to the server user name.
4. Changes its uid to the server user's uid.
5. Calls `user_server_main( )` with the client address and the load request packet as arguments. When `user_server_main( )` returns, the child process exits.

If, for any reason, the daemon is unable to start the user server, it then sends a load request response to the client containing an error code.

The user server function `user_server_main()` starts by creating a UDP socket for communicating with the client. It then:

1. Finds a free unprivileged UDP and binds the socket to this address.
2. Processes the load packet passed to it (by calling `ProcessPacket()`) and sends the response to the client.
3. Enters its main loop; in this loop it reads packets from the client, processes them, and sends the responses back to the server.
4. Breaks out of this loop and exits from the user server when the client disconnects from it.

One complication in the code of the communicator is that the daemon has to be able to handle the receipt of duplicate load messages. The client sends such duplicate load messages when the server's load response message is lost or does not reach the client within the client's time-out time.

To handle such duplicate load messages, a pipe is created between the daemon and each user server. When the daemon receives a duplicate load message, it uses this pipe to pass it on the appropriate user server. The user server treats this message like any other duplicate message.

### B.7.3 Evaluation Board Monitor

The evaluation boards' Ethernet driver software passes received frames to other parts of the monitor's software through call-back functions. A component of the monitor that wishes to receive frames on a UDP port calls a registration function provided by the Ethernet driver software.

The registration functions take as an argument the address of the call-back function to be called when such frames are received. When a component registers a call-back function, it can do either of the following:

- Register it for a particular port by calling `udp_register_well_known_port()`.
- Ask the ethernet driver to allocate a port by calling `udp_create_port()`. Any component of the monitor can then poll the ethernet at any time, by calling `ethernet_process_one_packet()`.

If the ethernet hardware has received a packet for any registered UDP port then the driver will call the appropriate call-back function. The call-back function called may be in a completely different component of the monitor from that which called `ethernet_process_one_packet()`. Once any call-back function has completed its processing, `ethernet_process_one_packet()` will return with a result indicating whether any packets were processed.

All packets passed to the ethernet driver must be built in fixed sized buffers provided by the driver, so that the ethernet driver never has to copy any data. The ethernet driver allocates these buffers at addresses from which the ethernet devices can send data and to which they can receive data.

To avoid the need for complex allocation algorithms, and complex error handling if buffer allocation fails, any component of the monitor can allocate a number of buffers at start-up. To maintain this buffer count the ethernet send functions always return to the caller a buffer to replace the buffer containing the packet to be sent.

On completion, the call-back functions used to receive frames must always return an ethernet buffer to the ethernet drivers. This can be, but need not be, the buffer that contained the received frame.

The debugger server does not need its own pool of buffers, since it only sends a frame immediately following the receipt of a frame. As such it handles received frames in the following steps:

1. Receive a frame through call-back function.
2. Process the frame.
3. Send a response frame in the received buffer. The send function returns a buffer (most likely a different one).
4. Exit call-back function returning the buffer returned by the send function.

The server's communicator is largely implemented in C source file `server_read_loop.c` . This contains the following code:

- The interface functions previously described.
- Two call-back functions for receiving frames. The communicator uses one of these functions to receive frames on the connection port, and the other to receiving frames on the port assigned when a client has connected. Both call-back functions pass received packet to the protocol handler and send the response to the client before returning.
- The function `enable_ladbx_msg()` . This enables the receipt of connection messages from the client by registering the connection port. For compatibility with older versions of Ladebug it also registers a second connection port with an unprivileged port number.

- The function `read_packets( )` . The server calls this function whenever it wishes to poll the ethernet for received packets. It simply calls `ethernet_process_one_packet( )` until there are no more packets to process.
- The function `data_received( )` . This is a wrapper for `read_packets( )` that is used when the reason for polling the ethernet is that there has been an interrupt. It disables interrupts before calling `read_packets( )` and restores the interrupt state once `read_packets( )` returns.
- The function `app_poll( )` . Applications that have their own interrupt handlers (and therefore disable the monitor's interrupt handler) call this function to poll the ethernet for debugger frames. It stores its return point as the debuggee's program counter and then calls `data_received( )` . The reason for setting the debuggee's program counter is that this if the server receives a stop request then it will need to know where to put a breakpoint to stop the debuggee.
- An initialization function called `ladebug_server_init_module( )` . This function places a pointer to `app_poll( )` at a standard address in memory so that an independently linked application can call it.

**Note**

The evaluation board library contains an assembler function called `ladbx_poll( )` that calls `app_poll( )` through the pointer at this address. Applications that do their own interrupt handline should call `ladbx_poll( )` frequently (for example, every time they receive an interrupt) to ensure that the debugger server receives all debugger protocol packets without excessive delay.

In addition, the file `kutil.s` contains the source of the monitor's interrupt function. The monitor only enables interrupts when an application is running. When the monitor receives any interrupt, it saves the debuggee's state and polls the Ethernet for received frames. Since the monitor will normally receive regular 1ms timer interrupts this will ensure that it receives all the client's debugger frames.

### B.7.4 Porting the Communicators to Other Systems

It should be possible to port the Digital UNIX communicator to most other versions of Digital UNIX and Digital UNIX derivatives with few changes. For operating systems that are not derived from Digital UNIX, the mechanism for starting user servers may have to be significantly modified.

In particular, many operating systems have no exact equivalent of `fork()` and instead start a new process by running a new executable file. On such a system, the communicator will have to be split into two separate executable files (one for the daemon and the other for the server). Also in such systems the new process typically does not have access to data set up by its parent before it was created, so some other mechanism will have to be used to transfer the first packet, and other data, to the user process.

The mechanism for setting the user identifier of the user server will vary widely between operating systems. Be aware that although the term daemon is a Digital UNIX term almost all operating systems have some mechanism for installing and running privileged background processes.

The socket mechanism used to read data from the network is quite widely available. If this mechanism is not available, then any other mechanism that allows the communicator to wait for the receipt of UDP packets on particular ports can be used.

If the operating system does not provide any such mechanism (for example, a real time kernel that does not include networking support), then one option is to port part or all of the networking code in the Evaluation Board Monitor to this environment. In this case, it may be easier to base your communicator upon the that in the Evaluation Board Server rather than basing it upon that in the Digital UNIX Server.

For embedded servers, few (if any) changes should be needed to the Evaluation Board's communicator. However, for many such systems you will need to rewrite the network device drivers. These are contained in the ethernet code of the Evaluation Board Monitor.

## B.8 The Protocol Handler: Interface Functions and Implementation

The protocol handler's main interface function is `ProcessPacket()` . The only argument to this function is a pointer to the packet that it is to process. As a part of its processing of the packet `ProcessPacket()` converts the request packet passed to it into a response packet. The caller must ensure that the buffer pointed to by the argument is large enough to contain any possible response packet.

The function `DumpPacket()` can also be called by the communicator. This dumps the contents of packets passed to it, if the protocol handler is compiled with tracing enabled.

The code for the packet handler is identical for the two servers. It should not need to change for other server implementations. The source code is in the files `packet-handling.c` and `packet-util.c`; `packet-handling.c` contains the function ProcessPackets(). When this function receives a packet, it:

1. Checks whether the packet is a duplicate of the previous packet, by checking whether the sequence number is the same:

   - If the packet is a duplicate, the function copies the last response sent into the packet buffer, updates the retransmission count in this packet, and returns.
   - If the packet is not a duplicate, it reads the command code. It uses the value of the command code to check that the request is legal in the debuggee's current state (for example a Get Registers request is only legal when the debuggee is stopped).
   - If the request is not legal the function puts an appropriate error code in the return value field of the packet and returns it as the response.

2. Carries out the action appropriate to the request it has received.

   This normally consists of extracting some arguments from the packet and passing them to the appropriate debugger kernel function. Where a request changes the state of the connection to the client (for example the kill command disconnects from the client), it calls the appropriate communicator interface function to inform the communicator that this has happened. In some cases, it also retrieves data from kernel functions (for example, the contents of the registers) and copies them into the packet.

3. Converts the packet into a response packet by setting the top bit of the packet's command code.

   It also sets the packet's return value. Before it returns the response packet to the communicator, it makes a copy of it so that it can be resent if the next packet duplicates the request packet.

`packet-util.c` contains utility functions for reading and writing the fields of packets and for dumping the contents of a packet. To avoid any possible alignment problems the utility functions read and write packet fields a byte at a time.

## B.9 The Debugger Kernels

This section describes:

- The debugger kernel interface functions (see Section B.9.1)
- The Digital UNIX server debugger kernel (see Section B.9.2)
- The evaluation board server (see Section B.9.3)
- Porting the debugger kernels to other systems (see Section B.9.4)

### B.9.1 The Debugger Kernel Interface Functions

The debugger kernels provide the following interface functions to the protocol handler:

- **int kload_implemented(void)**—Returns TRUE if this server can load new processes; FALSE otherwise.
- **int kload(char * name, char * argv[ ], char * standardIn, char * standardOut, char * standardError, address_value loadAddress, address_value startAddress)**—

  Loads a new process. The arguments are:

  – **name**—The name of the process to be loaded as a NULL terminated string. This will normally be the name of an executable file.

  – **argv**—The argument array. Each argument is a NULL terminated string. The argument array is terminated by an empty string (i.e one with a NULL as its first character).

  – **standardIn**—The name of the file to which standard input should be directed as a NULL terminated string. An empty string means do not redirect standard input.

  – **standardOut**—The name of the file to which standard output should be directed as a NULL terminated string. An empty string means do not redirect standard output.

  – **standardError**—The name of the file to which standard error should be directed as a NULL terminated string. An empty string means do not redirect standard error.

  – **loadAddress**—The address at which the kernel should load the executable file (or -1 if unknown).

  – **startAddress**—The address at which the kernel should start executing the debuggee. It is ignored if the load address is unknown.

The result is TRUE if successful or FALSE if the load fails. If the load is successful the processes will become the new debuggee and stop at its entry point.

- **int kconnect_implemented(void)**—Returns TRUE if this server can connect to existing processes; FALSE otherwise.
- **int kconnect(int pid)**—Connects to an existing process with the given pid. The result is TRUE if the server connects to the process and FALSE if not. If the server manages to connect to this process, then it becomes the new debuggee.
- **int kkill_possible(void)**—Checks whether the kernel can kill the current debuggee. Returns TRUE if it can and FALSE if it cannot.
- **void kkill(void)**—Kills the current process.
- **int kdetach_possible(void)**—Checks whether the kernel can detach from the current debuggee without killing it. Returns TRUE if it can and FALSE if it cannot.
- **void kdetach(void)**—Detaches from the current process.
- **int kpid(void)**—Returns the pid of the current process.
- **void kgo(void)**—Tells the debuggee to continue running until it hits a breakpoint.
- **void kstop(void)**—Tells the debuggee to stop as soon a possible.
- **int kaddressok(address_value address)**—Checks whether an address in the debuggee is readable. Returns TRUE if it is and FALSE if it is not.
- **ul kcexamine(address_value address)**—Gets the data at address. If address points at a breakpoint or there is a breakpoint within 8 bytes of address the data that was at the address before the breakpoint was inserted is returned.
- **int kcdeposit(address_value *address, ul value)**— puts value at address, updating the saved data for breakpoints if necessary.
- **void kstep(void)**—Steps one instruction.
- **address_value kpc(void)**—Returns the debuggee's program counter.
- **void ksetpc(address_value address)**—Sets the contents of the debuggee's program counter.
- **register_value kregister(int reg)**—Returns the contents of the debuggee's register number reg. For Alpha targets the register number of fixed point register n is n: and that of floating point register n is n+32.

- **void ksetreg(int reg, register_value value)**— sets the contents of register number reg with value.
- **short int kbreak(address_value address)**—Sets a breakpoint at address. Returns the result that should be returned to the client.
- **int kremovebreak(address_value address)**—Removes the breakpoint at address. Returns the result that should be returned to the client.
- **int kpoll(void)**—Returns the state of the debuggee. This is one of:
  - **PROCESS_STATE_PROCESS_RUNNING**—The debuggee is running.
  - **PROCESS_STATE_PROCESS_AT_BREAK** —The debuggee is stopped at a breakpoint or after a performing a single step.
  - **PROCESS_STATE_PROCESS_SUSPENDED** —The debuggee is stopped somewhere else due to a signal, trap or exception.
  - **PROCESS_STATE_PROCESS_TERMINATED** —The debuggee has exited.

### B.9.2 Digital UNIX Server Debugger Kernel

In the Digital UNIX server, the debugger kernel is implemented using the `ptrace( )` function. This is a Digital UNIX function that allows a parent process to examine and control its children. Since `ptrace( )` can only be used to debugger child processes the Digital UNIX debugger kernel only supports the loading of new processes and not connection to existing processes.

Since this means that the debuggee always runs as a child of the server, and the server is killed when the client disconnects, the Digital UNIX server does not support disconnecting from a debuggee without killing it.

When `kload( )` loads a new debuggee, it does so using the Digital UNIX functions `fork( )` and `exec( )` . `kload( )` creates the new process using the `fork( )` function. This new child process:

1. Makes a `ptrace( )` call to allow its parent to control it using `ptrace( )` calls. This also sets up a breakpoint on executing new images.
2. Opens the standard input, output, and error files. If it is unable to open any of these it terminates.
3. Loads and executes the debuggee by calling `exec( )` . When it reaches the debuggee's entry point, it will stop and its parent will receive a SIGCHLD signal.

The parent process meanwhile waits for a signal from the child. When it receives a signal it, checks that the debuggee has stopped at a breakpoint (rather than, for example, having exited). If it has then the kernel checks whether the debuggee uses shared libraries.

If the debuggee does use shared libraries the server tells it to continue (through a `ptrace()` call) and waits for it to stop once more.

**Note**

> This is done because starting a program that uses shared libraries on Digital UNIX involves executing two new images. The first is a special program loader and the second is the image of the program itself. The debuggee cannot be accessed by `ptrace()` until the child process stops for the second time.

Once the debuggee has been started the server stores the state of the debuggee in the variable child_state. The Digital UNIX kernel inserts breakpoints by using `ptrace()` to write a breakpoint PAL call to the address of the breakpoint. On Alpha Digital UNIX `ptrace()` always reads and writes 8-byte (2 instruction) quantities, so the kernel has to insert and remove breakpoints through a read, modify, write sequence.

The kernel implements stop request (`function kstop()` ) by sending a SIGINT signal to the debuggee. This will stop the debuggee unless it has disabled receiving SIGINT signals. `kkill()` uses the same technique to stop the debuggee before killing it.

When `kgo()` is called, it first checks whether there is a breakpoint at the current program counter. If there is a breakpoint there, then the kernel executes the original instruction at this location using the internal function `kstepoverbreak()` . This function temporarily puts the instruction back into the code and then uses `ptrace()` with the PT_STEP function code to execute this instruction. Once it has executed the instruction it restores the breakpoint.

When `kstepoverbreak()` returns `kgo()` calls the internal function `kasyncwait()` to set up `kstopped()` as a signal handler for the SIGCHLD signal. It then calls `ptrace()` with the PT_CONTINUE function code. This tells the debuggee to continue from its current program counter.

When a running debuggee stops for any reason, the server will receive a SIGCHLD signal. This will cause `kstopped()` to be called. `kstopped()` checks why the debuggee has stopped and sets the child_state appropriately. If the debuggee has stopped as a result of a breakpoint PAL call the program counter will point to the instruction after the breakpoint. Under these circumstances, `kstopped()` will move the program counter back one instruction to point at the breakpointed address.

When the kernel reads memory, it uses the breakpoint table functions to check whether there is a breakpoint on either of the longwords it is reading. If there is such a breakpoint, it reads the data for that longword from the breakpoint table rather than from the debuggee's memory. Similarly, when the kernel is asked to write to memory, it will update breakpoint table entries if necessary and will not overwrite breakpoint PAL calls.

The mapping of register to register number used by `ptrace()` is the same as that used by the kernel interface. This means that `kregister()` and `ksetreg()` translate very directly into `ptrace()` calls.

### B.9.3 Evaluation Board Server

The evaluation board server's debugger kernel is implemented by directly reading from and writing to memory. It runs in the same environment as the debuggee, with the same mapping of virtual to physical addresses.

As such, there is no distinction between its memory and the debuggee's memory. This means that it can satisfy requests to read or write the debuggee's memory by simply reading or writing its own virtual memory.

The evaluation board kernel implements breakpoints through a special additional PAL call, DBGSTOP. The monitor's PAL code provides this additional PAL call. It functionally is identical to the standard Digital UNIX breakpoint PAL call except that its system entry address can be set up independently by passing a different function code value to `wrest()`. This allows the monitor to set breakpoints even in applications (for example operating systems) that do their own breakpoint handling using the standard breakpoint PAL call.

The evaluation boards' debugger kernel is implemented in the C source file `kernel.c` and the assembler source file `kutil.s`. The functions in these files are also used to implement the low level debugger commands provided by the monitor on its dumb terminal interface. `kernel.c` contains the main body of the debugger kernel, including all the interface functions previously listed. `kutil.s` contains the system entry points for interrupts, traps and breakpoints; and functions that provide a C interface to various PAL calls.

#### B.9.3.1 Initialization

`kernel.c` contains three functions that are used to initialize the debugger kernel:

- `kstart( )` is called when the system starts up. It simply initializes some of `kernel.c`'s static variables and ensures that interrupts are disabled.
- `kinitpalentry( )` is called before the monitor runs any application. It reinitializes the PAL system entry points by calling the assembler function `kutilinitbreaks( )`. Once a debuggee has been started, and until it completes, monitor code will only be executed when it is called directly or indirectly from one of these system entry points. `kutilinitbreaks( )` defines the system entry point for interrupts to point to the monitor's interrupt function, and the system entry point for DBGSTOP point to the monitor's low level breakpoint function. All other system entry points are defined to point to the monitor's trap function.
- `kenableserver( )` is called to switch to remote debug mode. It is called when the user issues the `ladbx` command at the monitor's dumb terminal. At this point the user should have already loaded and started the debuggee using the monitor's dumb terminal commands, and it should be stopped at a breakpoint. It sets remote debugger mode, calls `enable_ladbx_msg( )` to enable the receipt of debugger messages by the server and then waits for such messages by calling `kwaitforcontinue( )`.

  Once the debuggee has been started the state of the debuggee is always in the variable child_state. `kpoll( )` simply reads this variable.

#### B.9.3.2 Setting Breakpoints

The kernel sets breakpoints by saving the original instruction in the breakpoint table and inserted by writing the DBGSTOP instruction to the location at which a breakpoint is required. To simplify other memory access in the debugger monitor the kernel does not write DBGSTOP instructions into memory until just before the program is allowed to run, and replaces them with the original instructions as soon as the program stops.

The function `kinstall_breakpoints( )` writes DBGSTOP instructions for all current breakpoints to memory, and the `restore_breakpoint_instrs( )` internal function restores the original instructions at these locations whenever the debuggee stops. Because any modification to the debuggee's memory can alter its instruction stream, the kernel follows all writes to the debuggee's memory with instruction barrier PAL calls.

#### B.9.3.3 Hitting a Breakpoint or an Exception

When the debuggee hits a breakpoint (i.e., executes a DBGSTOP PAL call), the PAL code calls the monitor's assembler breakpoint function (`dbgentry( )` ), which:

1. Saves processor's complete register set (including the program counter and the processor status) in a static area
2. Calls `kreenter( )` , which:
   a. Removes any temporary breakpoints (used for single stepping and to implement stop requests).
   b. Calls through a function pointer the kernel's current breakpoint continuation function.

Unless the debuggee has just single stepped or processed a stop request this function will be `katbpt( )` . `katbpt( )` steps the saved program counter back one instruction so that it:

1. Points at the actual breakpoint (rather than at the instruction following it).
2. Calls `kwaitforcontinue( )` , which:
   a. Restores the original instructions at all the breakpoints.
   b. Sets a flag to indicate that the debuggee is stopped.
   c. Listens for either debugger packets or commands by repeatedly calling either `read_packets( )` or `user_main( )` .

   `kwaitforcontinue( )` will stay in this loop until some other kernel function clears the stopped flag.

The handling of exceptions is similar to the handling of breakpoints. All the unused system entry points are initially set up to point to dbgtrap. This sets a flag (in a register that the PAL code has already saved) to indicate that the server was reentered as a result of an exception and then jumps to `dbgentry2`.

This is an alternative entry point to the function `dbgentry( )` . `dbgentry( )` saves the processor's registers, as previously described, but then, instead of calling `kreenter( )` , calls `ktrap( )` . `ktrap( )` removes any temporary breakpoints, then sets `child_state` appropriately and then calls `kwaitforcontinue( )` .

### B.9.3.4 Receiving and Processing Commands

A command can be received as a result of:

- The user typing in a monitor command when the monitor is in local debugger mode.
- A packet being received by the interrupt routine, or by the ethernet poll routine called by the debuggee, while the debuggee is running. Be aware that Ladebug will only send the server stop, disconnect, or poll commands while the debuggee is running.
- A packet being received (through the read_packets( ) call) while the monitor is in its breakpoint loop.

When a command is received, either the command processor or the protocol handler calls the appropriate kernel function. The functions that can be called are the previously listed interface functions. Table B–2 explains the behavior.

**Table B–2 kernel Functions**

| kernel Function | Action |
|---|---|
| `kload( )`<br>`kload_implemented( )` | Always return FALSE |
| `kconnect_implemented( )` | Returns TRUE |
| `kconnect( )` | Always returns TRUE, does nothing else |
| `kkill_possible( )` | Always returns FALSE |
| `kkill( )` | Does nothing |
| `kdisconnect_possible( )` | Always returns TRUE |
| `kdisconnect( )` | Does nothing |
| `kpid( )` | Always returns 0 |
| `kgo( )` | Checks whether the debuggee is stopped at a breakpoint. If it is, it uses `ksetstepbreak( )` to set the temporary breakpoints so that the debuggee stop again after executing one instruction. It also sets the breakpoint continuation function to be `ksteppedoverbreak( )` . If the debuggee is not at a breakpoint, `kgo( )` places breakpoint instructions (DBGSTOP PAL calls) at all the breakpoints and sets the breakpoint continuation function to be `katbpt( )` . Then, whether or not the debuggee was at a breakpoint, it clears the stopped flag so that the debuggee will continue the next time `kwaitforcontinue( )` checks the flag. |

(continued on next page)

**Table B–2 (Cont.) kernel Functions**

| kernel Function | Action |
|---|---|
| `kstop( )` | Checks whether the debuggee is still running or stopped. If the debuggee is stopped, `kstop( )` does nothing. If the debuggee is running, it places a temporary breakpoint at the current instruction. |
| `kaddressok( )` | Returns TRUE if the address is quadword aligned and FALSE otherwise. |
| `kcexamine( )` | Reads the requested location. It does not need to check the breakpoint table because it is only called when the debuggee is stopped. |
| `kcdeposit( )` | Writes to the requested location. |
| `kstep( )` | Uses `ksetstepbreak( )` to set up temporary breakpoints everywhere the program counter can be after executing the next instruction. This requires a maximum of two temporary breakpoints since `ksetstepbreak( )` can work out the destination of a jump instruction by reading the instruction's argument register. It also sets the breakpoint continuation function to be `katbpt` and clears the stopped flag. |
| `kpc( )` | Returns the saved program counter. |
| `ksetpc( )` | Modifies the saved program counter. |
| `kregister( )` | Returns the value of the appropriate entry in the saved register array. |
| `ksetreg( )` | Sets the value of the appropriate entry in the saved register array. |
| `kbreak( )` | Calls `bptinsert( )` . The kernel does not write to the debuggee's memory until the debuggee about to be run or resumed. |
| `kremovebreak( )` | Calls `bptremove( )` . |
| `kpoll( )` | Returns the value of `child_state`. |

### B.9.3.5 Continuing from a Breakpoint or Exception

When `kwaitforcontinue( )` sees that the stopped flag is clear, it returns (through a number of intermediate functions) to `bptentry( )` . This restores the processor registers and then calls the PAL RTI function to return to the debuggee.

If the debuggee was continuing from a (permanent) breakpoint as a result of a kgo() call, it will hit a new (temporary) breakpoint after executing one instruction. The state will be saved as it would be with a permanent breakpoint but the breakpoint continuation function called will be ksteppedoverbreak( ) . This backs up the program counter 1 instruction, places DBGSTOP PAL calls at all the breakpoints in the breakpoint table, and then once again returns to bptentry( ) to resume the debuggee.

The debuggee will now run until it is stopped by hitting a further permanent breakpoint or by an exception, or by a stop command.

#### B.9.3.6 Interrupt Handling

The assembler source file kutil.s contains the function dbgint( ) . This is the monitor's system entry point for interrupts. On receiving any interrupt the monitor save the previous state and call data_received( ) to tell the communicator that the ethernet device may have received data and that it should poll the ethernet driver.

The one complication in dbgint( ) is that if the server receives a Stop Request packet, then the debugger kernel will need to know the debuggee's current program counter. This is not necessarily the program counter saved by the PAL code because the interrupt routine can itself be interrupted (and therefore be called recursively).

The global variable containing the saved program counter is checked for a nonzero value. A value of zero is used to indicate that it is not in use. If it is already set, it is not reset but data_received( ) is called.

If the program counter has not already been saved in this global variable, the value that was saved on the stack by the PAL code is examined. If it is within dbgint( ) , then this is a recursive call to dbgint( ) with the second interrupt having happened before the first call to dbgint( ) saved the program counter. In these circumstances, there is no need to call data_received( ) since it will be called by the outer call to dbgint( ) . Otherwise, this value is saved as the debuggee's program counter and data_received( ) is called.

This procedure requires that the code that saves the value of the program counter should be in the function dbgint( ) and not within another function called by dbgint( ) .

The kernel also contains a function, knullipl( ) , that clears an interrupt. On the EB64 version of the kernel, it simply writes two commands to the 82C59 (the interrupt controller used on the EB64). This function will clearly have to be rewritten for target systems that use different interrupt controllers.

### B.9.4 Porting the Debugger Kernels to Other Systems

Few, if any, changes should be needed to port the Digital UNIX debugger kernel to other Digital UNIX like operating systems. The operating system functions used in the Digital UNIX debugger kernel seem to be available in all Digital UNIX dialects.

A problem that may arise is that in some Digital UNIX dialects, when the debuggee stops at a breakpoint the program counter, it may point to the actual breakpoint instruction rather than the instruction following the breakpoint. For some of the newer dialects of Digital UNIX, a server with greater functionality (in particular the additional ability to connect to existing processes) could be implemented by rewriting the debugger kernel to use the `/proc` debugger interface.

Porting the server to other operating systems will involve replacing the `ptrace( )`, `fork( )` and `exec( )` calls with the equivalent calls (if they exist) for the target operating system. Assuming it is possible to read and write a subprocess's memory and registers, this should not be difficult.

On operating systems where this is not possible you may have to link some low level debugger functions into the debuggee and communicate between these functions and the kernel through shared memory. On such operating systems, there is also a need for a mechanism for detecting that the debuggee has stopped at a breakpoint. How this is done will vary widely between operating systems.

Few changes are likely to be needed to port the Evaluation Board Server to other embedded systems that use the Digital UNIX PAL code interface. The changes that will often be needed are as follows:

- If the system uses a different interrupt controller, then `knullipl` will have to be rewritten.
- If the system's PAL code does not implement the DBGSTOP PAL call, then BPT PAL call should be used in its place. The code, used in the initialization functions to set up system entry points, will have to be altered to reflect this change.

If some other PAL code interface is used, then this will probably require changes in how breakpoints are set and how the server's entry points are called when the debuggee hits a breakpoint or receives an interrupt. It may also alter how much of the debuggee's state is saved by the PAL code before the server's entry points are called from the PAL code, and the value of the program counter that is passed to the server on reaching a breakpoint.

The most common problems that have arisen when modifying the code of the debugger kernels are as follows:

- Errors when fixing the program counter when the server is reentered.
- Missing instruction barrier PAL calls. These are often difficult to find since the problems they cause depend on the state of the processor's internal caches. A typical symptom of this problem is that the debuggee does not stop at a breakpoint.
- Errors when saving or restoring the state of the debuggee. In particular, errors when working out whether the debuggee is using the kernel or using the user stack pointer.

## B.10 The Breakpoint Table Handler: Interface Functions and Implementation

The breakpoint table handler provides the following interface functions:

- **void bptinitialize(void)**—Initializes the breakpoint table.
- **int bptinsert(address_value addr, instruction_value * savedinstr)**—Adds a breakpoint to the table. The address is addr and the instruction to be saved is savedinstr. Returns SUCCESS if successful and a negative error code otherwise.
- **int bptremove(address_value address, instruction_value * savedinstr)**—Removes a breakpoint from the table. If successful, it returns SUCCESS as its result and returns the saved instruction in `*savedinst` . If it fails to remove the breakpoint, it returns a negative error code as its result.
- **int bptgetn(int n, address_value*address, instruction_value * savedinstr)**—Finds a breakpoint by breakpoint number. If it finds the breakpoint it returns SUCCESS as its result. It also returns the address of the breakpoint in `*address` and the saved instruction in `*savedinst` . If it fails to find the breakpoint it returns a negative error code as its result.
- **int bptgeta(address_value address, instruction_value * savedinstr)**—Finds a breakpoint by address. If it finds the breakpoint, it returns SUCCESS as its result and returns the saved instruction in `*savedinst` . If it fails to find the breakpoint, it returns a negative error code as its result.
- **int bptisbreakat(address_value address)**—Returns true if there is a breakpoint at address.

- **int bptfull(void)**—Returns true if the breakpoint table is full.

The code for the breakpoint table handler is identical for the two servers. It should not need to change for other server implementations. The source code is in `bptable.c` . The table is implemented as 3 arrays of 100 entries each. The breakpoint number of a breakpoint is used as an index into these arrays. The three arrays are:

- In-use array
- Breakpoint addresses array
- Saved instructions array

New entries are inserted in the first available entry and entries are found by a linear search.

## B.11 Ladebug Remote Debugger Protocol

The Ladebug Remote Debugger Protocol is a request/response protocol running over UDP. The debugger client (Ladebug) initiates all transactions sending a request to the server. On receiving the request, the server acts upon the request and sends a response.

If the client does not receive a response within a time-out, it repeats the request (with an indication that the message is a duplicate). The time-out will vary between a tenth of a second and 10 seconds depending on how long it took to get responses to previous requests.

If the client does not receive a response, after a number of attempts and with increasing retry time-outs, it assumes that the communication path to the server has failed. The server never sends any messages except in response to messages received from the client.

Section B.11.1.1 through Section B.12 describe more about the Ladebug remote debugger protocol:

- Messages and formats
- Order of messages
- How to recover from packet loss
- The transport layer

## B.11.1 Messages and Message Formats

This section describes the Ladebug Remote Debugger Protocol messages and the format of each message.

### B.11.1.1 Message Headers

Table B–3 shows header names, byte numbers, format, and contents of the message headers. Section B.11.1.2 shows the possible values of the messages.

**Table B–3 Header Format**

| Name | Byte Number | Format | Content |
|---|---|---|---|
| Protocol Version | 0 | Integer | Should be 2 |
| Retransmit Count | 1 | Integer | In requests, 0 the first time a packet is transmitted: each retransmission of packet increments by a one.<br><br>In responses, The retransmit count of the request. |
| Command code | 2 to 3 | Integer in network order, most significant byte first[1] | Identifies the type of request or response. |
| Sequence Number | 4 to 7 | Integer in network order | Identifies the message. |
| Process ID | 8 to 11 | Integer in target machine order, least significant byte first for Alpha targets | Identifies the process being debugged. The value is not defined in load request messages. |
| Return value | 12 to 16 | Integer in network order | Ignored in requests. In replies tells the client whether the requested action was successful, and if not why not |

.

[1]The protocol sends multibyte integer fields whose meaning is independent of the target architecture in conventional network order (i.e most significant byte first). Examples of such fields are the command code or byte counts. Multibyte integer fields that can only be interpreted with knowledge of the target architecture, such as addresses or register values, are sent in target machine order. For Alpha targets this means that such fields are sent least significant byte first.

### B.11.1.2 Message Values

Table B–4 explains the values returned by messages.

**Table B–4 Message Table**

| Value | Message | Explanation |
|---|---|---|
| 0 | OK | Request succeeded |
| 1 | Bad process ID | The process ID of the message is not that of the debuggee, or, in the case of Connect to Process, the server could not connect to that process. |
| 2 | No resources | The server did not have the resources to carry out the request. |
| 3 | Not connected | The server is not connected to a debuggee. The request requires that it should be. |
| 4 | Not stopped | The debuggee is running. The request can only be carried out with the debuggee stopped. |
| 5 | Bad address | The address given in the request is bad. The precise meaning of this varies between the different types of responses that can give this return value. |
| 6 | Not implemented | The server does not implement this request. |
| 7 | Bad load name | See Section B.11.1.4 |
| 8 | Already connected | The server is already debugging the requested debuggee. |
| 9 | Cannot disconnect from process | See Section B.11.1.8 |
| 10 | Cannot kill process | See Section B.11.1.10 |
| 11 | Cannot step | See section Section B.11.1.12 |

### B.11.1.3 Load Process Request and Response

The load process request is a request to the server to load a new process and to start a new debugger session. They are transmitted in the request in the order shown with no unused bytes between the fields. Section B.11.1.4 describes the possible responses to a load process request.

Table B–5 shows the fields of the load process request:

**Table B–5 Fields of the Load Process Request Message**

| Name | Length | Format | Contents |
|---|---|---|---|
| Header | 16 bytes | See Table B–3 | See Table B–3 |
| Client User Name | Variable | Null terminated character string | Name of the user of the client on the host. This can be used by the server to check that the client is allowed to load the requested process. |
| Server User Name | Variable | Null terminated character string | User name of user to run the process on the target. This will be ignored by some servers. |
| Program Name | Variable | Null terminated character string | Name of program to be loaded. The form and interpretation of this name will vary between servers. |
| Number of arguments | 1 byte | Integer | Count of program argument fields |
| Arguments | Variable | Variable number of null terminated strings. The number of arguments field gives the number of strings. | Arguments to be passed to the loaded process. May be ignored by some servers. |
| Standard input | Variable | Null terminated string | File name of a file to which standard input is to be redirected. An empty string (just a 0 byte) indicates no redirection: otherwise the interpretation of the file name is server dependent. |
| Standard output | Variable | Null terminated character string | Name of file to which standard output is to be redirected. This file name is interpreted in the same way as the standard input file name. |
| Standard error | Variable | Null terminated character string | Name of file to which standard error is to be redirected. This file name is interpreted in the same way as the standard input file name. |

#### B.11.1.4 Responses to the Load Process Request

The command code for a Load Process request is 1. The server should ignore the PID received in a Load Process request.

The fields of a Load Process response are:

- The headers described in Table B–5, with the process ID field set to the process ID of the loaded process.
- A 1-byte processor type. At present the only defined type is 0, meaning an Alpha processor.
- A 1-byte process type. At present the only defined process types are:
  - Kernel process
  - User process
- Possible failure reasons are:
  - No resources—The server does not have the resources it requires to create a new process.
  - Bad load name—The server did not understand the name of the new process, or could not load the named process.
  - Not implemented—This server does not implement the load request.

The command code of a load response is 0x8001.

#### B.11.1.5 Connect to Process Request and Response

A connect request message requests that the server should start a debugger session by connecting to an existing process on the host. The fields of a connect request are:

- Packet header, as previously described. The process ID field identifies the process to which the server should connect.
- Client user name, as in load requests.
- Server user name, format as in load requests: but only used to check that the client has the correct privileges to debug the requested process.

The command code of a connect request is 2.

The format of a connect response is identical to that of a load response. Possible failure reasons are:

- Bad process ID—The server did not understand the process ID or could not connect to that process.

- No resources—The server does not have the resources it requires to create or debug a new process. A server that can only debug one process at a time will return this error if it is already debugging a process.
- Already connected—There is already a client connected to the same process.
- Not implemented—This server does not implement the connect request.

The command code of a connect response is 0x8002

#### B.11.1.6 Connect to Process Insist Request and Response

A connect insist request message requests that the server should take over debugging a process to which there may already be a server connected. The formats of connect insist requests and responses are differ from those of connect requests and responses only in the command codes. A Connect to Process Insist request has a command code of 3 and its response has a command code of 0x8003. A server should only return the Already Connected failure reason if it could not terminate the old debugger session.

#### B.11.1.7 Probe Process Request and Response

The Probe Process request asks the server what the state of the debuggee is. It contains no fields other than the standard header. Its command code is 0x81.

The Probe Process response returns the state of the debuggee. Following the standard header (at byte 16) it contains a 1-byte integer field giving the state of the debuggee. Possible values are:

- The debuggee is running.
- The debuggee has stopped at a breakpoint or following a single step.
- The debuggee has stopped unexpectedly. This might, for example, be as a result of it receiving a signal or executing a bad instruction. This value should also be returned if the debuggee has stopped as a result of a Stop Process request.
- The debuggee has exited.

Possible failure reasons are:

- Bad process ID
- Not connected

The command code of Probe Process response is 0x8081.

### B.11.1.8 Disconnect from Process Request and Response

The Disconnect from Process request asks the server to disconnect from both the current debuggee and the client. It will only succeed if the server can disconnect from the debuggee without killing it, or if the debuggee is already dead.

The effect on breakpoints of disconnecting from a process may vary between servers. In particular, the protocol does not define whether disconnecting from a stopped process will allow it to run on, or what happens if the processes reaches a breakpoint after the server has disconnected from it.

The request contains no fields other than the message header. Its command code is 0x82.

The Disconnect from Process response contains no fields other than the message header. Possible failure reasons are:

- Bad process ID
- Not connected
- Cannot Disconnect from Process—The server cannot disconnect from the debuggee without killing it

If the server cannot disconnect from the debuggee it will remain connected to the client and to the debuggee. The command code of Disconnect from Process response is 0x8082.

### B.11.1.9 Stop Process Request and Response

The Stop Process request asks the server to stop a running debuggee as soon as possible. It contains no fields other than the message header. Its command code is 0x83.

The Stop Process response contains no fields other than the message header. Possible failure reasons are:

- Bad process ID
- Not connected
- Not implemented—The server cannot stop running processes asynchronously

The command code of Stop Process response is 0x8083.

#### B.11.1.10 Kill Process Request and Response

The Kill Process request asks the server to kill the current debuggee and disconnect from the client. It will only succeed if the server can kill the debuggee, or if the debuggee is already dead. The request contains no fields other than the message header. Its command code is 0x84.

The Kill Process response contains no fields other than the message header. Possible failure reasons are:

- Bad process ID
- Not connected
- Cannot Kill Process—The server cannot kill the debuggee

If the server cannot kill the debuggee it will remain connected to the client and to the debuggee. The command code of Kill Process response is 0x8084.

#### B.11.1.11 Continue Process Request and Response

The Continue Process request asks the server to make the debuggee to run on until it hits a breakpoint, terminates, is stopped by the server acting on a Stop Process request, or stops for some other reason (e.g. executing a trap or exception). It contains no fields other than the message header. Its command code is 0xA1.

The Continue Process response contains no fields other than the message header. Possible failure reasons are:

- Bad process ID
- Not connected
- Not stopped

If the debuggee is terminated the request will succeed: but its effect is undefined. The command code of a Continue Process response is 0x80A1.

#### B.11.1.12 Step Request and Response

The Step request asks the server to make the debuggee execute one instruction. It contains no fields other than the message header. Its command code is 0xA2.

The Step response contains no fields other than the message header. Possible failure reasons are:

- Bad process ID
- Not connected
- Not stopped

- Cannot step—The server cannot single step the debuggee at this point. This might occur if, for example, the server would have to set a temporary breakpoint on an address to which it cannot write to single step the debuggee.

If the debuggee is terminated the request may succeed: but its effect is undefined. The command code of a Step response is 0x80A2.

#### B.11.1.13 Set Breakpoint Request and Response

The Set Breakpoint request asks the server to set a breakpoint in the code of the debuggee. Although a server is not required to be able to set a breakpoint on any particular address to be useful it must be able set breakpoint on a significant portion of the instructions of the debuggee.

The effect of setting a breakpoint on anything other than an instruction of the debuggee is not defined. Furthermore, the effect of setting a breakpoint on an instruction that the debuggee modifies or reads as data is not defined.

The fields of a Set Breakpoint request are:

- Message header
- Address of breakpoint—This is an 8-byte value sent in target machine byte order

The command code of a Set Breakpoint request is 0xA3.

The Set Breakpoint response contains no fields other than the message header. Possible failure reasons are:

- Bad process ID
- Not connected
- Not stopped
- Bad address—The server cannot set a breakpoint at the given address. The server may also return this failure reason if there is already a breakpoint at that address, although some servers can allow multiple breakpoints at the same address.
- No resources—The server did not have sufficient resources to add another breakpoint. The server may, for example, have a fixed sized breakpoint table that is full.

The command code of a Set Breakpoint response is 0x80A2.

#### B.11.1.14 Clear Breakpoint Request and Response

The Clear Breakpoint request asks the server to remove a breakpoint from the debuggee. Its fields are:

- Message header
- Address of breakpoint—This is an 8-byte value sent in target machine byte order

The command code of a Clear Breakpoint request is 0xA4.

The Clear Breakpoint response contains no fields other than the message header. Possible failure reasons are:

- Bad process ID
- Not connected
- Not stopped
- Bad address—The address is not the address of a breakpoint

The command code of a Clear Breakpoint response is 0x80A4.

#### B.11.1.15 Get Next Breakpoint Request and Response

Using Get Next Breakpoint requests the client can get a complete list of the breakpoints known to the server that affect the current debuggee. In some servers this will include breakpoints set in the debuggee by previous remote debugger sessions or through an alternative interface. For example, in the evaluation board server it includes breakpoints set by previous debugger sessions and those set through the local debugger interface.

To get a complete list of breakpoints the client should start by sending a Get Next Breakpoint with a breakpoint address of zero. It should then send further Get Next Breakpoint requests each containing the address returned by the previous Get Next Breakpoint response. A server that receives this sequence of requests with no other requests intervening must return each breakpoint it knows about precisely once. The protocol does not define the order in which the server will return the breakpoints it knows about.

The fields of a Get Next Breakpoint request are:

- Message header
- Address of previous breakpoint or zero if this is a request for the first breakpoint in the list. This is an 8-byte value sent in target machine byte order.

**Note**

A server must not set a breakpoint on address zero. This is not regarded as a serious restriction.

---

The command code of a Get Next Breakpoint request is 0xA5.

The fields of a Get Next Breakpoint response are:

- Message header
- Address of breakpoint or zero if there are no more breakpoints to be returned. This is an 8-byte value sent in target machine byte order

Possible failure reasons are:

- Bad process ID
- Not connected
- Not stopped
- Bad address—The address in the request is neither zero nor the address of a breakpoint

The command code of a Get Next Breakpoint response is 0x80A5.

#### B.11.1.16 Get Registers Request and Response

The Get Registers request asks the server to send the client the contents of all the debuggee's registers and pseudo registers. It contains no fields other than the message header. Its command code 0xA6.

The fields of the Get Registers response are:

- Message header
- Register list—For an Alpha target this is an array of 65 entries. Each entry is 8 bytes long. The value in each entry is in target machine byte order:
  - Entries 0 to 31 respectively contain the contents of fixed point registers 0 to 31.
  - Entries 32 to 63 respectively contain the contents of floating point registers 0 to 31.
  - Entry 65 contains the debuggee's current program counter.

Possible failure reasons are:

- Bad process ID

- Not connected
- Not stopped

The command code of a Get Registers response is 0x80A6.

#### B.11.1.17 Set Registers Request and Response

The Set Registers request asks the server to all the debuggee's registers and pseudo registers: including the debuggee's program counter. The request succeeds even if it is unable to change the values of some of the registers. Its fields are:

- Message header
- Register list—For an Alpha target this is an array of 65 entries. Each entry is 8 bytes long. The value in each entry is in target machine byte order:
  - Entries 0 to 31 respectively contain the contents of fixed point registers 0 to 31.
  - Entries 32 to 63 respectively contain the contents of floating point registers 0 to 31.
  - Entry 65 contains the debuggee's current program counter.

The command code of a Set Registers request is 0xA7.

A Set Registers response contains no fields other than the message header. Possible failure reasons are:

- Bad process ID
- Not connected
- Not stopped

The command code of a Set Registers response is 0x80A7.

#### B.11.1.18 Read Request and Response

A Read request asks the server to read a portion of the debuggee's memory. Its fields are, in order:

- Message header
- Start address—The address of the first byte to read. This is an 8-byte value sent in target machine byte order.
- Number of bytes to read—This is a 4-byte value in sent network order.

Its command code is 0xA8.

The fields of a Read response are, in order:

- Message header
- Start address—The address of the first byte read. This is an 8-byte value sent in target machine byte order.
- Number of bytes read—This is a 4-byte value in sent network order.
- Data read—This is a sequence of bytes. Its length is given by the previous field.

Possible failure reasons are:

- Bad process ID
- Not connected
- Not stopped
- Bad address—The server could not read data from the given address range

The command code of a Read response is 0x80A8.

#### B.11.1.19 Write Request and Response

A Write request asks the server to overwrite a portion of the debuggee's memory. Its fields are, in order:

- Message header
- Start address—The address of the first byte to read. This is an 8-byte value sent in target machine byte order.
- Number of bytes to write—This is a 4-byte value in sent network order.
- Data to write—This is a sequence of bytes. Its length is given by value of the previous field.

Its command code is 0xA9.

A Write response contains no fields other than the message header. Possible failure reasons are:

- Bad process ID
- Not connected
- Not stopped
- Bad address—The server could not write data to some address in the given address range.

The command code of a Write response is 0x80A9.

## B.11.2 Order of Messages

A server can be modeled as a single control thread plus a debuggee thread for each debuggee. Each thread is uniquely identified by its UDP port number. Once a client has sent a message to a thread, it cannot send further messages to that thread (other than copies of the original message) until it receives a response.

A server (control or debuggee) thread can only send responses to the requests it receives. Servers threads are expected to respond promptly to all requests they receive. A server thread must never send more than one response to each message it receives. If it receives a duplicate request it must send a copy of its original response, with an updated retransmission count, without acting a second time on the request.

The only messages a client can send to a control thread are Connect to Process, Connect to Process Insist, and Load Process requests. A positive response to either of these requests identifies a new debuggee thread that the client should use for debugging the new debuggee.

**Note**

None of the previous information suggests that a server must be implemented as a multithreaded program. A server that can only debug one process at a time can send replies containing no resource failure codes to clients that attempt to connect to it when it is already debugging a process.

A debuggee thread can be in either running or stopped state. Initially a debuggee thread is in running state. In running state it will accept the following requests:

- Probe Process
- Stop Process
- Disconnect from Process
- Kill Process

The client must not send any other requests to a debuggee thread when it is in running state. A debuggee thread will enter stopped state from running state whenever the debuggee stops. The client can discover the state of a debuggee thread by sending it a probe request. Any debuggee state other than running indicates that the debuggee thread is in stopped state. When it is in stopped state, the client can send the debuggee thread any request except Connect to Process, Connect to Process Insist, or Load Process.

A debuggee thread will return from stopped state to running state when it receives a Continue or Step request that it can act upon.

A debuggee thread will exit immediately after sending a positive Disconnect from Process or Kill Process response. It can also exit at any other time, either through some external cause, or as a result of some other client taking over the debuggee using a Connect to Process Insist request. Once a debuggee thread has exited the server will either ignore requests sent to it or send responses with not connected failure codes.

### B.11.3 Recovering from Packet Loss

The standard packet header contains two fields that are used to recover from packet loss:

- The sequence number
- The retransmission count

The sequence number is used to distinguish between different messages. The retransmission count is used to distinguish between copies of the same message. A client should give every message it sends to a particular server thread a different sequence number.

To avoid confusion between different clients started by the same user on the same host, it should also attempt to give load and connect requests sequence numbers that will not be used by other clients. One way to do this would be to base the sequence number upon the time at which the message is sent.

The first time it sends a message it will give it a retransmission count of 0. If it does not receive a response to a request within a reasonable time, it will increment the retransmission count and repeat the message. If after a number of attempts it has still not received a response with the same sequence number and retransmission count as the last message it sent it will assume that the server thread had exited or communications link has failed.

**Note**

The best values for "reasonable time" and "a number of attempts" are still being under investigation. At present, Ladebug starts off by waiting 1.6 seconds. Each time it receives a response within its time-out it halves its time-out down to a minimum of 1/10 of a second.

Each time it fails to receive a response it doubles its time-out up to a maximum of 12.8 seconds. It makes a maximum of 8 attempts at sending any message.

On receiving a duplicate packet, the server should copy the new retransmission count into its original response and send this updated response to the client.

The server cannot normally detect communication failure and will wait indefinitely for messages from a client.

## B.12 Transport Layer

The Ladebug Remote Debugger Protocol uses UDP running over an IP network layer as its transport. The client can use any UDP socket as its source but will always send load and connect requests to UDP socket 410. If the request is successful, the response will have as its source socket the socket allocated to the new debuggee thread. The client will send all messages for this debuggee thread to this socket.

**Note**

This socket has been allocated to this protocol by the Internet Assigned Number Authority (IANA).

The server should always send responses to the source socket of the associated request. The source socket for any message sent by the server should be either 410 (for responses to rejected load and connect messages) or the socket allocated to associated debuggee thread (for all other messages).

# C
# Support for International Users

Ladebug's international support is dependent on the support available in the underlying system. International support is provided through the global locale of the debugger, which is set automatically from the external user environment when you invoke the debugger.

The locale information can be printed using the debugger variable `$lc_ctype`. This (read-only) variable specifies the value of the current LC_CTYPE locale category in the debugger, which is used for all data interpretations.

Ladebug's international support provides the features described in Section C.1 through Section C.3.

## C.1 Support for Input of Local Language Characters in User Commands

Ladebug supports the input of local language characters in user commands. It analyzes user input commands to recognize and process locale-based text (including multibyte characters) in language specific expressions: file names, function names, variable names and values, debugger variable names and values, alias names and parameters, and the debugger prompt.

This support is based on the specific capabilities of the programming language of the debugged unit and of the global debugger locale.

## C.2 Support for Output of Local Language Characters

Ladebug prints all character data according to the current global locale set in the debugger.

Ladebug checks the character to be printed to see if it is a special character that is printed as an escaped expression (for example, `(char)8` is printed as `\\t` ). Ladebug then checks the XPG4 appropriate functions to find out if the character is a printable character in the current locale. If it is, the character is printed as is. Otherwise, the character is printed in octal notation (for example, `(char)1` is printed as `\\001` ).

## C.3 Support for Wide Character Type (wchar_t) in C and C++ Programs

Ladebug prints wcharacters of type `wchar_t` and wide strings of type `wchar_t *` in readable format, similar to their literals L'X' and L"XYZ" (where X, Y ,Z are character codes encoded in the current global locale).

Wide character/string input is accomplished by entering literals in the forms accepted by C and C++ compilers.

The display of these entities is made available through extensions to the following debugger commands:

- `startaddress, endaddress / mode`
- `startaddress / count mode`

Specify C mode (uppercase) to print each sizeof (`wchar_t`) as a wide character.

Specify S mode (uppercase) to print the contents of memory as a wide string.

Use the following command to print the result of the expression for wide characters or wide strings:

```
$  print expression [,...]
```

Wide characters in C and C++ are interpreted as unsigned integers by the Digital UNIX compilers and set as such in the symbol tables of object files. As a result, Ladebug can't distinguish between an unsigned `int` data and a `wchar_t` in the symbol table. You must specify the debugger variable `$wchar_t` to determine the user-preferred output format for unsigned integer expressions (either as integers or as wide characters/strings). If `$wchar_t` has a non-zero value, a wide character is printed in the format of L'X' in accordance with the codeset of the current locale. A wide string is printed in an L"XXXX" format.

Using the following form,

```
$  printf [format [expression,...]]
```

specify format specifier %C (uppercase) to print a wide character and format specifier %S (uppercase) to print a wide string.

Non printable wide characters (as determined by XPG4 library `iswprint( )` function) will be printed as escaped character expressions.

# Index