*dbx* User's Guide

CONTRIBUTORS

Written by Wendy Ferguson, Ken Jones and Leif Wennerberg
Edited by Christina Cary
Production by Gloria Ackley, Kay Maitz, and Lorrie Williams
Engineering contributions by Dave Anderson, Alan Foster, Jay Gischer, Ray Milkey
     and Jon Templeton
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson,
     Erik Lindholm, and Kay Maitz

dbx User's Guide
Document Number 007-0906-110

# Contents

**iii**

# List of Tables

# About This Guide

This guide explains how to use the source level debugger, *dbx*. You can use *dbx* to debug programs in C, C++, Fortran, and assembly language.

## What This Guide Contains

This guide describes the features of *dbx* and provides simple examples of how to use *dbx* to debug programs. Specifically, this guide includes:

Chapter 1, "Getting Started With dbx," introduces some basic *dbx* commands and offers some tips about how to approach a debugging session.

Chapter 2, "Running dbx," explains how to run *dbx* and perform basic *dbx* control functions.

Chapter 3, "Examining Source Files," explains how to examine source files under *dbx*.

Chapter 4, "Controlling dbx," describes features of *dbx* that affect its operation while debugging a program.

Chapter 5, "Examining and Changing Data," describes how to examine and change data in your program while running it under *dbx*.

Chapter 6, "Controlling Program Execution," describes how to use the *dbx* commands that control execution of your program.

Chapter 7, "Debugging Machine Language Code," explains how to debug machine language code.

Chapter 8, "Multiple Process Debugging," explains multiprocess debugging procedures.

Appendix A, "dbx Commands," lists and describes all *dbx* commands.

Appendix B, "Predefined Aliases," lists and describes all predefined *dbx* aliases.

Appendix C, "Predefined dbx Variables," lists and describes all predefined *dbx* variables.

## What You Should Know Before Reading This Guide

This manual is written for programmers, and assumes that you are familiar with general debugging techniques.

## Suggestions for Further Reading

This *dbx User's Guide* is part of the IRIS Developer Option (IDO), which provides the software and documentation that you can use to write applications for Silicon Graphics platforms. A few IDO online and printed manuals that may be of interest to you are listed below.

*Programming on Silicon Graphics Systems: An Overview* provides information about the IRIX programming environment and tools available for application programming. Topics covered include IRIX operating system, compilers, user interface and developer tools, and application libraries.

*Compiling and Performance Tuning Guide* describes the compiler system and programming tools and interfaces, and explains how to improve program performance.

*Topics in IRIX Programming* presents information about internationalizing an application, working with fonts, file and record locking, and inter-process communication.

*C Language Reference Manual* covers the syntax and semantics of the C programming language as implemented on the IRIX operating system.

Silicon Graphics offers software options to assist in software development. The *CASEVision/Workshop* option provides the WorkShop toolset: Debugger, Static Analyzer, Performance Analyzer, Tester, and Build Manager.

You can order a printed manual from Silicon Graphics by calling SGI Direct at 1-800-800-SGI1 (800-7441). Outside the U.S. and Canada, contact your local sales office or distributor.

Silicon Graphics also provides manuals online. To read an online manual after installing it, type `insight` or double-click the InSight icon. It's easy to print sections and chapters of the online manuals from InSight.

## Conventions Used in This Guide

The conventions used in this manual help make information easy to access and understand. The following list describes the conventions and how they are used:

- Command names, including *dbx* commands, appear in italics. For example:

  The *edit* command lets you edit files from within *dbx*.

- Examples, shell prompts, and information displayed on the screen appear in a typewriter font. For example:

  ```
  Process 946: [6] trace count in main
  ```

- Examples of what you enter are in boldface typewriter font. This example illustrates entering **edit soar.c** in response to a `(dbx)` prompt:

  ```
  (dbx) edit soar.c
  ```

- Command arguments you replace with actual values appear in italics. In this example, you replace *name* with the name of an alias:

  **alias** *name*

- Optional arguments are enclosed in square brackets ([ ... ]). In the following example, you can provide one or more directory names as arguments to the command:

  **use** [ *dir* ... ]

- Mutually exclusive arguments to a command are enclosed in braces ({ ... }) and separated by a pipe character ( | ). In the first example below, you can provide either an activation level or a procedure name as an argument to the command. In the second example, because the argument choices are enclosed in square brackets, you can use either the *call* or **return** argument, or omit an argument to the command:

  **func** { *activation_level* | *procedure* }
  **syscall catch** [{ **call** | **return** }]

- File and directory names appear in italics. For example:

  You can put any *dbx* command in the *.dbxinit* file.

- New terms appear in italics. For example:

  Each procedure on the stack defines an *activation level*.

# Getting Started With *dbx*

You can use *dbx* to trace problems in a program at the source code level, rather than at the machine code level. *dbx* enables you to control a program's execution, symbolically monitoring program control flow, variables, and memory locations. You can also use *dbx* to trace the logic and flow of control to acquaint yourself with a program written by someone else.

This chapter introduces some basic *dbx* commands and discusses some tips about how to approach a debugging session. Specifically, this chapter covers:

- "Examining Core Dumps to Determine Cause of Failure"
- "Debugging Your Programs"
- "Studying a New Program"
- "Avoiding Common Pitfalls"

## Examining Core Dumps to Determine Cause of Failure

Even if your program compiles successfully, it still can crash when you try to run it. When a program crashes, it generates a terminating signal that instructs the system to write out to a *core* file. The *core* file is the memory image of the program at the time it crashed.

You can examine the *core* file with *dbx* to determine at what point your program crashed. To determine the point of failure, follow these steps:

1. If the *core* file is not in the current directory, specify the pathname of the *core* file on the *dbx* command line.

   **Note:** If the source code for the program is on a different machine or the source was moved, provide *dbx* with the pathname to search for source code (also see "Specifying Source Directories" on page 15).

2. Invoke *dbx* for the failed program as described in "Invoking dbx" on page 6. *dbx* automatically reads in the local *core* file.

3. Perform a stack trace using the *where* command (described in "Examining the Stack" on page 64) to locate the failure point.

For example, suppose you examine the *core* file for a program called *test*. Suppose the stack trace appears as follows:

```
(dbx) where

>  0 foo2(i = 5) ["/usr/tmp/test.c":44, 0x1000109c]
   1 foo(i = 4) ["/usr/tmp/test.c":38, 0x1000105c]
   2 main(argc = 1, argv = 0xffffffad78) ["/usr/tmp/test.c":55, 0x10001104]
   3 __start() ["/shamu/crt1text.s":137, 0x10000ee4]
```

In this case, *test* crashed at line 44 of the source file *test.c*. The program crashed while executing the function **foo2**. **foo2** was called from line 38 in the function **foo**, which was in turn called from line 55 in the function **main**. You can use the other features of *dbx* to examine values of program variables and otherwise investigate why *test* crashed.

If you use *dbx* to debug code that wasn't compiled using the **–g** option, local variables are invisible to *dbx*, and source lines may appear to jump around as a result of various optimizations. If the code is stripped of its debugging information, *dbx* displays very little information.

## Debugging Your Programs

Debugging a program consists primarily of stopping your program under certain conditions and then examining the state of the program stack and the values stored in program variables.

You stop execution of your program by setting *breakpoints* in your program. Breakpoints can be *unconditional*, in which case they always stop your program when encountered, or *conditional*, in which case they stop your program only if a test condition that you specify is true. (See "Setting Breakpoints" on page 79 for more information.)

To use breakpoints to debug your program, examine your program carefully to determine where problems are likely to occur, and set breakpoints in these problem areas. If your program crashes, first determine which line causes it to crash, then set a breakpoint just before that line.

You can use several *dbx* commands to trace a variable's value. Here's a simple method for tracing a program variable:

1. Use the *stop* command (see "Setting Breakpoints" on page 79) to set breakpoints in the program at locations where you want to examine the state of the program stack or the values stored in program variables.

2. Use the *run* or *rerun* command (described in "Running Your Program" on page 9) to run your program under *dbx*. The program stops at the first breakpoint that it encounters during execution.

3. Examine the program variable as described in "Displaying the Value of a Variable" on page 47. Examine the program stack as described in "Examining the Stack" on page 64.

4. Use the *cont* command (see "Continuing Execution After a Breakpoint" on page 84) to continue execution past a breakpoint. However, you cannot continue execution past a line that crashes the program.

## Studying a New Program

Use *dbx* to examine the flow of control in a program. When studying the flow of control within a program, use the *dbx* commands *stop*, *run/rerun*, *print*, *next*, *step*, and *cont*. To study a new program:

1. Use the *stop* command to set breakpoints in the program. When you execute the program under *dbx*, it stops execution at the breakpoints.

   If you want to review every line in the program, set a breakpoint on the first executable line. If you don't want to look at each line, set breakpoints just before the sections you intend to review.

2. Use the *run* and *rerun* commands to run the program under *dbx*. The program stops at the first breakpoint.

3. Use the *print* command to print the value of a program variable at a breakpoint.

4. Use the *step*, *next*, or *cont* command to continue past a breakpoint and execute the rest of the program.

   - *step* executes the next line of the program. If the next line is a procedure call, *step* steps down into the procedure. *step* is described in "Stepping Using the step Command" on page 98.

- *next* executes the next line; if it is a procedure, *next* executes it but does not step down into it. *next* is described in "Stepping Using the next Command" on page 100.

- *cont* resumes execution of the program past a breakpoint and does not stop until it reaches the next breakpoint or the end of the program. *cont* is explained in "Continuing Execution After a Breakpoint" on page 84.

Another tool that you can use to follow the execution of your program is the *trace* command (described in "Tracing Program Execution" on page 85). With it you can examine:

- values of variables at specific points in your program or whenever variables change value

- parameters passed to and values returned from functions

- line numbers as they are executed

## Avoiding Common Pitfalls

You may encounter some problems when you debug a program. Common problems and their solutions are listed below.

- If *dbx* does not display variables, recompile the program with the **–g** compiler option. Note that in some cases, this may cause the problem to go away, or its symptoms to change.

- If the debugger's listing seems confused, try separating the lines of source code into logical units. The debugger may get confused if more than one source statement occurs on the same line.

- If the debugger's executable version of the code doesn't match the source, recompile the source code. The code displayed in the debugger is identical to the executable version of the code.

- If code appears to be missing, it may be contained in an include file or a macro. The debugger treats macros as single lines. To debug a macro, expand the macro in the source code.

# Running *dbx*

This chapter explains how to run *dbx*—specifically, it covers:

- "Compiling a Program for Debugging Under dbx"
- "Compiling and Linking Programs With Dynamic Shared Objects"
- "Invoking dbx" from a shell
- "Running Your Program"
- "Automatically Executing Commands on Startup"
- "Using Online Help"
- "Entering Multiple Commands on a Single Line"
- "Spanning a Command Across Multiple Lines"
- "Invoking a Shell" from *dbx*
- "Quitting dbx"

## Compiling a Program for Debugging Under *dbx*

Before using *dbx* to debug a program, compile the program using the **–g** option (for example, *cc –g*). The **–g** option includes additional debugging information in your program object so that *dbx* can list local variables and find source lines.

If you use *dbx* to debug code that was not compiled using the **–g** option, local variables are invisible to *dbx*, and source lines may appear to jump around oddly as a result of various optimizations. It is more difficult to debug code without reliable references to lines of source code.

## Compiling and Linking Programs With Dynamic Shared Objects

This section summarizes a few things you need to know if you compile and link your program with Dynamic Shared Objects (DSOs). A DSO is a relocatable shared library. By linking with a DSO, you keep your program size small and use memory efficiently.

If you compile and link with DSOs, *dbx* automatically notices their use in the program and picks up the relevant debugging information. The *dbx* command *listobj* shows any DSOs in a process. The *dbx* command *whichobj* lists all DSOs in which a specified variable is present. The *dbx* command *listregions* identifies DSO addresses at run time.

The *dbx* help section on *hint_dso* has more information on *dbx* and DSOs. For more information on DSOs, see "Using Dynamic Shared Objects" in the *Compiling and Performance Tuning Guide*.

## Invoking *dbx*

This section describes how to invoke dbx and includes:

- "dbx Options"
- "Specifying Object and Core Files"
- "The dbx Prompt"
- "Specifying Files with dbx Commands"

To invoke *dbx* from the shell command line, type **dbx**. The syntax is:

**dbx** [*options*] [*object_file* [*corefile*]]

## *dbx* Options

Table 2-1 lists options you can give to *dbx*. These options are described in detail later in this chapter.

**Table 2-1**        *dbx* Command-Line Options

| Option | Description |
|---|---|
| **–c** *file* | Selects a command file other than *.dbxinit* to execute on starting *dbx*. For information on *.dbxinit*, see "Automatically Executing Commands on Startup." |
| **–d** | Provides startup information to the shell when a program is started with the *run* command. |
| **–e** *num* | Chooses a large size for the evaluation stack (as large as you want). The default stack size is 20,000 bytes. *num* = number of bytes. If you see the message too large to evaluate, rerun *dbx* suppling a value greater than 20,000. |
| **–i** | Uses interactive mode. This option prompts for source even when it reads from a file and treats data in a file as if it comes from a terminal (**stdin**). This option does not treat "#" characters as comments in a file. |
| **–I** *dir* | Tells *dbx* to look in the specified directory (in addition to the current directory and the object file's directory) for source files. To specify multiple directories, use a separate **–I** for each. If no directory is specified when you invoke *dbx*, it looks for source files in the current directory and in the object file's directory. From *dbx*, changes the directories searched for source files with the *use* and *dir* commands. |
| **–k** | Turns on kernel debugging. When debugging a running system, specify */dev/kmem* as the core file. |
| **–N** | Sets the *dbx* variable $*nonstop* to 1 on startup: attaching to a process does not stop the process. Affects only the *dbx* options **-p** and **-P** and the *addproc* command. |
| **–P** *name* | Debugs the running process with the specified *name* (*name* as described in the *ps*(1) reference page). |
| **–p** *pid* | Debugs the process specified by the *pid* number. |

**Table 2-1 (continued)**        *dbx* Command-Line Options

| Option | Description |
|--------|-------------|
| **–R** | Allows breakpoints in **rld**. |
| **–r** *program* [*arg*] | Runs the named program upon entering *dbx*, using the specified arguments. The *.dbxinit* file (if any) is read and executed *after* executing the **object_file**. You cannot specify a core file with **–r**. |

## Specifying Object and Core Files

The *object_file* is the name of the executable object file that you want to debug. It provides both the code that *dbx* executes and the symbol table that provides variable and procedure names and maps executable code to its corresponding source code in source files.

A *corefile* is produced when a program exits abnormally and produces a core dump. *dbx* allows you to provide the name of a core file that it uses as "the contents of memory" for the program that you specify. If you provide a core file, *dbx* lists the point of program failure. You can then perform stack traces and examine variable values to determine why a program crashed. However, you cannot force the program to execute past the line that caused it to crash.

If you don't specify a *corefile*, *dbx* examines the current directory for a file named *core*. If it finds *core*, and if *core* seems (based on data in the core file) to be a core dump of the program you specified, *dbx* acts as if you had specified *core* as the core file.

You can specify object and core files either as arguments when you invoke *dbx* or as commands that you enter at the *dbx* prompt.

## The *dbx* Prompt

Once *dbx* starts, it displays the prompt:

```
(dbx)
```

To change this prompt, change the value of the *dbx $prompt* variable. "Setting dbx Variables" on page 24 describes how to set *dbx* variables.

### Specifying Files with *dbx* Commands

The *givenfile* and *corefile dbx* commands allow you to set the object file and the core file, respectively, while *dbx* is running.

**givenfile** [*file*]

If you provide a filename, *dbx* kills the currently running processes and loads the executable code and debugging information found in *file*.

If you do not provide a filename, *dbx* displays the name of the program that it is currently debugging without changing it.

**corefile** [*file*]

If you provide a filename, *dbx* uses the program data stored in the core dump *file*.

If you do not provide a filename, *dbx* displays the name of the current core file without changing it.

## Running Your Program

You can start your program under *dbx* using the *run* or *rerun* command.

**run run-arguments**

The *run* command starts your program and passes to it any arguments that you provide. The command uses your shell (the program named in the *SHELL* environment variable or */bin/sh* if an environment variable does not exist) to process a *run* command. The syntax allowed in your shell is allowed on the *run* command line. All shell processing is accepted, such as expansion and substitution of **\*** and **?** in filenames. Redirection of the program's standard input and standard output, and/or standard error is also done by the shell.

In other words, the *run* command does exactly what typing **target run-arguments** at the shell prompt does. You can specify *target* either on *dbx* invocation or in a prior *givenfile* command. *dbx* passes *./target* as **argv[0]** to *target* when you specify it as a relative pathname.

The *run* command does not invoke the initialization files of the Bourne, C, and Korn shells before it starts a program. If you use a non-standard shell, before you run a program set the *dbx* variable *$shellparameters* to a string that will instruct the shell to not load the initialization file. For example, for the C shell you would enter **set**

**9**

**$shellparameters = "-f"**.To verify exactly how your application is being started by the *run* or *rerun* command, start *dbx* with the **-d** option.

If the environment variable *SHELL* is set to a C shell and your program has file-descriptors other than the default values: 0,1,2, switch to the Bourne shell before you invoke the *run* command. This means you can only use **sh**-style redirections, but **csh** would close the extra file-descriptors. Make the switch, for the purpose of running your program, with the *dbx* command **setenv SHELL /bin/sh.**

A *run* command must appear on a line by itself and cannot be followed by another *dbx* command separated by a semi-colon (;). Terminate the command line with a return (new-line). Note that you cannot include a *run* command in the command list of a *when* command.

**rerun** [*run-arguments*]

> The *rerun* command, without any arguments, repeats the last *run* command if applicable. Otherwise *rerun* is equivalent to the *run* command without any arguments.

The *sort* command takes an input file and produces a sorted output file; you can specify input and output files either through command-line arguments or file redirection.

For example, from the command line you can enter:

```
% sort -i input -o output
% sort < input2 > output2
```

If you are debugging the *sort* program, the equivalent **dbx** commands are:

```
(dbx) run -i input -o output
(dbx) run < input2 > output2
```

If you execute these *run* commands in the order presented, you can repeat the last run command by using the *rerun* command:

```
(dbx) rerun
```

## Automatically Executing Commands on Startup

You can use an editor to create a *.dbxinit* command file. This file contains various *dbx* commands that automatically execute when you invoke *dbx*. You can put any *dbx*

command in the *.dbxinit* file. If a command requires input, the system prompts you for it when you invoke *dbx*.

On invocation, *dbx* looks for a *.dbxinit* file in the current directory. If the current directory does not contain a *.dbxinit* file, *dbx* looks for one in your home directory. (This assumes that you have set the IRIX system *HOME* environment variable.)

## Using Online Help

The *dbx* command *help* has several options:

`help`          shows the supported *dbx* commands

`help` *keyword*   shows information pertaining to the given *keyword*, such as **alias, help, most_used, quit, playback, record,** and so on

`help all`       shows the entire *dbx* help file

When you type `help all`, *dbx* displays the file using the command name given by the *dbx $pager* variable. The *dbx* help file is large and can be difficult to read even if you use a simple paging program like *more*(1). You can set the *$pager* variable to a text editor like *vi*(1) or to your favorite editor.

For example, just add the following command in your *.dbxinit* file:

```
set $pager = "vi"
```

When the above entry is in your *.dbxinit* file, *dbx* displays the help file in *vi*. You can then use the editor's search commands to look through the help file quickly. Quit the editor to return to *dbx*.

## Entering Multiple Commands on a Single Line

You can use a semicolon (*;*) as a separator to include multiple commands on the same command line. This is useful with commands such as *when* (described in "Writing Conditional Commands" on page 87) as it allows you to include multiple commands in the command block. For example:

```
(dbx) when at "myfile.c":37 {print a ; where ; print b}
```

## Spanning a Command Across Multiple Lines

You can use a backslash (\) at the end of a line of input to indicate that the command is continued on the next line. This can be convenient when entering complex commands such as an alias definition (aliases are discussed in "Creating and Removing dbx Aliases" on page 28).

For example:

```
(dbx) alias foll "print *(struct list *)$p ; \
set $p = (int)((struct list *)($p))->next"
```

**Hint:** You can also use the *hed* command for creating and modifying commands. "The History Editor" on page 27 has details on this command.

## Invoking a Shell

To invoke a subshell, enter **sh** at the *dbx* prompt, or enter **sh** and a shell command at the *dbx* prompt. After invoking a subshell, type **exit** or **<Ctrl-d>** to return to *dbx*.

The syntax for the *sh* command is:

**sh**                Invoke a subshell.

**sh** *command*     Execute the specified shell command. *dbx* interprets the rest of the line as a command to pass to the spawned shell process, unless you enclose the command in double-quotes or you terminate your shell command with a semicolon (;).

For example, to spawn a subshell, enter:

```
(dbx) sh
%
```

To display the end of the file *datafile*, enter:

```
(dbx) sh tail datafile
```

**12**

# Quitting *dbx*

To end a *dbx* debugging session, enter the *quit* command at the *dbx* prompt:

(dbx) **quit**

**13**

# Examining Source Files

This chapter explains how to examine source files under *dbx*. It describes:

- "Specifying Source Directories"
- "Listing Source Code"
- "Searching Through Source Code"
- "Changing Source Files"
- "Listing Inlines and Clones"
- "Calling an Editor"

## Specifying Source Directories

Based on the information contained in an object file's symbol table, *dbx* determines from which source files the program was compiled and prints portions of these files as appropriate.

Object files compiled with **–g** record the absolute path names to the source files. Each time *dbx* needs a source file, it first searches the absolute path for the source file. If the source file is not present (or if the object file was not compiled with **–g**), *dbx* checks its own list of directories for source files.

By default, the *dbx* directory list contains only the current directory (from which you invoked *dbx*) and the object file's directory (if it is different from the current directory). Each time *dbx* searches this list, it searches all directories in the list in the order in which they appear until it finds the file with the specified name.

### Specifying Source Directories With Arguments

You can specify additional source directories when you invoke *dbx* with the **–I** option. To specify multiple directories, use a separate **–I** for each.

For example, consider debugging a program called *look* in */usr/local/bin*, the source for which resides in */usr/local/src/look.c*. To debug this program, you can invoke *dbx* from the */usr/local/bin* directory by entering:

```
% dbx -I /usr/local/src look
```

## Specifying Source Directories With *dbx* Commands

The *dir* and *use* commands allow you to specify a source directory list while *dbx* is running.

**dir** [*dir* **...**]     If you provide one or more directories, *dbx* adds those directories to the end of the source directory list.

If you do not provide any directories, *dbx* displays the current source directory list.

**use** [*dir* **...**]     If you provide one or more directories, *dbx* replaces the source directory list with the directories that you provide.

If you do not provide any options, *dbx* displays the current source directory list.

**Note:** Both the *dir* and *use* commands recognize absolute and relative pathnames (for example, *../src*); however, they do not recognize C shell tilde (~) syntax (for example, *~kim/src*) or environment variables (for example, *$HOME/src*).

### Examples of *dir* and *use*

Let's debug the *look* program in */usr/local/bin*. Recall that the source resides in */usr/local/src/look.c*. If you invoke *dbx* from the */usr/local/bin* directory without specifying */usr/local/src* as a source directory, it will not initially appear in the directory list:

```
(dbx) dir
.
```

However, you can add */usr/local/src* with the *dir* command by entering:

```
(dbx) dir /usr/local/src
(dbx) dir
. /usr/local/src
```

If you use the *use* command instead, the current directory is no longer contained in the source directory list:

```
(dbx) use /usr/local/src
(dbx) use
/usr/local/src
```

**Path Remapping**

The debugging information for programs compiled with **–g** includes the full pathnames for source files. By default, *dbx* uses these pathnames to search for source files. However, if you are debugging a program that was compiled somewhere else and you want to specify a new path to the sources, you can use path remapping. You can substitute one pattern for another to remap the path so *dbx* can find the source file.

**dir** *pattern1*:*pattern2*

> The *dir* (or *use*) command allows you to remap directories and specify a new path to the source. *dbx* substitutes *pattern2* for *pattern1*.

For example, a compiled program's source is */x/y/z/kk.c* and the source was moved to */x/y/zzz/kk/kk.c*. Specify the *dir* (or *use*) command to remap the path:

```
(dbx) dir /z/:/zzz/kk/
```

The new path is */x/y/zzz/kk/kk.c*, where */z/* has been remapped to the string following the colon.

**Controlling use of Path Remappings and Your Source-Directory List**

The *dbx* variable *$sourcepathrule* controls how, in a source-file search, *dbx* uses path remappings and the source-directory list created by the *dir* and *use* commands. Table 3-1 summarizes the effects of *$sourcepathrule*.

**Table 3-1**    Effect of *$sourcepathrule* on Use of Path Remapping and Source Directory List

| Value | Effect |
|-------|--------|
| 0 (default) | Search for a source file by:<br>a) using the pathname in the object file's debugging information; if the file is not found, then<br>b) examine pathnames remapped by the *dir* or *use* command; if the file is still not found, then<br>c) reduce full pathnames to base file names and search the list of directories created by the *dir* or *use* command. |
| 1 | Permute the default source-file search sequence to: step b, step c, then step a. |
| 2 | Use only steps b and c of the default source-file search sequence. |

*$sourcepathrule* = 1 is useful when, for example, you move source files after you compile your program. You can direct *dbx* to the correct files.

*$sourcepathrule* = 2 is useful when, for example, your network is slow and you have full pathnames in your debugging information that point to files on other machines. The debugger ignores all pathnames in the debugging information and, hence, will not attempt access over the network.

# Changing Source Files

The *file* command changes the current source file to a file that you specify. The new file becomes the current source file, on which you can search, list, and perform other operations. For example, to set the current source file to *"Examining the Stack" on page 54procedure.c*, enter:

```
(dbx) file procedure.c
```

**Note:** If your program is large, typically you store the source code in multiple files. *dbx* automatically selects the proper source file for the section of code that you are examining. Thus, many *dbx* commands reset the current source file as a side effect. For example, when you move up and down activation levels in the stack using the *up* and *down* commands, *dbx* changes the current source file to whatever file contains the source for the procedure (see "Examining the Stack" on page 64 for more information on activation levels).

If you enter the *file* command without any arguments, *dbx* prints the current source file:

```
(dbx) file
procedure.c
```

You can also change the current source file by typing:

```
(dbx) func procedure
```

You can use the *tag* command to search the tag file for *procedure*:

```
(dbx) tag procedure
```

The *tag* command finds C preprocessor macros if they have arguments (**func** *procedure* cannot). For more information about the tag file, see *ctags*(1).

## Listing Source Code

The *list* command displays lines of source code. The *dbx* variable *$listwindow* defines the number of lines *dbx* lists by default. The *list* command uses the active frame and line of the current source file unless overridden by a *file* command. Any execution of the program overrides the *file* command by establishing a new current source file.

The syntax for the *list* command is:

**list**             Lists *$listwindow* lines beginning at the current line (or list the line of the current **pc** if the current line is unknown or not set).

**list** *exp*        Lists *$listwindow* lines starting with the line number given by the expression *exp*. The expression can be any valid expression that evaluates to an integer value as described in "Using Expressions" on page 37.

**list** *exp1***:***exp2*  Lists *exp2* lines, beginning at line *exp1*.

**list** *exp1***,***exp2*  Lists all source between line *exp1* and line *exp2* inclusive.

**list** *func*         Lists *$listwindow* lines starting at procedure *func*.

**list** *func***,***exp*  Lists all source between *func* and *exp*, inclusive.

**list** *func***:***exp*  Lists *exp* lines, beginning at *func*.

A > symbol prints to the left of the line that is the current line. A * symbol prints to the left of the line of the current **pc** location.

For example, to list lines 20–35 of a file, enter:

```
(dbx) list 20,35
```

In response to this command, *dbx* displays lines 20 through 35 and sets the current line to 36.

To list 15 lines starting with line 75, enter:

```
(dbx) list 75:15
```

In response to this command, *dbx* displays lines 75 through 89 and sets the current line to 90.

## Listing Inlines and Clones

The compiler may inline routines, replacing a call with quotes of code from the called routine, either as a result of optimization or C++ inline directives. Clones are specialized versions of routines that you can use to get faster-running code. The source for cloned routines is called a *root*.

In special cases, you may want to find inlined routines or clones. The commands *listinlines* and *listclones* find the routines, if enough debugging information is available. Compilations with the **-32** option or with IRIX 6.2 and earlier base compilers do not have the necessary information; *listinlines* and *listclones* will show nothing.

The syntax for the *listinlines* command is:

**listinlines**    Lists all inlined routines with their start and end addresses.

**listinlines** *func*
                Lists all the inlined instances of *func* with their start and end addresses.

For example, **::MultPoints** is a C++ routine and you enter:

(dbx) **listinlines** ::MultPoints

The *dbx* output lists the address ranges of all the instances where **::MultPoints** is inlined.

The syntax for the *listclones* command is similar:

**listclones**    Lists all the root functions and their derived clones.

**listclones** *func*
                Lists the root and all derived clones for *func*.

## Searching Through Source Code

Use the forward slash (/) and question mark (?) commands to search through the current file for regular expressions in source code. For a description of regular expressions, see the *ed*(1) reference page.

The search commands have the following syntax:

/[*reg_exp*]    Search forward through the current file from the current line for the regular expression *reg_exp*. If *dbx* reaches the end of the file without finding the regular expression, it wraps around to the beginning of the file. *dbx* prints the first source line containing a match of the search expression.

                If you don't supply a regular expression, *dbx* searches forward based on the last regular expression searched.

?[*reg_exp*]    Search backward through the current file from the current line for the regular expression *reg_exp*. If *dbx* reaches the beginning of the file without finding the regular expression, it wraps around to the end of the file. *dbx* prints the first source line containing a match of the search expression.

If you don't supply a regular expression, *dbx* searches backward based on the last regular expression searched.

For example, to search forward for the next occurrence of the string "errno," enter:

```
(dbx) /errno
```

To search backward for the previous occurrence of either "img" or "Img," enter:

```
(dbx) ?[iI]mg
```

## Calling an Editor

The *edit* command lets you edit files from within *dbx*:

**edit**             *The edit* command invokes an editor (*vi* by default) on the current source file. If you set the *dbx* variable *$editor* to the name of an editor, the *edit* command invokes that editor. If you do not set the *$editor*, *dbx* checks the environment variable *EDITOR* and, if set, invokes that editor. When you exit the editor, you return to the *dbx* prompt.

**edit** *file*      Invokes the editor on the given file.

**edit** *procedure*   Invokes the editor on the file that contains the source for the specified procedure. *dbx* extended naming does not work. You may only name procedures that *dbx* can find with a simple name: procedures in the current activation stack and global procedures.

For example, to edit a file named *soar.c* from within *dbx*, type:

```
(dbx) edit soar.c
```

The *edit* command is also useful for editing *dbx* script files. See "Executing dbx Scripts" on page 35 for more information on script files.

# Controlling *dbx*

This chapter describes features of *dbx* that affect its operation while debugging a program. Specifically, this chapter covers:

- "Creating and Removing dbx Variables"
- "Using the History Feature and the History Editor"
- "Creating and Removing dbx Aliases"
- "Recording and Playing Back dbx Input and Output"
- "Executing dbx Scripts"

## Creating and Removing *dbx* Variables

*dbx* allows you to define variables that you can use within *dbx* to store values. These variables exist entirely in *dbx*; they are not part of your program. You can use *dbx* variables for a variety of purposes while debugging. For example, you can use *dbx* variables as temporary storage, counters, or pointers that you use to step through arrays.

*dbx* also provides many predefined variables that control how various *dbx* commands function. Appendix C, "Predefined dbx Variables" provides a complete list of predefined *dbx* variables and their purposes.

*A dbx* variable does not have a fixed type. You can assign a *dbx* variable any type of value, even if it already has a value of a different type. However, a variable predefined by *dbx* does have a fixed predefined type.

You can use almost any name for *dbx* variables. A good practice to follow is to use a dollar sign ($) as the first character of all *dbx* variables to prevent conflicts with most program variable names. All of *dbx*'s predefined variables begin with a dollar sign.

**23**

The commands described in this section apply only to the manipulations of *dbx* variables, not program variables. "Displaying and Changing Program Variables" on page 46 describes how to manipulate program variables.

## Setting *dbx* Variables

The *set* command sets a *dbx* variable to a given value, defining the variable if it does not exist:

**set** *var* = *exp*    Define (or redefine) the specified *dbx* variable, setting its value to that of the expression you provide.

You can display the value of a variable with the *print* command. For example:

```
(dbx) set $k = 1
(dbx) print $k
1
(dbx) set $k = $k +23
(dbx) print $k
24
(dbx) print $k / 11
2
```

In the above example, *dbx* performs an integer division because both the variable $k and the constant 11 are integers. If you assign a floating point value to $k and evaluate the expression again, *dbx* performs a floating point division:

```
(dbx) set $k = 24.0
(dbx) print $k
24.0
(dbx) print $k / 11
2.1818181818181817
```

**Note:** We recommend that you begin a *dbx* variable with a **$** to avoid confusion with a program variable. A *dbx* variable without a leading **$** hides any program variable that has the same name. The only way to see the program variable is to remove the *dbx* variable with an *unset* command.

### Listing *dbx* Variables

If you enter the *set* command without providing any arguments, *dbx* displays (in alphabetical order) a list of all currently defined *dbx* variables, including predefined variables. Partial output looks like this:

```
(dbx) set

$addrfmt        "0x%x"
$addrfmt64      "0x%llx"
$assignverify   1
$casesense      2
$ctypenames     1
$curevent       3
$curline        44
$curpc          268439708
...
$stacktracelimit        1024
$stdc           0
$stepintoall    0
$tagfile        "tags"
```

### Removing Variables

The *unset* command removes a *dbx* variable. For example, to delete the variable *$k*, enter:

```
(dbx) unset $k
```

## Using the History Feature and the History Editor

The *dbx* history feature is similar to the C shell's history feature in that it allows you to repeat commands that you have entered previously. However, unlike the C shell's history feature, *dbx* does not allow you to execute a history command anywhere except the beginning of a line. Also, *dbx* does not support history substitution of command arguments such as the C shell **!$** argument.

### Examining the History List

*dbx* stores all commands that you enter in the history list. The value of the *dbx* variable *$lines* determines how many commands are stored in the history list. The default value is 100.

Display the history list with the *history* command. For example, after setting a breakpoint, running a program, and examining some variables, your history list might look something like this:

```
(dbx) history

   1      set $prompt = "(dbx)"
   2      set $page=0
   3      set $pimode=1
   4      stop in main
   5      history
```

### Repeating Commands

You can execute any of the commands contained in the history list. Each history command begins with an exclamation point (*!*):

| | |
|---|---|
| **!!** | Repeats the previous command. If the value of the *dbx* variable *$repeatmode* is set to 1, then entering a carriage return at an empty line is equivalent to executing *!!*. By default, *$repeatmode* is set to 0. |
| **!***string* | Repeats the most recent command that starts with the specified *string*. |
| **!***integer* | Repeats the command associated with the specified *integer* in the history list. |
| **!-***integer* | Repeats the command that occurred *integer* times before the most recent command. Entering **!-1** executes the previous command, **!-2** the command before that, and so forth. |

You can use the *!!* command to facilitate single-stepping through your program. (Single-stepping is described in "Stepping Through Your Program" on page 97.) The following illustrates using the *next* command to execute 5 lines of source code and then using the *!!* command to repeat the *next* command.

For example:

```
(dbx) next 5
```

```
Process 22545 (test) stopped at [main:60 ,0x10001150]
  60   total += j;
(dbx) !!
(!! = next 5)
Process 22545 (test) stopped at [main:65 ,0x100011a0]
  65   printf("i = %d, j = %d, total = %d\n",i,j,total);
```

Another convenient way to repeat a commonly used command is with **!***string*. For example, suppose that you occasionally print the values of certain variables using the *printf* command while running your program under *dbx*. (The *printf* command is described in "Printing Expressions" on page 41.) In this case, as long as you do not enter any command beginning with "pr" after you enter the *printf* command, you can repeat the *printf* command by entering **!pr**. For example:

```
(dbx) printf "i = %d, j = %d, total = %d\n", i, j, total
i = 4, j = 25, total = 1
  ...
(dbx) !pr
i = 12, j = 272, total = 529
```

Using **!***integer*, you can repeat any command in the history list. If you want to repeat the *printf* command, but you have entered a subsequent *print* command, examine the history list and then explicitly repeat the *printf* command using its reference number. For example:

```
(dbx) history

  1      set $prompt = "(dbx)"
  2      set $page=0
 ...
  45     printf "i = %d, j = %d, total = %d\n", i, j, total
  46     next
 ...
  49     print j
 ...
  53     history
(dbx) !45
(!45 = printf "i = %d, j = %d, total = %d\n", i, j, total)
i = 9, j = 43, total = 1084
```

## The History Editor

The history editor, *hed*, lets you use your favorite editor on any or all of the commands in the current *dbx* history list. When you enter the *hed* command, *dbx* copies all or part of the

history list into a temporary file that you can edit. When you quit the editor, any commands left in this temporary file are automatically executed by *dbx*.

If you have set the *dbx* variable *$editor* to the name of an editor, the *hed* command invokes that editor. If you have not set the *dbx* variable *$editor*, *dbx* checks whether you have set the environment variable *EDITOR* and, if so, invokes that editor. If you have not set either the *dbx* variable or the environment variable, *dbx* invokes the *vi* editor.

The syntax for the *hed* commands is:

**hed**              Edits only the last line of the history list (the last command executed).

**hed** *num1*        Edits line *num1* in the history list.

**hed** *num1***,***num2*
                     Edits the lines in the history list from *num1* through *num2*.

**hed all**          Edits the entire history list.

By default, *dbx* doesn't display the commands that it executes as a result of the *hed* command (the *dbx* variable *$pimode* is set to 0). If *$pimode* is set to 1, *dbx* displays the commands as it executes them. See *$pimode* in Appendix C, "Predefined dbx Variables" for more information.

## Creating and Removing *dbx* Aliases

You can create *dbx* aliases for debugger commands. Use these aliases as you would any other *dbx* command. When *dbx* encounters an alias, it expands the alias using the definition you provided.

*dbx* has a group of predefined aliases that you can modify or delete. These aliases are listed and described in Appendix B, "Predefined Aliases."

If you find that you often create the same aliases in your debugging sessions, you can include their definitions in your *.dbxinit* file so that they are automatically defined for you. See "Automatically Executing Commands on Startup" on page 10 for more information on the *.dbxinit* file.

## Listing Aliases

You can display the definition of aliases using the *alias* command:

**alias**          Lists all existing aliases.

**alias** *name*     Lists the alias definition for *name*.

For example, to display the definitions of the predefined aliases "l" and "bp," enter:

```
(dbx) alias l
"list"
(dbx) alias bp
"stop in"
```

## Creating Command Aliases

You can use the *alias* command to define new aliases:

**alias** *name command*
              Defines *name* as an alias for *command*.

**alias** *name "string"*
              Defines *name* as an alias for *string*. With this form of the *alias* command, you can provide command arguments in the alias definition.

**alias** *name***(***arg1* [**,** ... *argN*]**)** *"string"*
              Defines *name* as an alias for *string*. *arg1* through *argN* are arguments to the alias, appearing in the *string* definition. When you use the alias, you must provide values for the arguments, which *dbx* then substitutes in *string*.

The simplest form of an alias is to redefine a *dbx* command with a short alias. Many of the predefined *dbx* aliases fall into this category: "a" is an alias for the *assign* command, "s" is an alias for the *step* command. When you use one of these aliases, *dbx* simply replaces it with the command for which it is an alias. Any arguments that you include on the command line are passed to the command.

For example, if you to create "gf" as an alias for the *givenfile* command, enter:

```
(dbx) alias gf givenfile
(dbx) alias gf
"givenfile"
(dbx) gf
```

```
Current givenfile is test
(dbx) gf test2
Process 22545 (test) terminated
Executable /usr/var/tmp/dbx_examples/test2
(dbx) gf
Current givenfile is test2
```

More complex alias definitions require more than the name of a command. In these cases, you must enclose the entire alias definition string in double quotation marks. For example, you can define a brief alias to print the value of a variable that you commonly examine. Note that you must use the escape character (\) to include the double quotation marks as part of the alias definition. For example:

```
(dbx) alias pa "print \"a =\", a"
(dbx) alias pa
"print "a =", a"
(dbx) pa
a = 3
```

You can also define an alias so that you can pass arguments to it, much in the same way that you can provide arguments in a C language macro definition. When you use the alias, you must include the arguments. *dbx* then substitutes the values that you provide in the alias definition.

To illustrate this, consider the following alias definition:

```
(dbx) alias p(arg1, arg2, arg3, arg4) "print '|arg1|arg2|arg3|arg4|'"
(dbx) alias p
(arg1, arg2, arg3, arg4)"print '|arg1|arg2|arg3|arg4|'"
```

The "p" alias takes four arguments and prints them surrounded by vertical bars (|). For example:

```
(dbx) p(1,2,3,4)
|1|2|3|4|
(dbx) p( first, second, 3rd,4)
| first| second| 3rd|4|
```

In the previous example, *dbx* retains any spaces that you enter when calling an alias.

You can also omit arguments when calling an alias as long as you include the commas as argument separators in the alias call:

```
(dbx) p(a,,b,c)
|a||b|c|
```

```
(dbx) p(,first missing, preceding space,)
||first missing| preceding space||
(dbx) delete
delete
```

## Removing Command Aliases

The *unalias* command removes the alias you provide as an argument. For example, to remove the "pa" alias defined in the previous section, enter:

```
(dbx) unalias pa
```

You can remove any of the predefined *dbx* aliases; however, these aliases are restored the next time you start *dbx*.

## Alias Example

One way to follow linked lists is to use aliases and casts, another is to use the *duel* command (See "Using the High-Level Debugging Language duel" in Chapter 5). This example shows how to construct an alias that follows a simple linked list with members defined by the following structure:

```
struct list { struct list *next; int value; };
```

In this example, a *dbx* variable called *$p* is used as a pointer to a member of the linked list. You can define an alias called "foll" to print the contents of the list member to which *$p* currently points and then advance to the next list member. Because the command is too long to fit onto one line, this example uses the backslash character (\) to continue the command on a second line:

```
(dbx) alias foll "print *(struct list *)$p ; \
set $p = (long)((struct list *)($p))->next"
```

Casting *$p* to an integer type when following the link (the second assignment in the alias) is essential. If omitted, *dbx* may leave the *$p* reference symbolic and if so, goes into an infinite loop. (Type **Ctrl-c** to interrupt *dbx* if it gets into the infinite loop.)

Before using this alias, you must set *$p* to point at the first list member. In this example, assume that the program variable *top* points to the first list member. Then you can use the "foll" alias to follow the linked list, printing the contents of each member as you proceed:

```
(dbx) set $p = top
```

**31**

```
(dbx) foll
struct list {
    next = 0x7fffc71c
    value = 57
}
(dbx) foll
struct list {
    next = 0x7fffc724
    value = 3
}
(dbx) foll
struct list {
    next = 0x7fffc72c
    value = 12
}
```

## Recording and Playing Back *dbx* Input and Output

*dbx* allows you to play back your input and record *dbx*'s output. *dbx* saves the information that you capture in files, which allows you to create command scripts that you can use in subsequent *dbx* sessions.

### Recording Input

Use the *record input* command to start an input recording session. Once you start an input recording session, all commands to *dbx* are copied to the specified file. If the specified file already exists, *dbx* appends the input to the existing file. You can start and run as many simultaneous *dbx* input recording sessions as you need.

Each recording session is assigned a number when you begin it. Use this number to reference the recording session with the *unrecord* command described in "Ending a Recording Session" on page 33.

After you end the input recording session, use the command file with the *playback input* or *pi* commands to execute again all the commands saved to the file. See "Playing Back Input" on page 33.

For example, to save the recorded input in a file called *script*, enter:

```
(dbx) record input script
```

```
[4] record input script (0 lines)
```

If you do not specify a file to *record input*, *dbx* creates a temporary *dbx* file in the */tmp* directory. The name of the temporary file is stored in the *dbx* variable *$defaultin*. You can display the temporary filename using the *print* command:

(dbx) **print $defaultin**

Because the *dbx* temporary files are deleted at the end of the *dbx* session, use the temporary file to repeat previously executed *dbx* commands in the current debugging session only. If you need a command file for use in subsequent *dbx* sessions, you must specify the filename when you invoke *record input*. If the specified file exists, the new input is appended to the file.

## Ending a Recording Session

To end input or output recording sessions, use the *unrecord* command.

**unrecord** *session1* [**,** *session2* ...]
  Turns off the specified recording session(s) and closes the file(s) involved.

**unrecord all**  Turns off all recording sessions and closes all files involved.

For example, to stop recording session 4, enter the *dbx* command:

(dbx) **unrecord 4**

To stop all recording sessions, enter:

(dbx) **unrecord all**

The *dbx status* command does not report on recording sessions. To see whether or not any active recording sessions exist, use the *record* command described in "Examining the Record State" on page 35.

## Playing Back Input

Use *playback input* to execute commands that you recorded with the *record input* command. Two aliases exist for *playback input*: *pi* and *source*.) If you don't specify a filename, *dbx* uses the current temporary file that it created for the *record input* command.

If you set the *dbx* variable *$pimode* to nonzero, the commands are printed out as they are played back. By default, *$pimode* is set to zero.

## Recording Output

Use the *record output* command to start output recording sessions within *dbx*. During an output recording session, *dbx* copies its screen output to a file. If the specified file already exists, *dbx* appends to the existing file. You can start and run as many simultaneous *dbx* output recording sessions as you need.

By default, the commands you enter are not copied to the output file; however, if you set the *dbx* variable *$rimode* to a nonzero value, *dbx* also copies the commands you enter.

Each recording session is assigned a number when you begin it. Use this number to reference the recording session with the *unrecord* command described in "Ending a Recording Session" on page 33.

The *record output* command is very useful when the screen output is too large for a single screen (for example, printing a large structure). Within *dbx*, you can use the *playback output* command (described in "Playing Back Output" on page 35) to look at the recorded information. After quitting *dbx*, you can review the output file using any IRIX system text viewing command (such as *vi*(1)).

For example, to record the *dbx* output in a file called *gaffa*, enter:

```
(dbx) record output gaffa
```

To record both the commands and the output, enter:

```
(dbx) set $rimode=1
(dbx) record output gaffa
```

If you omit the filename, *dbx* saves the recorded output in a temporary file in */tmp*. The temporary file is deleted at the end of the *dbx* session. To save output for use after the *dbx* session, you must specify the filename when giving the *record output* command. The name of the temporary file is stored in the *dbx* variable *$defaultout*.

To display the temporary filename, type:

```
(dbx) print $defaultout
```

## Playing Back Output

The *playback output* command displays output saved with the *record output* command. This command works the same as the *cat*(1) command. If you don't specify a filename, *dbx* uses the current temporary file created for the *record output* command.

For example, to display the output stored in the file *script*, enter:

```
(dbx) playback output script
```

## Examining the Record State

The *record* command displays all *record input* and *record output* sessions currently active. For example:

```
(dbx) record
[4] record input /usr/demo/script (12 lines)
[5] record output /tmp/dbxoXa17992 (5 lines)
```

# Executing *dbx* Scripts

You can create *dbx* command scripts using an external editor and then execute these scripts using the *pi* or *playback input* command. This is a convenient method for creating and executing automated test scripts.

You can include comments in your command scripts by using a single pound sign (**#**) to introduce a comment. To include a **#** operator (described in "Operators" on page 38) in a *dbx* script, use two pound signs (for example, **##27**). When *dbx* sees a pound sign in a script file, it interprets all characters between the pound sign and the end of the current line as a comment.

# Examining and Changing Data

This chapter describes how to examine and change data in your program while running it under *dbx*. Topics covered include:

- "Using Expressions"
- "Printing Expressions"
- "Using Data Types and Type Coercion (Casts)"
- "Qualifying Names of Program Elements"
- "Displaying and Changing Program Variables"
- "Displaying and Changing Environment Variables Used by a Program"
- "Using the High-Level Debugging Language duel"
- "Determining Variable Scopes and Fully Qualified Names"
- "Displaying Type Declarations"
- "Examining the Stack"
- "Using Interactive Function Calls"
- "Obtaining Basic Blocks Counts"
- "Accessing C++ Member Variables"

## Using Expressions

Many *dbx* commands accept one or more expressions as arguments. Expressions can consist of constants, *dbx* variables, program variables, and operators. This section discusses operators and constants. "Creating and Removing dbx Variables" on page 23 describes *dbx* variables, and "Displaying and Changing Program Variables" on page 46 describes program variables.

## Operators

In general, *dbx* recognizes most expression operators from C, Fortran 77, and Pascal. *dbx* also provides some of its own operators. Operators follow the C language precedence. You can also use parentheses to explicitly determine the order of evaluation.

Table 5-1 lists the operators provided by *dbx*.

**Table 5-1**          *dbx* Language Independent Operators

| Operator | Description |
|----------|-------------|
| **not** | Unary operator returning false if the operand is true |
| **or** | Binary logical operator returning true if either operand is nonzero |
| **xor** | Binary operator returning the exclusive-OR of its operands |
| **/** | Binary division operator ("**//**" also works for division) |
| **div** | Binary operator that coerces its operands to integers before dividing |
| **mod** | Binary operator returning *op1* modulo *op2*. This is equivalent to the C "**%**" operator |
| **#***exp* | Unary operator returning the address of source line specified by *exp* |
| **"***file***" #***exp* | Unary operator returning the address of source line specified by *exp* in the file specified by *file* |
| *proc* **#***exp* | Unary operator returning the address of source line specified by *exp* in the file containing the procedure *proc* |

The **#** operator takes the line number specified by the expression that follows it and returns the address of that source line. If you precede the **#** operator with a filename enclosed in quotation marks, the **#** operator returns the address of the line number in the file you specify. If you precede the **#** operator with the name of a procedure, *dbx* identifies the source file that contains the procedure and returns the address of the line number in that file.

For example, to print the address of line 27 in the current source file, enter:

```
(dbx) print #27
```

To print the address of line 27 in the source file *foo.c* (assuming that *foo.c* contains source that was used to compile the current object file), enter:

(dbx) **print "foo.c" #27**

To print the address of line 27 in the source file containing the procedure **bar** (assuming that **bar** is a function in the current object file), enter:

(dbx) **print bar #27**

**Note:** A pound sign (**#**) introduces a comment in a *dbx* script file. When *dbx* sees a pound sign in a script file, it interprets all characters between the pound sign and the end of the current line as a comment. See "Executing dbx Scripts" on page 35 for more information on *dbx* script files. To include the **#** operator in a *dbx* script, use two pound signs (for example, **##27**).

Table 5-2 lists the C language operators recognized by *dbx*.

**Table 5-2**         C Language Operators Recognized by *dbx*

| Type | Operators |
|------|-----------|
| Unary | **!   &   +   −   *   sizeof()** |
| Binary | %  <<  >>  ==  <=  >=  !=  <>  &  &&  \|  \|\|  +  −  * /  [ ]  −>  . |

**Note:** C does not allow you to use the **sizeof** operator on bit fields. However, *dbx* allows you to enter expressions using the **sizeof** operator on bit fields; in these cases, *dbx* returns the number of bytes in the data type of bit fields (such as **int** or unsigned **int**). The C language "**^**" exclusive-OR operator is not supported. Use the dbx "**xor**" operator instead.

Table 5-3 lists the Pascal operators recognized by *dbx*.

**Table 5-3**         Pascal Operators Recognized by *dbx*

| Type | Operators |
|------|-----------|
| Unary | **not   ^   +   −** |
| Binary | mod  =  <=  >=  <>  <  >  and  or  +  −  *  /  div  [ ] |

Table 5-4 lists the Fortran 77 and Fortran 90 language operators recognized by *dbx*. Note that *dbx* does not recognize Fortran logical operators (such as **.or.** and **.TRUE.**).

**Table 5-4**        Fortran 77 and Fortran 90 Operators Recognized by *dbx*

| Type | Operators | | | | |
|------|-----------|---|---|---|---|
| Unary | **+** | **–** | | | |
| Binary | **+** | **–** | **\*** | **/** | **%** |

**Note:**  Fortran array subscripts may be in either square brackets, [ ],  or the standard parenthesis, ( ), and the Fortran 90 member selection operator (%) is allowed.

## Constants

You can use both numeric and string constants under *dbx*.

**Note:**  Expressions cannot contain constants defined by **#define** declarations to the C preprocessor.

### Numeric Constants

In numeric expressions, you can use any valid integer or floating point constants. By default, *dbx* assumes that numeric constants are in decimal. You can set the default input base to octal by setting the *dbx* variable *$octin* to a nonzero value. You can set the default input base to hexadecimal by setting the *dbx* variable *$hexin* to a nonzero value. If you set both *$octin* and *$hexin* to nonzero values, *$hexin* takes precedence.

You can override the default input type by prefixing "0x" to indicate a hexadecimal constant, or "0t" to indicate a decimal constant. For example, "0t23" is decimal 23 (which equals hexadecimal 0x17), and "0x2A" is hexadecimal 2A (which equals decimal 42).

By default, *dbx* prints the value of numeric expressions in decimal. You can set the default output base to octal by setting the *dbx* variable *$octints* to a nonzero value. You can set the default output base to hexadecimal by setting the *dbx* variable *$hexints* to a nonzero value. If you set both *$octints* and *$hexints* to nonzero values, *$hexints* takes precedence.

### String Constants

Most *dbx* expressions cannot include string constants. The *print* and *printf* commands are two of the *dbx* commands that accept string constants as arguments. You can also use the *set* command to assign a string value to a *dbx* variable.

Otherwise, string constants are useful only as arguments to functions that you call interactively. See "Using Interactive Function Calls" on page 71 for information on interactive function calls.

You can use either the double-quotation mark (**"**) or the single-forward quotation mark (**'**) to quote strings in *dbx*. In general, *dbx* recognizes the following escape sequences in quoted strings (following the standard C language usage):

```
\\ \n \r \f \b \t \' \" \a
```

Enclosing a character string in back quotation marks (**`**) indicates that the whole string is the name of a program element, not a character-string constant. This is useful, for example, when referring to C++ templates, which include in their names the greater-than (>) and less-than (<) characters. Without back quotation marks, *dbx* would attempt to interpret the characters as operators. For further discussion, see the sections "Qualifying Names of Program Elements" in this chapter and "Referring to C++ Functions" in Chapter 6.

## Printing Expressions

*dbx* provides the following commands for printing values of expressions:

**print** [*exp1* [**,** *exp2***,** ...] ]
> Prints the value(s) of the specified expression(s).

**printd** [*exp1* [**,** *exp2***,** ... ] ]
> Prints the value(s) of the specified expression(s) in decimal. (**pd** is an alias for **printd**. See "Creating and Removing dbx Variables" on page 23 for more information about *dbx* aliases.)

**printo** [*exp1* [**,** *exp2***,** ... ] ]
> Prints the value(s) of the specified expression(s) in octal. (**po** is an alias for **printo**.)

**printx** [*exp1* [**,** *exp2***,** ... ] ]

>Prints the value(s) of the specified expression(s) in hexadecimal. (**px** is an alias for **printx**.)

For displaying information about variables, the *duel* command is a flexible alternative to the *print* command; see "Using the High-Level Debugging Language duel" on page 51.

The variable types are listed in Table 5-5.

**Table 5-5**     Variable Types

| Type | Variable Name | Value |
| --- | --- | --- |
| signed char | sc | 0xff |
| unsigned char | usc | 0xff |
| signed short | ssh | 0xffff |
| unsigned short | ush | 0xffff |

Examples include:

```
(dbx) pd sc
-1
(dbx) pd ssh
-1
(dbx) px sc
0xff
(dbx) px ssh
0xffff
(dbx) pd usc
255
(dbx) pd ush
65535
```

*dbx* always prints the bits in the appropriate type. **pd** is an exception; it expands signed types with sign extension so the decimal value looks correct.

Another example:

```
(dbx) print sc, usc
'\377' '\377'
```

42

If **$hexchars** is set, this command displays '0xff' '0xff'. (This is a change from releases previous to IRIX 5.2. Previously, the **px, po** cases on signed char expanded to 32 bits, so **px sc** printed 0xffffffff.)

If the printed data type is **pointer**, *dbx* uses the format specified in the **$addrfmt** or **$addrfmt64** predefined *dbx* variable (**$addrfmt64** is used on only 64-bit processes).

**printf** *string* [**,** *exp1* [**,** *exp2***,** ... ] ]

>   Print the value(s) of the specified expression(s) in the format specified by the string, *string*. The *printf* command supports all formats of the IRIX **printf** command except "%s." For a list of formats, see the **printf**(3S) reference page.

## Using Data Types and Type Coercion (Casts)

You can use data types for type conversion (casting) by including the name of the data type in parentheses before the expression you want to cast. For example, to convert a character into an integer, use **(int)** to cast the value:

```
(dbx) print (int) 'b'
98
```

To convert an integer into a character, use **(char)** to cast the value:

```
(dbx) print (char) 67
'C'
```

This is standard C language type casting.

## Qualifying Names of Program Elements

You can use the same name for different program elements, such as variables, functions, statement labels, several times in a program. This is convenient and, during program execution, the potential ambiguity presents no problem. For example, you can use a temporary counter named "i" in many different functions. The scope of each variable is local; space is allocated for it when the function is called and freed when the function returns. However, in *dbx* you sometimes need to distinguish occurrences of identical names.

*dbx* allows unambiguous reference to all program elements by using source file and routine names as qualifying information that makes otherwise indistinguishable names unique. To find the fully qualified name of the active version of a name, use the *which* command. To find the fully qualified names of all versions of a name, use the *whereis* command.  Note that if a variable, such as *i*, is used many times, *whereis* can generate many lines of output.

The fully qualified name of a program element allows you not only to refer within a procedure to variables of the same name with different scopes, but to refer unambiguously to program elements outside your current frame or activation stack.

*dbx* qualifies names with the file (also called module), the procedure, a block, or a structure. You can manually specify the full scope of a variable by separating scopes with periods. For example:

```
mrx.main.i
```

In this expression, *i* is the variable name, *main* is a procedure in which it appears, and *mrx* is the source file (omitting the file extension) in which the procedure is defined.

For languages without 'modules,' such as C, C++, and Fortran, the base name of the source file, that is the file name up to the first dot in the name, is taken as a module name. For example, if **b** is a Fortran subroutine in  *t.f*, then *t.b* names the routine.

To illustrate how names are qualified, consider a C program called *test* that contains a function **compare**. In this example, the variable *i* is declared in both the **main** procedure and the **compare** function:

```
int compare ( int );

main( argc, argv )

int argc;
char **argv;
{
   int i;
...
}

int compare ( arg1, arg2 )
{
   int i;
...
}
```

To trace the value of the *i* that appears in the function **compare**, enter:

```
(dbx) trace test.compare.i
```

To print the value of the *i* that appears in the procedure **main**, enter:

```
(dbx) print test.main.i
```

It is possible to have variable scopes in C and C++ that are in unnamed program blocks. *dbx* provides names for these scopes, starting with *__$$blk1* and followed by *__$$blk2*, *__$$blk3*, etc, which it places in the fully qualified name of the variable as it would an explicit scope name. The *whereis* and *which* commands show the names assigned.

*dbx* provides a special name *__aout* for your base executable. So for example, you can use *__aout.main* to refer to a global C function **main** in your executable. You can, of course, also refer to the function using the name of your executable; if your executable is named *myaout*, *myaout.main* also refers to the global C function **main**.

If you wish to refer to a variable that occurs in a DSO, *dbx* uses a naming convention similar to that for variables in your executable. If, for example, **strcpy** is a function from the file *stuff.c* in the library *libc.so.1*, then both *libc.stuff.strcpy* and *libc.strcpy* refer to the function **strcpy**.

In C, **struct**, **union**, and **enum** tags can conflict with other names. From the context, *dbx* usually interprets correctly a reference to one of these tags. However, if *dbx* does not, prefix the tag with the marker *__$T_* to prevent confusion with variables or functions. For example; use *__$T_foo* if you wish to refer to:

```
struct foo { /* ... */ }
```

In ANSI C, statement label names also can conflict with other names. The ambiguity is removed by applying a prefix of *__$L_* to the label name. Thus, for example:

```
int myfoo { int x;   x:  goto x; ++x}
```

If you enter:

```
(dbx) print &x
```

The output is the address of the variable *x*. If you enter:

```
(dbx) print &__$L_x
```

**45**

The output is the address of label *x*. The –32 compiler provides no debugging information on C labels.

To refer to Fortran statement labels you must either use the __*$L*_ prefix or use back quotation marks to force *dbx* to recognize a numerical label as a name. For example, if you have:

```
do 10  i = 1,10
10 continue
```

Both of the following commands lists the address of the label:

(dbx) **print &`10`**

(dbx) **print &__$L_10**

You may have multiple source files with the same name, for example *myfile.c*, that are in different directories. The 'module' name *myfile* may refer to either source file. *dbx* resolves this ambiguity by prefixing the fully qualified file names with a unique, numeric label. The *which* and *whereis* commands show the label used. For example, suppose the main executable has two *myfile.c* source files, then *__aout.myfile* refers to either source file, *__aout._$1_myfile* refers to one of them, and *__aout._$2_myfile* refers to the other.

A leading dot (a period at the beginning of the identifier) tells *dbx* that the first qualifier is not a module (file).

The leading dot is useful when a file and a procedure have the same name. For instance, suppose *mrx.c* contains a function called **mrx**. Further, suppose that *mrx.c* contains a global variable called *mi* and a local variable, also called *mi*. To refer to the global variable, use the qualified form *.mrx.mi*, and to refer to the local variable, use the qualified form *mrx.mrx.mi*.

## Displaying and Changing Program Variables

You can use the value of program variables in *dbx* expressions. You can also change the value of program variables while running your program under *dbx* control.

## Variable Scope

You can access the value of a variable only while it is in scope. The variable is in scope only if the block or procedure with which it is associated is active.

After you start your program, whenever your program executes a block or procedure that contains variables, your program allocates space for those variables and they "come into scope." You may access the values of those variables as long as the block or procedure is active. Once the block or procedure ends, the space for those variables is deallocated and you may no longer access their values.

## Displaying the Value of a Variable

You can display the value of a program variable using the *print*, *printd*, *printf*, *printo*, and *printx* commands and the *pd*, *po*, and *px* aliases described in "Printing Expressions" on page 41. For example, to print the value of the program variable *total*, enter:

```
(dbx) print total
235
```

The *print* command also displays arrays, structures, and other complex data structures. For example, if *message* is a character array (a string), *dbx* prints the string:

```
(dbx) print message
"Press <Return> to continue."
```

As a more complex example, consider a simple linked list stored as an array of elements, each element consisting of a pointer to the next element and an integer value. If the array is named *list*, print the entire array by entering:

```
(dbx) print array
```

*dbx* prints the value of each element in the array:

```
{
    [0] struct list {
        next = (nil)
        value = 1034
    }
    [1] struct list {
        next = 0x10012258
        value = 1031
    }
    [2] struct list {
```

**47**

```
                next = 0x10012270
                value = 1028
        }
        [3] struct list {
                next = 0x10012288
                value = 1025
        }
        [4] struct list {
                next = 0x100122a0
                value = 1022
        }
        [5] struct list {
                next = 0x100122b8
                value = 1019
        }
...
}
```

To print an individual element, enter a command such as:

```
(dbx) print array[5]
struct list {
    next = 0x100122b8
    value = 1019
}
```

If your array consists of simple elements such as integers or floating point values, you can directly examine the contents of memory using the / (examine forward) command described in "Examining Memory and Disassembling Code" on page 109.

Suppose a single-precision floating point array is named *float_vals*. To see the six consecutive elements beginning with the fifth element, enter:

```
(dbx) &float_vals[4] / 6f
10012018:  0.25000000000000000 0.20000000298023224 0.16666699945926666
0.14280000329017639
10012028:  0.12500000000000000 0.11111100018024445
```

You can also print parts of arrays and complex structures with *duel*, a high-level debugging language. For more information, see "Using the High-Level Debugging Language duel."

## Changing the Value of a Variable

The *assign* command changes the value of existing program variables. You can also use the *assign* command to change the value of machine registers, as described in "Changing Register Values" on page 109.

The syntax of the *assign* command is:

**assign** *variable* = *expression*
> Assigns the value of *expression* to the program variable, *variable*.

For example:

```
(dbx) assign x = 27
27
(dbx) assign y = 37.5
37.5
```

If you receive an incompatible types error when you try to assign a value to a pointer, use casts to make the assignment work. For example if *next* is a pointer to a structure of type "element," you can assign *next* a null pointer by entering:

```
(dbx) assign *(int *) (&next) = 0
0
(dbx) assign next = 0
(nil)
(dbx) assign next = (struct list*) 0;
(nil)
```

In this example, nil denotes that the value of the pointer is 0; nil is similar to NULL in the C language.

## Conflicts Between Variable Names and Keywords

When naming variables in your program, avoid using any *dbx* keywords. If you have a variable with the same name as a *dbx* keyword and you attempt to use that variable in a *dbx* command, *dbx* reports a syntax error.

If you do have a program variable with the same name as a *dbx* command, you can force *dbx* to treat it as a variable by enclosing the variable in parentheses. For example, if you try to print the value of a variable named *in* by entering the following command, *dbx* displays an error.

```
(dbx) print in
print in
      ^ syntax error
Suggestion: in is a dbx keyword; a revised command is in history.
Type !16 or !! to execute: print (in)
```

The correct way to display the value of *input* is to enter:

```
(dbx) print (in)
34
```

*dbx* keywords include:

```
all    not
and    or
at     pgrp
div    pid
if     sizeof
in     to
mod    xor
```

## Case Sensitivity in Variable Names

Whether or not *dbx* is case sensitive when it evaluates program variable names depends on the value of the *dbx* variable *$casesense*.

If *$casesense* is 2 (the default), then the language in which the variable was defined is taken into account (for example, C and C++ are case sensitive while Pascal and Fortran are not). If *$casesense* is 1, case is always checked. If *$casesense* is 0, case is always ignored. Note that file (module) names are always case sensitive since they represent UNIX file names.

# Displaying and Changing Environment Variables Used by a Program

You can control the values of environment variables used by a program without affecting the shell. The *dbx* commands *printenv, setenv,* and *unsetenv* control the debugging environment much like their `csh` counterparts control the shell environment. However, they only affect your program while it is being debugged. *dbx* passes the values shown by the *printenv* command to the shell as the environment to use while your program is run by the *run* or *rerun* commands.

The following commands control your program's environment variables:

**printenv**      Prints the list of environment variables affecting the program being debugged.

**setenv**        Same as *printenv*.

**setenv** *VAR*    Sets the environment variable *VAR* to an empty value.

**setenv** *VAR value*
> Sets the environment variable VAR to *value*, where *value* is not a dbx variable.

**setenv** *VAR $var*
> Sets the environment variable *VAR* to *$var*, where *$var* is a *dbx* variable.

**setenv** *VAR "charstring"*
> Sets the environment variable VAR to charstring.

**unsetenv** *VAR* Removes the specified environment variable.

If you attempt to change the environment variables *PAGER* or *EDITOR*, the effect on *dbx* is undefined; the new values may, or may not, affect how *dbx* runs.

## Using the High-Level Debugging Language *duel*

The *duel* language is a high-level debugging language that you can invoke from *dbx*. *duel* is an acronym for *Debugging U (might) Even Like*.

The *duel* language provides the following commands for printing parts of arrays and complex structures:

**duel**         Invokes the *duel* debugging language.

**duel alias**   Shows all current *duel* aliases.

**duel clear**   Deletes all *duel* aliases.

To invoke *duel* from within *dbx*, type:

```
(dbx) duel
```

For example, to print the array elements **x[1]** to **x[10]** that are greater than 5, enter:

```
(dbx) duel x[1..10] >? 5
x[3] = 14
```

```
x[8] = 6
```

The output includes the values `14` and `6`, as well as their symbolic representation `x[3]` and `x[8]`.

## Using *duel* Quick Start

The *duel* language is implemented by adding the *duel* command to *dbx*. All *dbx* commands work as before. The *duel* command, however, is interpreted by *duel*, and its concepts are not understood by other *dbx* commands.

**Note:** This version of *duel* does not understand or allow interactive function calls.

*duel* is based on expressions that return multiple values. The **x..y** operator returns the integers from *x* to *y*; the **x,y** operator returns **x** and then **y**. For example:

```
(dbx) duel (1,9,12..15,22)
```

This command prints `1`, `9`, `12`, `13`, `14`, `15`, and `22`. You can use such expressions wherever a single value is used. For example:

```
(dbx) duel x[1,9,12..15,22]
```

This command prints elements `1`, `9`, `12`, `13`, `14`, `15`, and `22` of the array **x**. *duel* incorporates C operators, and casts C statements as expressions.

The semicolon (**;**) prevents *duel* output. *duel* aliases are defined with **x:=y** and provide an alternative to variable declaration. You can also return **x[i]** instead of using **printf**:

```
(dbx) duel if(x[i:=0..99]<0) x[i]
x[i] = -4
```

The symbolic output `x[i]` can be fixed by surrounding **i** with **{}**. For example:

```
(dbx) duel if(x[i:=0..99]<0) x[{i}]
x[7] = -4
```

The **{}** are like **()**, but force the symbolic evaluation to use **i**'s value, instead of **i**. You can usually avoid this altogether with direct *duel* operators:

```
(dbx) duel x[..100] <? 0
x[7] = -4
```

The **..n** operator is a shorthand for **0..n-1**. For example, ..100 is the same as 0..99. The **x<?y, x==?y, x>=?y,** and so forth, operators compare their left side operand to their right side operand as in C, but return the left side value if the comparison result is true. Otherwise, they look for the next values to compare, without returning anything.

*duel*'s **x.y** and **x->y** allow an expression **y**, evaluated under **x**'s scope:

```
(dbx) duel emp[..100].(if(code>400) (code,name))
emp[46].code = 682
emp[46].name = "Ela"
```

The **if()** expression is evaluated under the scope of each element of emp[], an array of structures. In C language terms, we have to write:

```
for(i = 0; i < 100; i++ ) {
  if(emp[1].code > 400) {
    printf("%d %s\n",emp[i].cond,emp[i].name);
```

A useful alternative to loops is the **x=>y** operator. It returns **y** for each value of **x**, setting the underbar (_) to reference **x**'s value. For example:

```
(dbx) ..100 => if(emp[_].code>400) emp[_].code,emp[_].name
```

Using _ instead of **i** also avoids the need for **{i}**. Finally, the **x-->y** operator expands lists and other data structures. If **head** points to a linked list threaded through the next field, then:

```
(dbx) duel head-->next->data
head->data = 12
head->next->data = 14
head-->next[[2]]->data = 20
head-->next[[3]]->data = 26
```

This produces the data field for each node in the list. **x-->y** returns
**x, x->y, x->y->y, x->y->y->y, ...** until a NULL is found. The symbolic output **x-->y[[n]]** indicates that **->y** was applied *n* times. **x[[y]]** is also the selection operator:

```
(dbx) duel head-->next[[50..60]]->data
```

This example returns the 50th through the 60th elements in the list. The **#/x** operator counts the number of values. For example:

```
(dbx) duel #/( head-->next->data >? 50 )
```

This example counts the number of data elements over 50 on the list. Several other operators, including **x@y**, **x#y**, and active call stack access are described in the "duel Operators."

### *duel* **Operator Summary**

Most *duel* operators have the same precedence as their C counterparts. Table 5-6 lists *duel* operators in decreasing precedence.

**Table 5-6**        *duel* Operator Summary

| Associativity | Operators | Details |
|---|---|---|
| left | {} () [] -> . f() --> | x-->y expands x->y x->y->y ... |
| | x[[y]] x#y x@y | generate x; select, index or stop-at y |
| right | #/ - * & ! ~ ++ -- (cast) | #/x number of x values |
| | frame(n) sizeof(x) | reference to call stack activation level *n* |
| | = | simple assignment |
| left | x/y x*y x%y | multiply, divide, reminder |
| left | x-y  x+y | add, subtract |
| left | x<<y x>>y | shift left/right |
| none | x..y ..y x.. | ..y  0..y-1. x..y return x, x+1...y |
| left | < > <= >= <? >? <=? >=? | x>?y return x if x>y |
| left | == != ==? !=? | x==?y return x if x=y |
| left | x&y | bit-and |
| left | x^y | bit-xor |
| left | x\|y | bit-or |
| left | x&&y &&/x | &&/x are all x values nonzero? |
| left | x\|\|y \|\|/x | \|\|/x is any x value nonzero? |
| right | x? y:z | for each x, if(x) y else z |

**Table 5-6 (continued)**     *duel* Operator Summary

| Associativity | Operators | Details |
|---|---|---|
| right | x:=y | x:=y set x as a *duel* alias to y |
| left | x,y | return x, then y |
| right | x=>y | for each x, evaluate y with x value '_' |
| right | if() else while() for() | C statements cast as operators |
| left | x;y | evaluate and ignore x, return y |
| right | ,, | Fortran multi-dimensional array separator: x[7,,5]. Note the square brackets; x(7,,5) will not work in *duel*. |

## *duel* EXAMPLES

Table 5-7 lists and briefly explains *duel* examples.

**Table 5-7**     *duel* Examples

| Example | Explanation |
|---|---|
| duel (0xff-0x12)*3 | compute simple expression |
| duel (1..10)*(1..10) | display multiplication table |
| duel x[10..20,22,24,40..60] | display x[i] for the selected indexes |
| duel x[9..0] | display x[i] backwards |
| duel x[..100] >? 5 | display x[i] that are greater than 5 |
| duel x[..100] >? 5 <? 10 | display x[i] if 5<x[i]<10 |
| duel x[..100] ==? (6..9) | same as above |
| duel x[0..99]=>if(_>5 && _<10) _ | same as above |
| duel y[x[..100] !=? 0] | display y[x[i]] for each nonzero x[i] |
| duel emp[..50].code | display emp[i].code for i=0 to 49 |
| duel emp[..50].(code,name) | display emp[i].code & emp[i].name |

**Table 5-7 (continued)**    *duel* Examples

| Example | Explanation |
|---|---|
| duel val[..50].(is_dbl? x:y) | display val[i].x or val[i].y depending on val[i].is_dbl. |
| duel val[..50].if(is_dbl) x else y | same as above |
| duel (hash[..1024]!=?0)->scope | hash[i].scope for non-null hash[i] |
| duel x[i:=..100] >? x[i+1] | check if x[i] is not sorted |
| duel x[i:=..100] ==? x[j:=..100]=> if(i<j) x[{i,j}] | check if x has nonunique elements |
| duel if(x[i:=..99] == x[j:=i+1..99]) x[{i,j}] | same as above |
| duel (x[..100] >? 0)[[0]] | the 1st (0th element) positive x[i] |
| duel (x[..100] >? 0)[[2]] | return the 3rd positive x[i] |
| duel (x[..100] >? 0)[[..5]] | return the first 5 positive x[i] |
| duel (x[0..] >? 6)[[0]] | return the first x[i]>6, no limit on i |
| duel argv[0..]@0 | argv[0] argv[1] .. until first null |
| duel x[0..]@20 >? 9 | x[0..n]>9 where n is first x[n]==20 |
| duel emp[0..]@(code==0) | emp[0]..emp[n-1] where emp[n].code==0 |
| duel head-->next->val | val of each element in a linked list |
| duel head-->next[[20]] | the 21st element of a linked list |
| duel *head-->next[[20]] | display above as a struct |
| duel #/head-->next | count elements on a linked list |
| duel x-->y[[#/x-->y - 1]] | last element of a linked list |
| duel x-->y[[#/x-->y - 10..1]] | last 10 elements of a linked list |
| duel head-->next-> if(next) val >? next->val | check if the list is sorted by val |
| duel head-->(next!=?head) | expand cyclic linked list (tail->head) |
| duel head-->(next!=?_) | handle termination with p->next==p |

**Table 5-7 (continued)**     *duel* Examples

| Example | Explanation |
|---|---|
| duel root-->(left,right)->key | expand binary tree, show keys |
| duel root-->(left,right)->( (left!=?0)->key>=?key, (right !=?0 )->key<=?key) | check bin tree sorted by key |
| duel (T mytype) x | convert x to user defined type mytype |
| duel (struct s*) x | convert x to struct s pointer |
| duel if(x) y; else z *ERR* | ';' must be followed by an expression |
| duel {x} y *ERR* | '}' requires ';' if followed by exp |
| fortarray[2..5,, 6,7] | print 2-dimensional Fortran array elements |

## *duel* SEMANTICS

The *duel* semantics are modeled after the Icon programming language. The input consists of expressions that return sequences of values. C statements are cast as expressions, too. Expressions are parsed into abstract syntax trees, which are traversed during evaluation. The evaluation of most nodes (operators) recursively evaluates the next value for each operand, and then applies the operator to produce the next result. Only one value is produced each time, and *duel*'s **eval** function keeps a *state* for each node (backtracking, co-routines, consumer-producer or threads are good metaphors for the evaluation mechanism).

For example, in **(5,3)+6..8**, the evaluation of '**+**' first retrieves the operands **5** and **6**, to compute and return **5+6**. Then **7**, the next right operand is retrieved and **5+7** is returned, followed by **5+8**. Since no other right operand value exists, the next left operand, **3** is fetched. The right operand's computation is restarted returning **6**, and **3+6** is returned. The final return values are **3+7** and **3+8**.

The computation for operators like **x>?y** is similar, but when **x<=y**, the next values are fetched instead of returning a value, forming the basis for an implicit search. Operators like **..** return a sequence of values for each pair of operands.

The *duel* values follow the C semantics. A value is either an **lvalue** (can be used as the left-hand side of assignment), or an **rvalue**. Therefor, objects like arrays can not be directly manipulated. However, operators like **x..y** can accomplish such tasks.

The *duel* types also follow the C semantics, with some important differences. C types are checked statically; *duel* types are checked when operators are applied. For example, **(1,1.0)/2** returns **0** (int) and **0.5** (double); **(x,y).z** returns **x.z** and **y.z** even if **x** and **y** are of different types, as long as they both have a field **z**.

Values and types of symbols are looked up at run-time (using the *dbx* lookup rules).

To avoid this ambiguity, the keyword **T** must precede a user-defined type. For example, if **value** is a **typedef**, C's **(value (\*)())  x** is written in *duel* as
**(T value (\*)()) x**. Types that begin with a reserved keyword don't need **T**. For example, **(struct value\*) x** and **(long \*[5]) y** are accepted. As special cases, **(type)x** and **(type\*)x** are accepted but discouraged (it causes **(printf)("hi")**, which is valid in C, to fail). A side effect is that **sizeof x** must be written as **sizeof(x)**.

## *duel* Operators

The *duel* operators are described below.

**x=y x+y x-y x\*y x/y x%y x^y x│y x&y x<<y x>>y**

**x>y x<y x>=y x<=y x==y x!=y  x[y]**
These binary operators follow their C semantics. For each value of **x**, they are evaluated for every value of **y**. For example, **(5,2)>(4,1)** evaluates as **5>4, 5>1, 2>4, 2>1** returning **1**, **1**, **0**, **1**.

The **y** values are reevaluated for each new value of **x**. For example, **i=4; (4,5)>i++** evaluates as **4>4** and **5>5**. Beware of multiple **y** values in assignment. For example, **x[..3]=(4,6,9)** does not set **x[0]=4**, **x[1]=6**, and **x[2]=9**. It assigns **4**, **6**, and **9** to each element, which has the same effect as **x[..3]=9**. Use **x[i:=..3]=(4,6,9)[[i]]** to achieve the desired effect.

**-x ~x &x \*x !x ++x --x x++ x-- sizeof(x) (type)x**
These unary operators follow their C semantics. They are applied to each value of **x**. The increment and decrement operators require an **lvalue**, so **i:=0 ; i++** produces an error because **i** is a *duel* alias to **0**, an **rvalue**. Parenthesis must be used with **sizeof(x)**. Note that **sizeof x** is not

allowed. Cast to user defined type requires generally requires **T**. For example, **(T val(\*)())x**, but **(val)x** and **(val\*)x** are accepted as special cases.

**x&&y x||y**   These logical operators also follow their C semantics, but have nonintuitive results for multi-valued **x** and **y**. For example, **(1,0,0) ||  (1,0)** returns **1,1,0,1,0** — the right hand-side **(1,0)** is returned for each left-hand side **0**. It is best to use these operators only in single value expressions.

**x? y:z   if(x)y   if(x)y else z**

These expressions return the values of **y** for each nonzero value returned by **x**, and the values of **z** for each zero value returned by **x**. For example, **if(x[..100]==0) y** returns **y** for every **x[i]==0**, not if all **x[i]** are zero (**if(&&/(x[..100]==0))**); **y** does that).

Also, **if(x) y; else z** is illegal. *duel*'s semicolon is an expression separator, not a terminator.

**while(x)y   for(w;x;y)z**

The **while(x)y** expression returns **y** as long as all values of **x** are nonzero. The **for()** expression is similar and both have the expected C semantics. For example, **for(i=0 ; i<100 ; i++) x[i]** is the same as **x[..100]**. Unlike the **if()** expression, **while(x[..100]==0)** continues to execute only if all elements of **x** are zero, that is, the condition is evaluated into a single value using an implicit **&&/x**.

At present, assignments are not supported, so the **for** is of limited utility except to assign aliases.

**x,y   x..y   ..x   x..**

These operators produce multiple values for single value operands. **x,y** returns **x**, then **y**. **x..y** returns the integers from **x** to **y**. When **x>y**, the sequence is returned in descending order, that is, **5..3** returns **5**, **4**, **3**.

The **..x** operator is a shorthand for **0..x-1**. For example, **..3** returns **0**, **1**, **2**. The **x..** operator is a shorthand for **x..maxint**. It returns increasing integer values starting at **x** indefinitely, and should be bounded by **[[n]]** or **@n** operators.

A comma (**,**) retains its precedence level in C. The precedence of **..**  is above **<** and below arithmetic operators, so **0..n-1** and **x==1..9** work as expected.

**x„y**
The **„** operator is very low precedence, is only usable inside the **[]** array operators, and is used to separate the dimension expressions of Fortran multi-dimensional arrays. Note the deviation from Fortran and *dbx* command-line usage; array operators are square brackets, **[ ]**, not parentheses, **( )**

**x<?y x>?y x>=?y x<=?y x!=?y x==?y**
These operators work like their C counterparts but return **x** if the comparison is true. If the comparison is false, the next **(x,y)** value is tried, forming the basis of an implicit search.

**(x) {x} x;y x=>y**

Both **()** and **{}** act as C parenthesis.

The **{}** set the returned symbolic value as the actual value. For example, **if i=5** and **x[5]=3**, then **x[i]** produces the output **x[i] = 3**, **x[{i}]** produces **x[5] = 3**, and **{x[i]}** produces **3**.

The semicolon is an operator. **x;y** evaluates **x**, ignoring the results, then evaluates and returns **y**. For example, **(i:=1..3 ; i+5)** sets **i** to **3** and returns **8**.

The **x=>y** operator evaluates and returns **y** for each value of **x**. For example, **(i:=1..3 => i+5)** returns **6**, **7**, and **8**. The value returned by **x** is also stored implicitly in **_**, which can be used in **y**. For example, **1..5 => z[_][_]** produces **z[1][1]**, **z[2][2]**, and so forth. The symbolic value for **_** is that of the left side value, hence **{_}** is not needed.

Semicolon (**;**) has the lowest precedence, so it must be used inside **()** or **{}** for compound expressions. The precedence of **=>** is just below comma (**,**).

Be aware that **if(a) x; else {y;} z** is illegal; a semicolon is not allowed before **}** or **else** and must be inserted before **z**.

**x->y   x.y**
These expressions work as in C for a symbol **y**. If **y** is an expression, it is evaluated under the scope of **x**. For example, **x.(a+b)** is the same as **x.a+x.b**, if **a** and **b** are fields of **x** (if they are not, they are looked up as local or global variables). **x** may return multiple values of different types. For example, **(u,v).a** returns **u.a** and **v.a**, even if **u** and **v** are different structures.

Also, the value of **x** is available as **_** inside **y**. For example, **x[..100].(if(a) _)** produces **x[i]** for each **x[i].a!=0**. Nested **x.y** are allowed. For example, **u.(v.(a+b))** looks up **a** and **b** first under **v**, then under **u**.

**x:=y**　　The *duel* aliases store a reference to **y** in **x**. Any reference to **x** is then replaced by **y**. If **y** is a constant or an **rvalue**, its value is replaced for **x**. If **y** is an **lvalue** (e.g., a variable), a reference to same **lvalue** is returned. For example, **x:=emp[5] ; x=9** assigns **9** to **emp[5]**.

Aliases retain their values across invocation of the *duel* command. A *duel* alias to a local variable references a stray address when the variable goes out of scope.

The special command *duel clear* delete all the *duel* aliases; *duel alias* shows all current *duel* aliases. Symbols are looked up as *duel* aliases first, so a *duel* alias **x** will hide a local **x**.

The *duel* aliases are separate from *dbx* aliases. Currently, *duel* aliases are shared across all processes.

**x-->y**　　The expansion operator **x-->y** expands a data structure **x** following the **y** links.

It returns **x, x->y, x->y->y**, until a null is found. If **x** is null, no values are produced. If **y** returns multiple values, they are stacked and each is further expanded in a depth-first notion. For example, if **r** is the root of a tree with children **u->childs[..u->nchilds]**, then **u-->(childs[..nchilds])** expands the whole tree. **y** is an arbitrary expression, evaluated exactly like **x->y** (this includes _).

**x@y**　　The expression **x@y** produces the values of **x** until **x.y** is nonzero. For example, **for(i=0 ; x[i].code!= -1 && i<100 ; i++) x[i]** can be written as **x[..100]@(code==-1)**.

The evaluation of **x** is stopped as soon as **y** evaluates to true. **x->y** (or **x=>y**) is used to evaluate **y** when **x** is not a struct or a union. If **y** is a constant, **(_==y)** is used. For example, **s[0..]@0** produces the characters in string **s** up to but not including the terminating null.

**#/x　&&/x　||/x**

These operators return a single *summary* value for all the values returned by **x**. The **#/x** returns the number of values returned by **x**. For example, **#/(x[..100]>?0)** counts the number of positive **x[i]**. The **&&/x** returns **1** if all the values produced by **x** are nonzero, and ||/**x** returns **1** if any of **x**'s values are nonzero. Like in C, the evaluation stops as soon as possible.

For example, ||/**(x[..100]==0)** and **&&/(x[..100]==0)** check if one or all of **x[i]** are zero, respectively.

**x#y x[[y]]**    The operator **x#y** produces the values of **x** and arranges for **y** to be an alias for the index of each value in **x**. It is commonly used with **x-->y** to produce the element's index. For example, **head-->next->val#i=i** assigns each **val** field its element number in the list.

The selection operator **x[[y]]** produces the *y*th result of **x**. If **y** returns multiple value, each select a value of **x**. For example, **(5,7,11,13)[3,0,2]** returns **13**, **5**, and **11** (**13** is the third element, **5** is the 0th element).

Don't use side effects in **x**, since its evaluation can be restarted depending on **y**. For example, after **(x[0..i++])[[3,5]]** the value of **i** is unpredictable.

**Note:** Within a *duel* command, the **#** operator does not have anything to do with line numbers or *dbx* comments.

**frame(n)  frames_no  func.x**

The **frame(n)** for an integer **n** returns a reference to the *n*th frame, or activation level, on the stack (**0** is the inner most function and **frame(frames_no-1)** is **main()**).

Frame values can be compared to function pointers. For example, **frame(3)==myfunc** is true if the fourth frame is a call to **myfunc**, and in scope resolution. For example, **frame(3).x** returns the local variable **x** of the fourth frame.

The **frames_no** is the number of active frames on the stack. For example, **(frames(..frames_no) ==? myfunc).x** displays **x** for all active invocations of **myfunc**. As a special case, **(frames(..frames_no)==?f)[[0]].x** can be written as **f.x** (**x** can be an expression).

## Differences from Other Languages

The following paragraphs describe the differences between *duel*, C, and Fortran languages.

### Differences from C

Both **{}** and **;** are operators, not statements or expression separators. For example, **if(x) y; else {z;} u** is illegal; use **if(x) y else {z} ; u**. Ambiguities require preceding user-defined types (**typedef**) with the keyword **T**. For example, if **value** is a user type, C's

**sizeof(value\*)** is written **sizeof(T value\*)**, except for the casts **(t)x** and **(t\*)x**; **sizeof(x)** requires parenthesis for variable **x**.

### Differences from Fortran

Because the comma (**,**) is used to separate a sequence of values, the usual *dbx* syntax for multi-dimensional array references of **myarr[3,4]** does not mean the same thing to *duel* as it does to *dbx*.

In *duel*, refer to the dimensions of a multi-dimensional Fortran array using **,,** as the dimension separator. In other words, if **myarr** is a two-dimensional array, **myarr[3,,4]** refers to the Fortran array element **myarr(3,4)**.

The base *dbx* syntax for this element remains unchanged. For example, to show that element of **myarr**, use one of the following:

```
(dbx) print myarr[3,4]
(dbx) duel  myarr[3,,4]
```

## Determining Variable Scopes and Fully Qualified Names

The *which* command allows you to determine the scope of a variable. This command is useful for programs that have multiple variables with the same name occurring in different scopes.

The *which* command prints the fully qualified name of the active version of a specified variable. For example, to determine the scope of the variable *i*, enter:

```
(dbx) which i
.foo.foo2.i
```

In the example above, the variable *i* that is currently active is local to the procedure **foo2** that appears in the module *foo* (corresponding to the file *foo.c* in a C language program).

The *which* command also determines the fully qualified name of other program elements, such as procedures or type descriptors, that are submitted as arguments for the command.

The *whereis* command prints the fully qualified names of all versions of the name of any program element. *dbx* searches (a possibly limited part of) your program for all occurrences of the name and returns the fully qualified names. The range of the search is

**63**

determined by the *dbx* variable *$whereisdsolimit*. By default, *$whereisdsolimit* is 1 and only the main executable is checked by *whereis*. To search all objects, set *$whereisdsolimit* to 0. To check just the first *n* objects, set *$whereisdsolimit* to *n*.

## Displaying Type Declarations

The *whatis* command displays the type declaration for a specified variable or procedure in your program.

For example, to display the type declaration for the variable *i*, enter:

```
(dbx) whatis i
int i;
```

The following example illustrates the output of *whatis* for an array of structures:

```
(dbx) whatis array
struct list {
    struct list* next;
    int value;
} array[12];
```

When you provide a procedure name to *whatis*, *dbx* reports the type of the value returned by the procedure and the types of all arguments to the procedure:

```
(dbx) whatis foo
int foo(i)
int i;
(dbx) whatis main
int main(argc, argv)
int argc;
char** argv;
```

## Examining the Stack

Each time your program executes a procedure, the information about where in the program the call was made from is saved on a stack. The stack also contains arguments to the procedure and all of the procedure's local variables. Each procedure on the stack defines an *activation level* or *frame*. Activation levels can also consist of blocks that define local variables within procedures.

The most recently called procedure or block is numbered 0. The next active procedure (the one that called the current procedure) is numbered 1. The last activation level is always the main program block.

The stack determines the scope of many *dbx* commands and expressions. For example, unless you qualify a variable, as described in "Qualifying Names of Program Elements" on page 43, *dbx* assumes that variables you reference are local to the current activation level. If a variable does not appear in the current activation level, *dbx* successively examines previous activation levels in the stack until it finds the referenced variable. The maximum number of activation levels examined is determined by the *dbx* variable *$stacktracelimit*, which has a default value of 100.

## Printing Stack Traces

The *where* command prints stack traces. Stack traces show the current activation levels (procedures) of a program. For example, consider the following stack trace for a program called *test*:

```
(dbx) where
>  0 foo2(i = 5) ["/usr/var/tmp/dbx_examples/test.c":44, 0x1000109c]
   1 foo(i = 4) ["/usr/var/tmp/dbx_examples/test.c":38, 0x1000105c]
   2 main(argc = 1, argv = 0xffffffad78) ["/usr/var/tmp/dbx_examples/test.c":55,
0x10001104]
   3 __start() ["/shamu/lib/libc/libc_64/crt1text.s":137, 0x10000ee4]
```

This program has four activation levels. The most recent, a call of the procedure **foo2**, is numbered 0. The currently selected activation level is 0, indicated by the ">" character.

The stack trace also reports that **foo2** was passed one argument: the value 5 was assigned to the local variable *i*. The trace indicates that the program was stopped at line 44 of the file *test.c*, which translates to machine address 0x1000109c.

The stack trace reports similar information for the next two activation levels in this example. You can see that the function **foo** called **foo2** from line 38 in *test.c*. In turn, **foo** was called by **main** at line 55 of the file *test.c*. Finally, the run-time start-up level was called at line 137 from the file *ctrltext.s*.

If a program is highly recursive, stack traces can get quite long. The *dbx* variable *$stacktracelimit* controls the maximum number of activation levels that appear in a stack trace. In the example above, setting *$stacktracelimit* = 2 before issuing the *where* command reduces the set of reported frames to just levels 0 and 1.

If you compile with **–g0** or with no **–g** option, limited symbols are reported. In cases such as this, where detailed symbolic information is not available, the four hexadecimal values returned represent *dbx*'s guess that the function has four integer arguments. The following example illustrates such a case:

```
(dbx) where
>   0 fooexample(0x300000000, 0x4000000ff, 0x5000000ff, 0x0)
["/usr/var/tmp/dbx_examples/test3.c":10, 0x10000cf8]
    1 main(0x3, 0x4, 0x5, 0x0) ["/usr/var/tmp/dbx_examples/test3.c":5,
0x10000cbc]
    2 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137,
0x10000c64]
(dbx) quit
Process 22582 terminated
int fooexample(int,int,int);
int main()
{
        fooexample(3,4,5);
        return 0;
}
int fooexample(int i, int j, int k)
{
        int x = i + j + 3*k;
        return x;
}
```

The examples below show register values from code compiled without a **–g** option. MIPS1 or MIPS2 code using the 32-bit ABI (for example, on an Indy):

```
(dbx) where
> 0 subr1(0x3, 0x7fffaf14, 0x7fffaf1c, 0x0) ["t.c":3, 0x4009ec]
  1 test(0x3, 0x7fffaf14, 0x7fffaf1c, 0x0) ["t.c":8, 0x400a10]
  2 main(0x1, 0x7fffaf14, 0x7fffaf1c, 0x0) ["t.c":13, 0x400a48]
  3 __start() ["crt1text.s":133, 0x40099c]
```

There are four hexadecimal values displayed in most lines of the code above since the 32-bit MIPS ABI has four integer argument passing registers. No user-useful registers are passed to __start().

MIPS3 or MIPS4 code using the 64-bit ABI (for example, on a Power Challenge):

```
(dbx) where
> 0 subr1(0x3, 0xfffffffaed8, 0xfffffffaee8, 0x0, 0x2f, 0x10, 0x0, 0xfbd82a0)
["/usr/people/doc/debug/t.c":3, 0x10000c9c]
```

```
  1 test(0x3, 0xfffffffaed8, 0xfffffffaee8, 0x0, 0x2f, 0x10, 0x0, 0xfbd82a0)
["/usr/people/doc/debug/t.c":9, 0x10000ce8]
  2 main(0x1000000ff, 0xfffffffaed8, 0xfffffffaee8, 0x0, 0x2f, 0x10, 0x0,
0xfbd82a0) ["/usr/people/doc/debug/t.c":14, 0x10000d2c]
  3 __start() ["/shamu/redwood2/work/irix/lib/libc/libc_64/csu/crt1text.s":137,
0x10000c70]
```

There are eight hexadecimal values displayed in most lines of the code above since the 64-bit MIPS ABI has eight integer argument passing registers. No user-useful registers are passed to __start().

The values listed as *arguments* are the integer argument-passing register values. Typically, only the 0 entry of the stack has those argument values correct. Correctness is not guaranteed because the code generator can overwrite the values, using the registers as temporary variables.

The debugger reports the integer argument-passing registers because this information may be of some value.

For example, for the code samples above, the following code calls subr1():

```
int test(void)
{
     subr1(3);
}
```

This code displays 0x3 as the argument register value. The other registers listed for subr1 contain arbitrary data.

## Moving Within the Stack

The *up* and *down* commands move up and down the activation levels in the stack. These commands are useful when examining a call from one level to another. You can also move up and down the activation stack with the *func* command described in "Moving to a Specified Procedure" on page 69.

The *up* and *down* commands have the following syntax:

**up** [*num*]     Moves up the specified number of activation levels in the stack. The default is one level.

**down** [*num*]   Moves down the specified number of activation levels in the stack. The default is one level.

When you change activation levels, your scope changes. For example, unless you qualify a variable, as described in "Qualifying Names of Program Elements" on page 43, *dbx* assumes that variables you reference are local to the current activation level. Also, *dbx* changes the current source file to the file containing the procedure's source.

Consider examining the stack trace for a program called *test4* and moving up in the activation stack:

```
(dbx) where
>  0 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
   1 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
   2 main(argc = 1, argv = 0xffffffad78)
["/usr/var/tmp/dbx_examples/test4.c":25, 0x10000fa0]
   3 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137, 0x10000f34]
(dbx) print i
5
(dbx) up
foo:  40  r = foo2(i+1);
```

The current activation level is now the procedure **foo**. As indicated in the output, the variable *i* receives the argument passed to **foo** and is therefore local to **foo**. The variable *i* at this activation level is different from the variable *i* in the **foo2** activation level. You can reference the currently active *i* as "i"; whereas you must qualify the reference to the *i* in *foo2*:

```
(dbx) print i
4
(dbx) print foo2.i
<symbol not found>
```

Moving up one more activation level brings you to the **main** procedure:

```
(dbx) up
main:  25  j = foo(j);
(dbx) file
/usr/var/tmp/dbx_examples/test4.c
```

In this example, the source for **main** is in *test4.c*, whereas the source for **foo** and **foo2** is in *foo.c*; therefore, *dbx* changes the current source file when you move up to the **main** activation level.

*dbx* resets the source file when you return to the **foo2** activation level:

```
(dbx) down 2
foo2:  46  printf("foo2 arg is %d\n",i);
(dbx) file
/usr/var/tmp/dbx_examples/foo.c
```

## Moving to a Specified Procedure

The *func* command moves you up or down the activation stack. You can specify the new activation level by providing either a procedure name or an activation level number.

The syntax for the *func* command is:

**func** {*activation_level* | *procedure*}

> Changes the current activation level. If you specify an activation level by number, *dbx* changes to that activation level. If you specify a procedure, *dbx* changes to the activation level of that procedure. If you specify a procedure name and that procedure has called itself recursively, *dbx* changes to the most recently called instance of that procedure.

**func**

> Displays the name of the procedure corresponding to the current activation level.

When you change your activation level, your scope changes. For example, unless you qualify a variable as described in "Qualifying Names of Program Elements" on page 43, *dbx* assumes that variables you reference are local to the current activation level. Also, *dbx* changes the current source file to the one containing the procedure's source and the current line to the first line of the procedure.

You can also give the *func* command the name of a procedure that is not on the activation stack, even when your program is not executing. In this case, *dbx* has no corresponding activation level to make current. However, *dbx* still changes the current source file to the one containing the procedure's source and the current line to the first line of the procedure.

For example, consider the following activation stack:

```
(dbx) where
>  0 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
   1 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
   2 main(argc = 1, argv = 0xffffffad78)
["/usr/var/tmp/dbx_examples/test4.c":25, 0x10000fa0]
```

**69**

```
   3 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137, 0x10000f34]
```

In this case, you can go to the main activation stack by entering:

```
(dbx) func main
main:  25  j = foo(j);
```

This command changes the current activation level to "2" and changes the current source file to *test4.c*.

If you use the *func* command to go to a function that is not on the activation stack, *dbx* changes only the current source file to the one containing the procedure's source and the current line to the first line of the procedure:

```
(dbx) func bar
   3 {
(dbx) file
/usr/var/tmp/dbx_examples/bar.c
```

## Printing Activation Level Information

The *dump* command prints information about the variables in an activation level:

**dump**　　　　　　Prints information about the variables in the current procedure.

**dump** *procedure*　Prints information about the variables in the specified procedure. The procedure must be active. Starts searching for procedure at the current activation level as set by the *up* or *down* command. (See "Moving Within the Stack" on page 67 for more information about the *up* and *down* commands.)

**dump .**　　　　　　Prints information about the variables in all procedures in all activation levels.

For example, executing *dump* while in a function called **foo2** appears as:

```
(dbx) dump
foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
```

To examine the information for the procedure **main**, enter:

```
(dbx) dump main
main(argc = 1, argv = 0xffffffad78) ["/usr/var/tmp/dbx_examples/test4.c":25,
0x10000fa0]
j = 4
```

```
i = 12
r = <expression or syntax error>
a = 0
total = 0
```

To perform a complete dump of the program's active variables, enter:

```
(dbx) dump .
>  0 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
   1 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
r = 0
   2 main(argc = 1, argv = 0xffffffad78)
["/usr/var/tmp/dbx_examples/test4.c":25, 0x10000fa0]
j = 4
i = 12
r = <bad operand>
a = 0
total = 0
```

## Using Interactive Function Calls

You can interactively call a function in your program from *dbx*.

If the function returns a value, you can use that function in a normal *dbx* expression. For example, consider a function **prime** defined in your program that accepts an integer value as an argument, and returns 1 if the value is prime and 0 if it is not. You can call this function interactively and print the results by entering a command such as:

```
(dbx) print prime(7)
1
```

## Using ccall

If your function does not return a value, or if you want to execute a function primarily for its side effects, you can execute the function interactively with the *dbx* command *ccall*:

**ccall** *func***(***arg1, arg2, ... , argn***)**
> This command calls a function with the given arguments. Regardless of the language the function was written in, the call is interpreted as if it were written in C, and normal C calling conventions are used.

**Note:** Structure and union arguments to a function, and structure and union returns from a function, are not supported.

Functions called interactively honor breakpoints. Thus you can debug a function by setting breakpoints and then calling it interactively.

If you perform a stack trace using the *where* command while stopped in a routine executed interactively, *dbx* displays only those activation levels created by your interactive function call. The activation levels for your active program are effectively invisible. For example, a stack trace looks like this during an interactive function call:

```
(dbx) where
>  0 foo2(i = 9) [“/usr/var/tmp/dbx_examples/foo.c”:46, 0x10001214]
   1 foo(i = 8) [“/usr/var/tmp/dbx_examples/foo.c”:40, 0x100011d4]

===== interactive function call =====

   2 foo2(i = 5) [“/usr/var/tmp/dbx_examples/foo.c”:46, 0x10001214]
   3 foo(i = 4) [“/usr/var/tmp/dbx_examples/foo.c”:40, 0x100011d4]
   4 main(argc = 1, argv = 0xffffffad78)
[“/usr/var/tmp/dbx_examples/test4.c”:25, 0x10000fa0]
   5 __start() [“/shamu/lib/libc/libc_64/csu/crt1text.s”:137, 0x10000f34]
```

If you stop execution of an interactively called function, you are responsible for eventually “unstacking” the call and returning from the function call. To unstack a call, you can complete the call using *dbx* commands such as *cont*, *resume*, *next*, or *step* as many times as necessary. If you *run* or *rerun* your program, *dbx* automatically unstacks all interactive function calls.

## Using clearcalls

Another way to unstack an interactive function call is to execute the *clearcalls* command, which clears all stopped interactive calls.

```
(dbx) clearcalls
```

When stopped or faulted within one or more nested interactive calls, the *clearcalls* command removes these calls from the stack and returns the program to its regular callstack. This command is useful when a segmentation fault, infinite loop, or other fatal error is encountered within the interactive call.

When stopped in an interactive call, the call stack displayed by *where* shows the following line at the end of each stack of interactive call instantiation.

```
==== interactive function call ====
```

For example, if the procedure **foo()** is interactively called from **main()**, you see the following stack:

```
>  0 foo2(i = 9) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
   1 foo(i = 8) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]

===== interactive function call =====

   2 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
   3 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
   4 main(argc = 1, argv = 0xffffffad78)
["/usr/var/tmp/dbx_examples/test4.c":25, 0x10000fa0]
   5 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137, 0x10000f34]
```

## Nesting Interactive Function Calls

You can also nest interactive function calls. In other words, if you have one or more breakpoints in a function, and you call that function repeatedly, each interactive call is stacked on top of the previous call. Breakpoints in a function affect all nesting levels, so you cannot have different breakpoints at different nesting levels.

The *where* command shows the entire stack trace from which you can determine the nesting depth. The following example has two nesting levels.

```
(dbx) where
>  0 foo2(i = 17) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
   1 foo(i = 16) ["/usr/var/tmp/src/dbx_examples/foo.c":40, 0x100011d4]

===== interactive function call =====

   2 foo2(i = 9) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
   3 foo(i = 8) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]

===== interactive function call =====

   4 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
   5 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
   6 main(argc = 1, argv = 0xfffffffad78)
["/usr/var/tmp/src/dbx_examples/test4.c":25, 0x10000fa0]
   7 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137,
0x10000f34]
```

To set a conditional breakpoint, for example, type:

```
(dbx) stop in foo if j == 7
Process    0: [3] stop in foo if j==7
```

If **j** is not within the scope of **foo**, then you will receive an error message if you attempt to call **foo** interactively. To prevent this, disable or delete any such breakpoints, conditional commands, or traces before executing the interactive function call.

## Obtaining Basic Blocks Counts

The *dbx* command allows interactive control of a *pixie* instrumented binary.

**pixie clear**   Clear the basic block counts for the current execution.

**pixie write**   Write out the counts file with the current basic block counts. The counts reflect the execution of the program since the *run* command or since the last *pixie clear* command, whichever was more recent.

When you debug a program that has been instrumented by *pixie*, it is often desirable to perform experiments over different code paths and do comparisons of the results. You can do this by capturing the *pixie* basic block counts at any point in the program's execution.

Suppose you want to determine the basic block counts for the section of code between lines 10 and 15 of a given file. Just set breakpoints at the two lines of interest, zero the counts when the first breakpoint is encountered, and then write out the counts file when the second breakpoint is encountered. For example:

```
(dbx) stop at "pix.c":15
Process 0: [3] stop at "pix.c":15
(dbx) stop at "pix.c":20
Process 0: [4] stop at "pix.c":20
(dbx) run
Process 997 (pix.pixie) started
[3] Process 997 (pix.pixie) stopped at [main:15 ,0x400a48  (pixie
0x404570)] 15 first = 12;
(dbx) pixie clear
(dbx) cont
[4] Process 997 (pix.pixie) stopped at [main:20 ,0x400aa8 (pixie
0x404684)] 20 total = multiply(total, 2);
(dbx) pixie write
(dbx) sh prof -pixie prog

------------------------------------------------------------------------
Profile listing generated Tue Feb 14 11:08:46 1995
    with:      prof -pixie prog
------------------------------------------------------------------------
Total cycles  Total Time  Instructions  Cycles/inst     Clock   Target
         53    5.3e-07s            27        1.963   100.0MHz   R4000

        10: Total number of Load Instructions executed.
        40: Total number of bytes loaded by the program.
         3: Total number of Store Instructions executed.
```

```
    12: Total number of bytes stored by the program.

     2: Total number nops executed in branch delay slot.
     0: Total number conditional branches executed.
     0: Total number conditional branches actually taken.
     0: Total number conditional branch likely executed.
     0: Total number conditional branch likely actually taken.

    18: Total cycles waiting for current instr to finish.
    26: Total cycles lost to satisfy scheduling constraints.
     5: Total cycles lost waiting for operands be available.
*------------------------------------------------------------------*
-p[rocedures] using basic-block counts.                            *
 Sorted in descending order by the number of cycles executed in each *
 procedure. Unexecuted procedures are not listed.                  *
*------------------------------------------------------------------*
        cycles(%)  cum %    secs    instrns     calls procedure(file)

        27(50.94)  50.94    0.00         19     1 main(prog:prog.c)
        18(33.96)  84.91    0.00          4     1 multiply(prog:prog.c)
         8(15.09) 100.00    0.00          4     2 add(prog:prog.c)
```

The above example uses the *sh* command to invoke *prof* directly from *dbx*.

For an explanation of the above listing and information on the *prof* and *pixie* commands, see "Using the Performance Tools" in the *Compiling and Performance Tuning Guide* and the *prof*(1) and *pixie*(1) reference pages.

## Accessing C++ Member Variables

Debugging a program written in C++ is somewhat different from debugging programs written in other languages. This section describes features that affect how you access variables. See also the section in the following chapter, "Referring to C++ Functions."

Typically you use standard C++ syntax to access member variables of objects. For example, if the string _name is a member variable of the object *myWindow*, you can print its value by entering:

```
(dbx) print myWindow._name
0x1001dc1c = "MenuWindow"
```

To display a static member variable for a C++ class, you must specify the variable with the class qualifier. For example, to print the value of the static member variable *costPerShare* of the class *CoOp*, enter:

```
(dbx) print CoOp::costPerShare
25.0
```

# Controlling Program Execution

A program typically runs until it exits or encounters an unrecoverable error. You can use *dbx*, however, to stop a program under various conditions, step through your program line by line, stop execution on receiving a signal, and execute conditional commands based on your program's status.

This chapter covers:

- "Setting Breakpoints"
- "Continuing Execution After a Breakpoint"
- "Tracing Program Execution"
- "Writing Conditional Commands"
- "Managing Breakpoints, Traces, and Conditional Commands"
- "Using Signal Processing"
- "Stopping on C++ Exceptions"
- "Stopping at System Calls"
- "Stepping Through Your Program"
- "Starting at a Specified Line"
- "Referring to C++ Functions"

## Setting Breakpoints

*Breakpoints* allow you to stop execution of your program. Breakpoints can be *unconditional*, in which case they always stop your program, or *conditional*, in which case they stop your program only if a test condition that you specify is true.

**Note:** All breakpoints halt program execution *before* executing the line on which they are set. Therefore, if you want to examine the effects of a line of code, you should set the breakpoint on the line of code following the one whose effects you want to study.

Each breakpoint is assigned a number when you create it. Use this number to reference a breakpoint in the various commands provided for manipulating breakpoints (for example, *disable*, *enable*, and *delete*, all described in "Managing Breakpoints, Traces, and Conditional Commands" on page 89).

## Setting Unconditional Breakpoints

To set an unconditional breakpoint, you simply specify the point at which you want to stop program execution, using one of the following forms of the *stop* command:

**stop at**      Sets a breakpoint at the current source line.

**stop at line**   Sets a breakpoint at the specified source line in the current source file.

**stop in procedure**
              Sets a breakpoint to stop execution upon entering the specified procedure. Execution will stop in all inlined or cloned instances of the procedure.

**stop at file:line**
              Sets a breakpoint in the specified file at the specified line.

**Caution:** If your program has multiple source files, be sure to set the breakpoint in the correct file. To do so, you can explicitly set the source file using *dbx*'s *file* command (see "Changing Source Files" on page 18) or you can use the *func* command to go to a source file containing a specified function (see "Moving to a Specified Procedure" on page 69).

## Setting Conditional Breakpoints

An unconditional breakpoint is the simplest type of breakpoint; your program stops every time it reaches a specified place. On the other hand, a *conditional* breakpoint stops your program only if a condition that you specify is true. The two conditions that you can test are:

• Has the value of a variable or other memory location changed?

• Is a test expression true?

## Stopping If a Variable or Memory Location Has Changed

By including a *variable* clause in your *stop* command, you can cause *dbx* to stop if the value of a variable or the contents of a memory location has changed.

If you provide only a variable name in your variable clause, the breakpoint stops your program if the value of the variable has changed since the last time *dbx* checked it. If instead of a variable name, you provide an expression of type pointer, *dbx* checks the data pointed to. If the data pointed to is a structure, *dbx* checks that structure. If you provide an expression that's not of type pointer, *dbx* evaluates the expression and uses the result as an address in memory. The breakpoint stops your program if the contents of the memory location (32 bits) has changed since the last time *dbx* checked it.

The points at which *dbx* checks the value of a variable or memory location depend on the command that you use to set the breakpoint:

**stop** [**expression** | **variable**]

Inspects the value before executing each source line. If the expression is of type pointer, look at the data pointed to and watch until it changes.

If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address). For example, consider the command:

```
stop (struct s*) 0x12345678
```

This command checks the contents of the structure located at `0x12345678`.

**stop** [**expression** | **variable**] **at line**

Inspects the value at the given source line. Stops if the value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stop** [**expression** | **variable**] **in procedure**

Inspects the value at every source line within a given procedure. Stops if the value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes.

If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

## Using Fast Data Breakpoints

You can use fast watchpoints, also known as data breakpoints, with the *stop* command. A fast watchpoint watches a specified variable or memory address without severely impacting the performance of the program being debugged.

InIRIX 4 and earlier versions of *dbx*, the debugger had to single-step the process being debugged and check if the value of a variable had changed after each instruction. With fast watchpoints, the debugger uses a hardware virtual memory **write protect mechanism** to allow the program to run freely until the variable being watched changes. The program being debugged stops only when the virtual memory page containing the variable is written to. If the value of the variable being watched does not changed, *dbx* continues the execution of the process. If a **write** modifies a watched variable, *dbx* notifies you of the change.

Consider a small program that contains a global variable called `global`:

```
stop global
```

This command causes the program to stop if the value of the variable `global` changes. The program runs virtually at full speed until `global` gets assigned a new value. Similarly, consider the command:

```
stop 0x100100
```

This command stops when the 32- bit integer residing at address `0x100100` is modified, and runs at nearly full speed until the value changes. This form of the *stop* command is useful for watching the contents of anonymous memory, such as the memory returned by *malloc()*.

*dbx* still needs to use the single-step approach if the *stop* command contains an expression to watch, such as in `stop if global == 1`. The performance of the debugged program can be greatly enhanced by including a variable to watch in the *stop* command.

For example, the previous *stop* command can be expressed equivalently as `stop global if global == 1`. This instructs the debugger to check only the expression `global == 1` if the value of `global` changes. For situations where the expression does not depend

upon a particular variable getting modified such as `stop if global == x * 3`, the single-step approach is the only way to achieve the desired behavior.

## Stopping If a Test Expression Is True

By including a test clause in your *stop* command, you can cause *dbx* to stop if the value of an expression is true. You can use any valid numerical expression as a test. If the result of the expression is nonzero, the expression is true and the test is successful.

The point at which *dbx* evaluates the test expression depends on the command that you use to set the breakpoint:

**`stop if` expression**

> Evaluates the expression before executing each source line. Note that execution is very slow if you choose this type of conditional breakpoint.

**`stop at` line `if` expression**

> Evaluates the expression at the given line.

**`stop in` procedure `if` expression**

> Evaluates the expression at every source line within a given procedure.

## Conditional Breakpoints Combining Variable and Test Clauses

You can create conditional breakpoints that combine both variable and test clauses. In these cases, the overall test evaluates to true only if both clauses are true.

The following forms of the *stop* command combine both the variable and test clauses:

**`stop` [expression1 | variable] `if` expression2**

> Tests both conditions before executing each source line. Stops if both conditions are true.
>
> If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**`stop` [expression1 | variable] `at` line `if` expression2**

> Tests both conditions at the given source line. Stops if both conditions are true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stop** [**expression1**|**variable**] **in procedure if expression2**

Tests both conditions at every source line within a given procedure. Stops if both conditions are true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

## Continuing Execution After a Breakpoint

The *cont* command allows you to continue execution after any type of breakpoint. In its simplest form, program execution continues until the end of the program or until another breakpoint is reached. You can also tell *dbx* to continue your program until it reaches a given line or procedure; this is similar to setting a temporary, "one-shot" breakpoint and then continuing.

The syntax of the *cont* command is:

**cont**          Continues execution with the current line.

**cont** {**at** | **to**} **line**

Sets a temporary breakpoint at the specified source line, then resumes execution with the current line. When your program reaches the breakpoint at **line**, *dbx* stops your program and deletes the temporary breakpoint. The keywords **at** and **to** are equivalent.

**cont in procedure**

Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current line. When your program reaches the breakpoint in **procedure**, *dbx* stops your program and deletes the temporary breakpoint.

If your program stopped because *dbx* caught a signal intended for your program, then *dbx* will send that signal to your program when you continue execution. You can also explicitly send a signal to your program when you continue execution. Sending signals to your program upon continuation is discussed in "Continuing After Catching a Signal" on page 93.

When you debug multiprocess programs, the *resume* command can be more helpful than the *cont* command. Refer to "Resuming a Suspended Process" on page 127 for more information about the *resume* command.

## Tracing Program Execution

The *trace* command allows you to observe the progress of your program as it executes. With it, you can print:

- values of variables at specific points in your program or whenever variables change value

- parameters passed to and values returned from functions

Each trace is assigned a number when you create it. Use this number to reference the trace in the various commands provided for manipulating traces (for example, *disable*, *enable*, and *delete*, all described in "Managing Breakpoints, Traces, and Conditional Commands" on page 89).

The syntax of the *trace* command is:

**trace variable**  Whenever the specified variable changes, *dbx* prints the old and new values of that variable.

**trace procedure**

Prints the values of the parameters passed to the specified procedure whenever your program calls it. Upon return, *dbx* prints the return value.

**trace [expression | variable] at line**

Whenever your program reaches the specified line, *dbx* prints the value of the variable if its value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**trace [expression | variable] in procedure**

Whenever the variable changes within the procedure, *dbx* prints the old and new values of that variable.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**trace** [expression1 | **variable**] **at line if** expression2

Prints the value of the variable (if changed) whenever your program reaches the specified line and the given expression is true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**trace** [expression1 | **variable**] **in procedure if** expression2

Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable, if the given expression is true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

To examine the parameters passed to and values returned from a function, you can trace that function. For example, if the function name is **foo**, set the trace by entering:

```
(dbx) trace foo
```

When you execute your program, *dbx* prints the values of the parameters passed to **foo** whenever your program calls it. Upon return from **foo**, *dbx* prints the return value:

```
(dbx) run
[3] calling foo(text = 0x10000484 = "Processing...\n", i = 4) from
function main
[4] foo returning -1 from foo
```

In the example shown above, **foo** receives two parameters: a character string variable named *text* containing the value "Processing...\n" and an integer variable named *i* containing the value 4. The trace also indicates that **foo** returns a value of -1.

You can also examine a variable as it changes values. For example, you can monitor the value of a string variable named *curarg* as you use it to process an argument list. To set the trace, enter:

```
(dbx) trace curarg
Process 2395: [6] trace .test.main.curarg in main
```

When you set a trace on a variable, examine the confirmation that *dbx* prints. If you use the same variable name in multiple functions in your program, *dbx* may not set the trace on the variable that you want. If *dbx* sets the trace on an incorrect variable, delete the trace and set a new trace using a qualified variable format as described in "Qualifying Names of Program Elements" on page 43. For more information on deleting traces, see "Deleting Breakpoints, Traces, and Conditional Commands" on page 91.

So, in this example, if you use the variable *curarg* in both **main** and a function called **arg_process**, and you want to trace the *curarg* in **arg_process**, first delete this trace and then set a new trace:

```
(dbx) delete 6
(dbx) trace arg_process.curarg
Process 2395: [7] trace .test.arg_process.curarg in arg_process
```

When you execute your program, whenever *curarg* changes, *dbx* prints its old and new values:

```
(dbx) run
[7] curarg changed before [arg_process: line 53]:
                new value = (nil);
[7] curarg changed before [arg_process: line 86]:
                old value = 0;
                new value = 0x7fffc7e5 = "-i";
[7] curarg changed before [arg_process: line 86]:
                old value = 2147469285;
                new value = 0x7fffc7eb = "names.out";
[7] curarg changed before [arg_process: line 86]:
                old value = 2147469291;
                new value = 0x7fffc7f5 = "names.in";
```

## Writing Conditional Commands

A conditional command created with the *when* command is similar to a breakpoint set with the *stop* command, except that rather than stopping when certain conditions are met, *dbx* executes a list of commands. The command list can consist of any *dbx* commands, separated by semicolons if you include more than one command in the command list. Additionally, you can use the keyword *stop* in the command list to stop execution, just like a breakpoint.

Each conditional command is assigned a number when you create it. You use this number to reference the conditional command in the various commands provided for

manipulating conditional commands (for example, *disable*, *enable*, and *delete*, all described in "Managing Breakpoints, Traces, and Conditional Commands" on page 89).

The syntax of the *when* command is:

**when** [**expression | variable**] **{command-list}**

> Inspects the value before executing each source line. If it has changed, executes the command list.

> If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**when** [**expression | variable**] **at line {command-list}**

> Inspects the value at the given source line. If it has changed, executes the command list.

> If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**when** [**expression | variable**] **in procedure {command-list}**

> Inspects the value at every source line within a given procedure. If it has changed, executes the command list.

> If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**when if expression {command-list}**

> Evaluates the expression before executing each source line. If it is true, executes the command list. Note that execution is slow if you choose this type of conditional command execution.

**when at line if expression {command-list}**

> Evaluates the expression at the given line. If it is true, executes the command list.

**when in procedure if expression {command-list}**

> Evaluates the expression at every source line within a given procedure. If it is true, executes the command list.

**when** [expression1 | variable] **if** expression2 {**command-list**}
>
>Checks if the value of the variable has changed. If it has changed and the expression is true, executes the command list.
>
>If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**when** [expression1 | variable] **at line if** expression2 {**command-list**}
>
>Checks if the value of the variable has changed each time the line is executed. If the value has changed and the expression is true, executes the command list.
>
>If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**when** [expression1 | variable] **in procedure if** expression2 {**command-list**}
>
>Checks if the value of variable has changed at each source line of the given procedure. If the value has changed and the expression is true, executes the command list.
>
>If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

## Managing Breakpoints, Traces, and Conditional Commands

*dbx* provides commands that allow you to disable, enable, delete, and examine the status of the breakpoints, traces, and conditional commands that you set in your programs.

Each breakpoint, trace, and conditional command is assigned a number when you create it. Use these numbers as identifiers in the various commands provided for manipulating these debugging controls.

### Listing Breakpoints, Traces, and Conditional Commands

The *status* command lists all of the breakpoints, traces, and conditional commands that you have set and indicates whether they are enabled or disabled.

For example, consider executing the following commands while debugging a program called *test*:

```
(dbx) stop in foo
Process     0: [3] stop in foo
(dbx) r
Process 22631 (test) started
[3] Process 22631 (test) stopped at [foo:38 ,0x10001050]
  38  r = foo2(i+1);
(dbx) trace total
Process 22631: [4] trace total in foo
(dbx) when at 60 {print i,j }
Process 22631: [5] when at "/usr/var/tmp/dbx_examples/test.c":60 {
print i, j }
```

If you enter *status*, you see the following:

```
(dbx) status
Process 22631: [3] stop in foo
Process 22631: [4] trace total in foo
Process 22631: [5] when at "/usr/var/tmp/dbx_examples/test.c":60 {
print i, j }
```

## Disabling Breakpoints, Traces, and Conditional Commands

The *disable* command allows you to temporarily disable a breakpoint, trace, or conditional command so that it is inoperative and has no effect on program execution. *dbx* remembers all information about a disabled breakpoint, trace, or conditional command, and you may enable it using the *enable* command described in "Enabling Breakpoints, Traces, and Conditional Commands" on page 91.

The syntax of the *disable* command is:

**disable item** [, **item** ... ]

> Disables the specified breakpoint(s), trace(s), or conditional command(s). This command has no effect if the item you specify is already disabled.

For example, to disable the conditional command set in "Listing Breakpoints, Traces, and Conditional Commands" on page 89 enter:

```
(dbx) disable 4
```

If you enter *status*, you see the following:

```
(dbx) status
Process 22631: [3] stop in foo
Process 22631: [4] (disabled) trace total in foo
Process 22631: [5] when at "/usr/var/tmp/dbx_examples/test.c":60 {
print i, j
```

## Enabling Breakpoints, Traces, and Conditional Commands

The *enable* command reverses the effects of a *disable* command: The breakpoint, trace, or conditional command that you specify is enabled and once again affects the execution of your program. The syntax of the *enable* command is:

**enable item** [, **item** ...]
> Enables the specified breakpoint(s), trace(s), or conditional command(s).

For example, to enable the conditional command disabled in "Disabling Breakpoints, Traces, and Conditional Commands" on page 90, enter:

```
(dbx) enable 4
```

Executing the *status* command shows that the condition command is now enabled:

```
(dbx) status
Process 22631: [3] stop in foo
Process 22631: [4] trace total in foo
Process 22631: [5] when at "/usr/var/tmp/dbx_examples/test.c":60 {
print i, j
```

## Deleting Breakpoints, Traces, and Conditional Commands

The *delete* command allows you to delete breakpoints, traces, and conditional commands:

**delete** {**item** [, **item** ...] | **all**}
> Deletes the item or items specified. If you use the keyword *all* instead of listing individual items, *dbx* deletes all breakpoints, traces, and conditional commands.

For example, to delete the breakpoint and trace set in "Listing Breakpoints, Traces, and Conditional Commands" on page 89, enter:

```
(dbx) delete 3, 4
```

If you enter *status*, you see the following:

```
(dbx) status
Process 22631: [5] when at "/usr/var/tmp/dbx_examples/test.c":60 {
print i, j }
```

To delete all breakpoints, traces, and conditional commands, enter:

```
(dbx) delete all
```

## Using Signal Processing

*dbx* can detect any signals sent to your program while it is running and, at your option, stop the program.

### Catching and Ignoring Signals

With the *catch* command, you can instruct *dbx* to stop your program when it receives any specified signal. The *ignore* command undoes the effects of a *catch* command.

The *catch* and *ignore* commands have the following syntax:

**catch** {**signal** | **all**}
          Instructs *dbx* to stop your program whenever it receives the specified signal. If you use the keyword *all* rather than giving a specific signal, *dbx* catches all signals.

**ignore** {**signal** | **all**}
          Instructs *dbx* to ignore the specified signal. All ignored signals are passed to your program normally. If you use the keyword *all* rather than giving a specific signal, *dbx* ignores all signals.

**catch**          Prints a list of all signals caught.

**ignore**        Prints a list of all signals ignored.

You can use the signal names and numbers as listed in the **signal**(2) reference page. You can also abbreviate the signal names by omitting the "SIG" portion of the name. You can use uppercase or lowercase for the signal names.

**Note:** Because "int" (in lowercase) is a *dbx* keyword, you cannot use it as an abbreviation for the SIGINT signal. You must use uppercase ("INT"), the full signal name ("SIGINT" or "sigint"), or the signal number ("2"). SIGINT is the only signal name with such a restriction.

If you instruct *dbx* to catch a signal, whenever that signal is directed to your program, *dbx* intercepts it and stops your program. Your program does not see this signal until you continue your program with the *cont* command. If your program has a handler for the signal, the signal is then passed to the program. If there is no handler for the signal, the program does not see the signal. You can suppress passing the signal to the program's signal handler by issuing a *step* or *next* command, rather than *cont*.

If you issue a SIGINT signal at the keyboard (usually by pressing `<Ctrl-C>`) while you are running an application under *dbx*, what happens depends on the circumstances:

• If the process is in the same IRIX process group as *dbx*, the interrupt signal is sent to both *dbx* and the process. Both *dbx* and the process stop running. You are left at the *dbx* command line.

• If the process was added with `addproc`, `dbx -P`, or `dbx -p`, it is not in the same IRIX process group as *dbx*. In this case, the signal interrupt is sent to *dbx* but not to the process. *dbx* stops running, but the process continues to run. Use the *showproc* command to see whether the process is still running. Then use the *suspend* command to stop the process.

## Continuing After Catching a Signal

The *cont* command allows you to continue execution after catching a signal. You can also use the *cont* command to specify a different signal to send to your program than the one that *dbx* caught. Using the same syntax, you can also send a signal to your program when you continue, even if your program did not stop because of a caught signal.

Use the following forms of the *cont* command when handling signals. In each case, if you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution:

**cont** [**signal**]   Continues execution with the current line and sends the specified signal to your program.

**93**

**cont** [**signal**] {`at` | `to`} **line**

> Sets a temporary breakpoint at the specified source line, then resumes execution with the current line and sends the specified signal to your program.

**cont** [**signal**] **`in` procedure**

> Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current line and sends the specified signal to your program.

For example, if your program stopped because *dbx* caught a SIGINT signal, *dbx* will automatically send that signal to your program, if you enter:

```
(dbx) cont
```

Suppose you have a procedure called **alarm_handler** to handle an alarm signal sent to your program. If you want to test this procedure by single-stepping through it, you can execute the following command:

```
(dbx) cont SIGALRM in alarm_handler
```

This sets a temporary breakpoint to stop your program upon entering **alarm_handler**, continues execution of your program, and sends a SIGALRM signal to your program. Your program then enters the **alarm_handler** procedure and stops. You can then single-step through the procedure and observe execution.

## Stopping on C++ Exceptions

The *intercept* command stops program execution on C++ exceptions. You can append a conditional expression to an *intercept* command by using the **`if`** clause. However, the context of an *intercept* break is not that of the throw; the context is the exception handling code of the C++ runtime library. Hence, only global variables have unambiguous interpretation in the **`if`** clause. To refer to a variable whose scope is that of the throw, use the fully qualified name for the variable.

The syntax of the *intercept* command is:

**intercept** {**all** | *item*}

> Stops on all C++ exceptions, or exceptions that throw the base type *item*.

**intercept unexpected** {[**all**] | [*item* [, *item*] ]}

> Stops on all C++ exceptions that have either no handler or are caught by an "unexpected" handler. You may omit **all**. If you specify *item*, stops on exceptions that throw the base type *item*.

**intercept ... if expression**

> You can append the **if** clause to all *intercept* commands. Your program stops only if **expression** is non-zero. Note that the context for evaluation of **expression** is the C++ runtime library, not that of the throw, so use global variables or fully qualified names in **expression**.

*bx* is an alias for *intercept* and *unx* is an alias for *unexpected*.

The following program example illustrates the **if** clause with the *intercept* command:

```
int global = 1;

main (){
    int local = 2;
    try {
        throw -1;
    }
    catch (int key) {
        printf ("exception: %d.\n", key);
    }
}
```

To set a break with a condition on the global variable, enter:

(dbx) **intercept int if global != 0**

Use a fully qualified name to set a break with a condition on the local variable:

(dbx) **intercept int if main.local != 0**

Do not include complex expressions involving operators such as **\*** and **&** in your type specification for an *intercept* command. Note, however, that if you use the *intercept* command with a specific base type, you will also stop your program on throws of pointer, reference, const and volatile types. For example:

```
(dbx) bx char
```

Your program will stop on throws of type **char**, **char \***, **char&**, **const char&, volatile char\***, and so forth.

Like all other break points, **pgrp** or a **pid** clause can be appended to an *intercept* command. For example:

```
(dbx)intercept int pid 12345
```

```
(dbx)intercept char pgrp
```

## Stopping at System Calls

Because system calls are part of the operating system and their source is generally not available for debugging purposes, you cannot set breakpoints in system calls using the same method that you use for your program's procedures. Instead, *dbx* provides the *syscall* command to allow you to stop your program when it executes system calls. With the *syscall* command you can catch (breakpoint) system calls either at the entry to the system call or at the return from the system call.

The syntax of the *syscall* command is:

**syscall catch** {**call** | **return**} {**system_call** | **all**}
> Sets a breakpoint to stop execution upon entering (**call**) or returning from (**return**) the specified system call. Note that you can set *dbx* to catch both the call and the return of a system call.
>
> If you use the keyword **all** rather than giving a specific system call, *dbx* catches all system calls.

**syscall ignore** {**call** | **return**} {**system_call** | **all**}
> Clears the breakpoint to stop execution upon entering (**call**) or returning from (**return**) the specified system call.
>
> If you use the keyword **all** rather than giving a specific system call, *dbx* clears the breakpoints to stop execution upon entering (*call*) or returning from (**return**) all system calls.

**syscall catch** [{**call** | **return**}]
> Prints a list of all system calls caught upon entry (**call**) or return (**return**). If you provide neither the **call** nor **return** keyword, *dbx* lists all system calls that are caught.

**syscall ignore** [{**call** | **return**}]

> Prints a list of all system calls not caught upon entry (**call**) or return (**return**). If you provide neither the **call** nor **return** keyword, *dbx* lists all system calls that are ignored.

**syscall**        Prints a summary of the catch and ignore status of all system calls. The summary is divided into four sections: 1) caught at call, 2) caught at return, 3) ignored at call, and 4) ignored at return.

**Note:** The **fork** and **sproc** system calls are treated specially as they invoke new processes. The returns from these system calls are controlled by the *dbx* variables *$promptonfork* and *$mp_program*, not by *syscall*. This is discussed in "Handling fork System Calls" on page 130 and "Handling sproc System Calls and Process Group Debugging" on page 132. The **execv** and **execve** system calls are treated specially as they change a process into a new program. For more information, see "Handling exec System Calls" on page 131.

The system calls are listed in */usr/include/sys.s*. *dbx* ignores the case of the system call names in all *syscall* commands; therefore, you can use uppercase or lowercase in these commands.

A particularly useful setting is:

(dbx) **syscall catch call exit**

This stops your program upon entry to **exit**. With your program stopped, you can do a stack trace before the termination to see why **exit** was called.

## Stepping Through Your Program

*Stepping* is a process of executing your program for a fixed number of lines and then automatically returning control to *dbx*. *dbx* provides two commands for stepping through lines of code: *step* and *next*.

For both *step* and *next*, *dbx* counts only those source lines that actually contain code; for the purposes of stepping, *dbx* ignores blank lines and lines consisting solely of comments.

The *next* and *step* commands differ in their treatment of procedure calls. When *step* encounters a procedure call, it usually "steps into" the procedure and continues stepping through the procedure (counting each line of source). On the other hand, when *next* encounters a procedure call, it "steps over" the procedure—executing it without

stopping but not counting lines in the procedure—and continues stepping through the current procedure.

The following code fragment illustrates the difference between *step* and *next*:

```
 55  foo( arg1, arg2 )
 56  int arg1, arg2;
 57  {
 58        if ( arg1 < arg2 ) {
...        ...
 78        return( 0 );
 79  }
...
211  x = foo( i, j );
212  y = 2 * x;
```

In this example, if at line 211 you execute a *step* command to advance one line, *dbx* allows the process to proceed to line 58 (the first code line of the **foo** procedure). However, if you execute a *next* command, *dbx* executes line 211—calling **foo**—and advances the process to line 212.

## Stepping Using the *step* Command

The format of the *step* command is:

**step** [**integer**]   Executes the specified number of lines of source code, stepping into procedures. If you do not provide an argument, *step* executes one line. If *step* encounters any breakpoints, it immediately stops execution.

By default, *step* steps into only those procedures that are compiled with the debugging options **-g**, **-g2**, or **-g3**  for which line numbers are available in the symbol table. Note that this does not include standard library routines because they are not compiled using debugging options.

You can modify this behavior, even force *dbx* to step into procedures not compiled with full debugging information, by changing the value of the *dbx* variable *$stepintoall*.

Table 6-1 summarizes how the value of *$stepintoall* affects the *step* command.

**Table 6-1**     Effect of *$stepintoall* Variable on the *step* Command

| *$stepintoall* value | Effect on *step* Command |
| --- | --- |
| 0 (default) | Steps into all procedures that are compiled with debugging options **–g**, **-g2**, or **-g3** for which line numbers are available in the symbol table. |
| 1 | In addition to the above procedures, steps into any procedures for which a source file can be found. Note that when you debug a source file compiled without symbols or compiled with optimization, the line numbers may jump erratically. |
| 2 | Steps into all procedures. Note that if *dbx* cannot locate a source file, then it cannot display source lines as you step through a procedure. |

If your program has DSOs, set the environment variable *LD_BIND_NOW* to 1 before you run your program. This will force complete run-time linking. Otherwise, you can accidentally step into the run-time-linker, **rld(1)**, which becomes part of your program at run time. Useful stack traces are then impossible. To avoid this situation, enter the following before the *run* command:

```
(dbx) setenv LD_BIND_NOW 1
```

### Stepping Using the *next* Command

The format of the *next* command is:

**next** [**integer**]  Executes the specified number of lines of source code, stepping over procedures. If you do not provide an argument, *next* executes one line. If *next* encounters any breakpoints, even in procedures that it steps over, it immediately stops execution.

### Using the *return* Command

If you step into a procedure and then decide you don't want to step through the rest of it, use *return* to finish executing the procedure and return to the calling procedure.

The format of the *return* command is:

**return**  Continues execution until control returns to the procedure that invoked the *return* command.

**return** *proc*  Continues execution until control returns to the named procedure. Execution continues, unless stopped by a breakpoint, until the latest invocation of the procedure named by *proc* at the time the command was issued is reached. Execution doesn't stop at subsequent invocations of the same procedure. The search for the frame to return to starts with the previous frame, because the current frame is skipped in looking for a frame whose name matches *proc*. If execution is stopped for any reason, this command is cancelled.

## Starting at a Specified Line

When you continue your program, you typically do so at the place where it stopped using the *cont* command. However, you can also force your program to continue at a different address by using the *goto* command:

**goto** **line**  Begins execution at the specified line. You may not use the *goto* command to resume execution with a line outside of the current procedure.

## Referring to C++ Functions

As discussed in the section "Accessing C++ Member Variables" in Chapter 5, debugging a program written in C++ has some unique features. This section discusses setting breakpoints in C++ functions.

For the purpose of *dbx* debugging, functions in C++ programs fall into three general categories:

Member functions

> Refers to member functions using the syntax *classname::functionname*. For example, refers to the member function **foo** in the class **Window** as **Window::foo**.

Global C++ functions

> Refers to global functions using the syntax *::functionname*. For example, refers to the global function **foo** as **::foo**.

Non-C++ functions

> Refers to non-C++ functions using the syntax *functionname*. For example, refers to the function **printf** as **printf**.

When using *dbx* with C++, you cannot distinguish between overloaded functions. For example, consider two functions:

```
print(int);
print(float);
```

The following command sets a breakpoint in both functions:

```
(dbx) stop in ::print
```

The following example illustrates various possibilities:

```
#include <stdio.h>
class foo {
   int n;
   public:
   foo() {n = 0;}
   foo(int x);
   int bar();
   int bar(int);
};

int foo:: bar()
```

```
{
   return n;
}

int foo:: bar(int x)
{
   return n + x;
}

foo::foo(int x)
{
   n = x;
}

int square(int x)
{
   return x*x;
}

main()
{
   foo a;
   foo b = 11;
   int x = a.bar();
   int y = b.bar(x) + square(x);
   printf("y = %d\n", y);
}
```

If you enter:

(dbx) **stop in foo::foo**

*dbx* stops execution in the constructor for the variable *b*; *dbx* also stops in the constructor for the variable *a*.

If you enter:

(dbx) **stop in foo::bar**

*dbx* stops execution both when **a.bar** is called and when **b.bar** is called, because *dbx* is unable to distinguish between the overloaded functions.

To stop in **square**, enter:

(dbx) **stop in ::square**

To stop in **printf** (a C function), enter:

```
(dbx) stop in printf
```

To set breakpoints in a specific function from a C++ template, the name of the function must be in back quotation marks to force *dbx* to interpret the entire character string as the name of the function. Otherwise the < and > characters in the template name are interpreted by *dbx* as operators.

*dbx* sets breakpoints in all instantiations of the template if you do not use back quotation marks and simply leave out the template's type-argument list, that is leave out the two characters < and > and the characters included between them.

The following code illustrates these points:

```
template <class T>  myclass {
myclass() { /*... */ }
~myclass() { /*... */ }
myfunc(T) { /* ... */ }};
```

To set a breakpoint only in the **<int>** template function for **myfunc** enter:

```
(dbx) stop in `myclass<int>::myfunc`
```

To set breakpoints in all functions **myfunc** for all instantiations of the template class enter:

```
(dbx) stop in  myclass::myfunc
```

# Debugging Machine Language Code

This chapter explains how to debug machine language code by:

*   "Examining and Changing Register Values"
*   "Examining Memory and Disassembling Code"
*   "Setting Machine-Level Breakpoints"
*   "Continuing Execution After a Machine-Level Breakpoint"
*   "Tracing Execution at the Machine Level"
*   "Writing Conditional Commands at the Machine Level"
*   "Stepping Through Machine Code"

## Examining and Changing Register Values

Using *dbx*, you can examine and change the hardware registers during execution of your program. Table 7-1 lists the machine form of the register names and the alternate software names as defined in the include file *regdef.h*.

**Table 7-1**          Hardware Registers and Aliases

| Register | Software Name | Description |
| --- | --- | --- |
| $r0 | $zero | Always 0 |
| $r1 | $at | Reserved for assembler |
| $r2... $r3 | $v0... $v1 | Expression evaluations, function return values, static links |
| $r4... $r7 | $a0... $a3 | Arguments |
| $r8... $r11 | $t0... $t7 | Temporaries (32 bit) |
|  | $a4... $a7, $ta0... $ta3 | Arguments (64 bit) |

**Table 7-1 (continued)**        Hardware Registers and Aliases

| Register | Software Name | Description |
| --- | --- | --- |
| $r12... $r15 | $t4... $t7, $t0... $t3 | Temporaries (32 bit) |
|  | $ta0... $ta3 | Temporaries (64 bit) |
| $r16... $r23 | $s0... $s7 | Saved across procedure calls |
| $r24... $r25 | $t8... $t9 | Temporaries |
| $r26... $r27 | $k0... $k1 | Reserved for kernel |
| $r28 | $gp | Global pointer |
| $r29 | $sp | Stack pointer |
| $r30 | $s8 | Saved across procedure calls |
| $r31 | $ra | Return address |
| $mmhi |  | Most significant multiply/divide result register |
| $mmlo |  | Least significant multiply/divide result register |
| $fcsr |  | Floating point control and status register |
| $feir |  | Floating point exception instruction register |
| $cause |  | Exception cause register |
| $d0, $d2, ... $d30 |  | Double precision floating point registers (32 bit) |
| $d0, $d2, ... $d31 |  | (64 bit) |
| $f0, $f2, ... $f30 |  | Single precision floating point registers (32 bit) |
| $f0, $f1, ... $f31 |  | (64 bit) |

For registers with alternate names, the *dbx* variable *$regstyle* controls which name is displayed when you disassemble code (as described in "Examining Memory and Disassembling Code" on page 109). If *$regstyle* is set to 0, then *dbx* uses the alternate form of the register name (for example, "zero" instead of "r0," and "t1" instead of "r9"); if *$regstyle* is anything other than 0, the machine names are used ("r0" through "r31").

## Printing Register Values

Use the *printregs* command to print the values stored in all registers.

The base in which the register values are displayed depends on the values of the *dbx* variables *$octints* and *$hexints*. By default, *dbx* prints the register values in decimal. You can set the output base to octal by setting the *dbx* variable *$octints* to a nonzero value. You can set the output base to hexadecimal by setting the *dbx* variable *$hexints* to a nonzero value. If you set both *$octints* and *$hexints* to nonzero values, *$hexints* takes precedence.

To examine the register values in hexadecimal, enter the following:

```
(dbx) set $hexints = 1
(dbx) printregs
r0/zero=0x0     r1/at=0x19050
r2/v0=0x8       r3/v1=0x100120e0
r4/a0=0x4       r5/a1=0xffffffad78
r6/a2=0xffffffad88      r7/a3=0x0
r8/a4=0x10      r9/a5=0x20
r10/a6=0x0      r11/a7=0xfbd5990
r12/t0=0x0      r13/t1=0x0
r14/t2=0x65     r15/t3=0x0
r16/s0=0x1      r17/s1=0xffffffad78
r18/s2=0xffffffad88     r19/s3=0xffffffaf70
r20/s4=0x0      r21/s5=0x0
r22/s6=0x0      r23/s7=0x0
r24/t8=0x0      r25/t9=0x10001034
r26/k0=0x0      r27/k1=0x20
r28/gp=0x1001a084       r29/sp=0xffffffaca0
r30/s8=0x0      r31/ra=0x1000110c
```

```
mdhi=0x0         mdlo=0xe0
cause=0x24       pc=0x10001050
fpcsr=0x0
f0=0.0000000e+00        f1=0.0000000e+00        f2=0.0000000e+00
f3=0.0000000e+00        f4=0.0000000e+00        f5=0.0000000e+00
f6=0.0000000e+00        f7=0.0000000e+00        f8=0.0000000e+00
f9=0.0000000e+00        f10=0.0000000e+00       f11=0.0000000e+00
f12=0.0000000e+00       f13=0.0000000e+00       f14=0.0000000e+00
f15=0.0000000e+00       f16=0.0000000e+00       f17=0.0000000e+00
f18=0.0000000e+00       f19=0.0000000e+00       f20=0.0000000e+00
f21=0.0000000e+00       f22=0.0000000e+00       f23=0.0000000e+00
f24=0.0000000e+00       f25=0.0000000e+00       f26=0.0000000e+00
f27=0.0000000e+00       f28=0.0000000e+00       f29=0.0000000e+00
f30=0.0000000e+00       f31=0.0000000e+00
d0=0.000000000000000e+00        d1=0.000000000000000e+00
d2=0.000000000000000e+00        d3=0.000000000000000e+00
d4=0.000000000000000e+00        d5=0.000000000000000e+00
d6=0.000000000000000e+00        d7=0.000000000000000e+00
d8=0.000000000000000e+00        d9=0.000000000000000e+00
d10=0.000000000000000e+00       d11=0.000000000000000e+00
d12=0.000000000000000e+00       d13=0.000000000000000e+00
d14=0.000000000000000e+00       d15=0.000000000000000e+00
d16=0.000000000000000e+00       d17=0.000000000000000e+00
d18=0.000000000000000e+00       d19=0.000000000000000e+00
d20=0.000000000000000e+00       d21=0.000000000000000e+00
d22=0.000000000000000e+00       d23=0.000000000000000e+00
d24=0.000000000000000e+00       d25=0.000000000000000e+00
d26=0.000000000000000e+00       d27=0.000000000000000e+00
d28=0.000000000000000e+00       d29=0.000000000000000e+00
d30=0.000000000000000e+00       d31=0.000000000000000e+00
```

(Note that there are twice as many floating point registers with 64-bit programs.) You can also use the value of a single register in an expression by typing the name of the register preceded by a dollar sign ($).

For example, to print the current value of the program counter (the *pc* register), enter:

```
(dbx) printx $pc
0x10001050
```

## Changing Register Values

In the same way you change the values of program variables, you can use the *assign* command to change the value of registers:

**assign** *register* = *expression*

> Assigns the value of *expression* to *register*. You must precede the name of the register with a dollar sign ($).

For example:

```
(dbx) assign $f0 = 3.14159
3.1415899999999999
(dbx) assign $t3 = 0x5a
0x5a
```

By default, the *assign register* command changes the register value in the current activation level, which is a typical operation. To force the hardware register to be updated regardless of the current activation level, use the *$ set $framereg* command.

## Examining Memory and Disassembling Code

The *listregions* command shows all memory regions, along with their sizes, in use by your program. This overview can be particularly useful if you want to know to what piece of your program a given data address corresponds. Since *listregions* shows the sizes of the memory regions, it allows you to easily determine the sizes of the data and stack regions of your program.

The forward slash (/) and question mark (?) commands allow you to examine the contents of memory. Depending on the format you specify, you can display the values as numbers, characters, or disassembled machine code. Note that all common forms of *address* are supported. Some unusual expressions may not be accepted unless enclosed in parentheses, as in (*address*)/*count format*.

The commands for examining memory have the following syntax:

*address* **/** *count format*
> Prints the contents of the specified address, or disassembles the code for the instruction at the specified address. Repeat for a total of *count* addresses in increasing address—in other words, an "examine forward" command. The format codes are listed in Table 7-2.

*address* **?** *count format*
> Prints the contents of the specified address or, disassembles the code for the instruction at the specified address. Repeat for a total of *count* addresses in decreasing address—in other words, an "examine backward" command. The format codes are listed in Table 7-2.

*address* **/** *count* **L** *value mask*
> Examines *count* 32-bit words in increasing addresses; prints those 32-bit words which, when ORed with *mask*, equals *value*. This command searches memory for specific patterns.

**./**
> Repeats the previous examine command with increasing address.

**.?**
> Repeats the previous examine command with decreasing address.

**Table 7-2**     Memory Display Format Codes

| Format Code | Displays Memory in the Format |
|---|---|
| i | print machine instructions (disassemble) |
| d | print a 16-bit word in signed decimal |
| D | print a 32-bit word in signed decimal |
| dd | print a 64-bit word in signed decimal |
| o | print a 16-bit word in octal |
| O | print a 32-bit word in octal |
| oo | print a 64-bit word in octal |
| x | print a 16-bit word in hexadecimal |
| X | print a 32-bit word in hexadecimal |
| xx | print a 64-bit word in hexadecimal |
| v | print a 16-bit word in unsigned decimal |

**Table 7-2 (continued)**     Memory Display Format Codes

| Format Code | Displays Memory in the Format |
|---|---|
| V | print a 32-bit word in unsigned decimal |
| vv | print a 64-bit word in unsigned decimal |
| L | like X but use with *val mask* |
| b | print a byte in octal |
| c | print a byte as character |
| s | print a string of characters that ends in a null byte |
| f | print a single-precision real number |
| g | print a double-precision real number |

For example, to display 10 disassembled machine instructions starting at the current address of the program counter, enter:

```
(dbx) $pc/10i
*[main:26, 0x400290]    sw      zero,28(sp)
 [main:27, 0x400294]    sw      zero,24(sp)
 [main:29, 0x400298]    lw      t1,28(sp)
 [main:29, 0x40029c]    lw      t2,32(sp)
 [main:29, 0x4002a0]    nop
 [main:29, 0x4002a4]    slt     at,t1,t2
 [main:29, 0x4002a8]    beq     at,zero,0x4002ec
 [main:29, 0x4002ac]    nop
 [main:31, 0x4002b0]    lw      t3,28(sp)
 [main:31, 0x4002b4]    nop
```

To disassemble another 10 lines, enter:

```
(dbx) ./
 [main:31, 0x4002b8]    addiu   t4,t3,1
 [main:31, 0x4002bc]    sw      t4,28(sp)
 [main:32, 0x4002c0]    lw      t5,24(sp)
 [main:32, 0x4002c4]    lw      t6,28(sp)
 [main:32, 0x4002c8]    nop
 [main:32, 0x4002cc]    addu    t7,t5,t6
 [main:32, 0x4002d0]    sw      t7,24(sp)
 [main:33, 0x4002d4]    lw      t8,28(sp)
 [main:33, 0x4002d8]    lw      t9,32(sp)
```

```
[main:33, 0x4002dc]    nop
```

To examine ten 32-bit words starting at address 0x7fffc754, and print those whose least significant byte is hexadecimal 0x19, enter:

```
(dbx) 0x7fffc754 / 10 L 0x19 0xff
7fffc758:  00000019
```

Consider a single-precision floating point array named *array*. You can examine the six consecutive elements, beginning with the fifth element, by entering:

```
(dbx) &array[4] / 6f
7fffc748:  0.2500000 0.2000000 0.1666667 0.1428571
7fffc758:  0.1250000 0.1111111
```

## Setting Machine-Level Breakpoints

*dbx* allows you to set breakpoints while debugging machine code just as you can while debugging source code. You set breakpoints at the machine code level using the *stopi* command.

The conditional and unconditional versions of the *stopi* commands work in the same way as the *stop* command described in "Setting Breakpoints" on page 79, with these exceptions:

- The *stopi* command checks its breakpoint conditions on a machine-instruction level instead of a source-code level.

- The *stopi at* command requires an address rather than a line number.

Each breakpoint is assigned a number when you create it. Use this number to reference the breakpoint in the various commands provided for manipulating breakpoints (for example, *disable*, *enable*, and *delete*, all described in "Managing Breakpoints, Traces, and Conditional Commands" on page 89).

## Syntax of the *stopi* Command

The syntax of the *stopi* command is:

**stopi at**          Sets an unconditional breakpoint at the current instruction.

**stopi at address**
                Sets an unconditional breakpoint at the specified address.

**stopi  in procedure**
                Sets an unconditional breakpoint to stop execution upon entering the specified procedure.

**stopi [expression|variable]**
                Inspects the value before executing each machine instruction and stops if the value has changed.

                If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stopi [expression|variable] at address**
                Inspects the value when the program is at the given address and stops if the value has changed (for machine-level debugging).

                If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stopi [expression|variable] in procedure**
                Inspects the value at every machine instruction within a given procedure and stops if the value has changed.

                If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stopi if expression**
                Evaluates the expression before executing each instruction and stops if the expression is true. Note that execution is very slow if you choose this type of conditional breakpoint.

**stopi at address if expression**
                Evaluates the expression at the given address and stops if the expression is true.

**113**

**stopi in** procedure **if** expression

> Evaluates the expression at every instruction within a given procedure and stops if the expression is true.

**stopi** [expression1 | variable] **if** expression2

> Tests both conditions before executing each machine instruction. Stops if both conditions are true.

> If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stopi** [expression1 | variable] **at** address **if** expression2

> Tests both conditions at the given address (for machine-level debugging). Stops if both conditions are true.

> If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stopi** [expression1 | variable] **in** procedure **if** expression2

> Tests the expression each time that the given variable changes within the given procedure.

> If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**Note:** When you stop execution because of a machine-level breakpoint set by one of the *stopi in* commands, a *where* command at the point of stop may yield an incorrect stack trace. This is because the stack for the function is not completely set up until several machine instructions have been executed. *dbx* attempts to account for this, but is sometimes unsuccessful.

## Linking With DSOs

If you link with a DSO, be careful when you use the *stopi at* command. For example, suppose you enter:

```
dbx() stopi at functionx
```

The breakpoint at *functionx* is hit only if the *gp_prolog* instruction is executed. (*gp_prolog* is a short sequence of instructions at the beginning of the routine.)

To avoid this problem, use the *stopi in* command:

dbx() **stopi in functionx**

If you really want to use *stopi at*, a safe alternative is to disassemble *functionx* and put the breakpoint after the *gp_prolog*. For more information on *gp_prolog*, see the *MIPSpro Assembly Language Programmer's Guide*.

The *tracei at*, *wheni at*, and *conti at* commands described in the following sections also follow this pattern. Use their "*in*" versions to ensure that the function breakpoint is hit.

## Continuing Execution After a Machine-Level Breakpoint

The *conti* command continues executing assembly code after a breakpoint. As with the *cont* command, if your program stops because *dbx* catches a signal intended for your program, then *dbx* sends that signal to your program when you continue execution. You can also explicitly send a signal to your program when you continue execution. Signal processing and sending signals to your program is discussed in "Using Signal Processing" on page 92.

The syntax of the *conti* command is:

**conti** [**signal**]   Continues execution with the current instruction.

**conti** [**signal**] {**at** | **to**} **address**
> Sets a temporary breakpoint at the specified address, then resumes execution with the current instruction. When your program reaches the breakpoint at **address**, *dbx* stops your program and deletes the temporary breakpoint.

**conti** [**signal**] **in procedure**
> Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current instruction. When your program reaches the breakpoint in **procedure**, *dbx* stops your program and deletes the temporary breakpoint.

See also "Linking With DSOs" on page 114 for a description on using the *conti in* and *conti at* commands with DSOs.

## Tracing Execution at the Machine Level

The *tracei* command allows you to observe the progress of your program while debugging machine code, just as you can with the *trace* command while debugging source code. The *tracei* command traces in units of machine instructions instead of in lines of code.

Each trace is assigned a number when you create it. Use this number to reference the breakpoint in the various commands provided for manipulating breakpoints (for example, *disable*, *enable*, and *delete*, all described in "Managing Breakpoints, Traces, and Conditional Commands" on page 89).

The syntax of the *tracei* command is:

**tracei** [**expression** | **variable**]

> Whenever the specified variable changes, *dbx* prints the old and new values of that variable. (For machine-level debugging.) Note that execution is very slow if you choose this type of trace.
>
> If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**tracei procedure**

> This command is equivalent to entering **trace procedure**. *dbx* prints the values of the parameters passed to the specified procedure whenever your program calls it. Upon return, *dbx* prints the return value.

**tracei** [**expression** | **variable**] **at address**

> Prints the value of the variable whenever your program reaches the specified address. (For machine-level debugging.)
>
> If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**tracei** [**expression** | **variable**] **in procedure**

> Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable. (For machine-level debugging.)

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**tracei** [**expression1 | variable**] **at address if expression2**

Prints the value of the variable whenever your program reaches the specified address and the given expression is true. (For machine-level debugging.)

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**tracei** [**expression1 | variable**] **in procedure if expression2**

Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable, if the given expression is true. (For machine-level debugging.)

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

See also "Linking With DSOs" on page 114 for a description on using the *tracei in* and *tracei at* commands with DSOs.

## Writing Conditional Commands at the Machine Level

Use the *wheni* command to write conditional commands for use in debugging machine code. The *wheni* command works in the same way as the *when* command described in "Writing Conditional Commands" on page 87. The command list is a list of *dbx* commands, separated by semicolons. When the specified conditions are met, the command list is executed. If one of the commands in the list is *stop* (with no operands), then the process stops when the command list is executed.

**wheni if** *expression* {*command-list*}
>  Evaluates the expression before executing each machine instruction. If the expression is true, executes the command list.

**wheni at** *address* **if** *expression* {*command-list*}
>  Evaluates the expression at the given address. If the expression is true, executes the command list.

**wheni in** *procedure* **if** *expression* {*command-list*}
>  Evaluates the expression in the given procedure. If the expression is true, executes the command list.

**wheni** *variable* **at** *address* **if** *expression* {*command-list*}
>  Tests both conditions at the given address. If the conditions are true, executes the command list. (For machine-level debugging.)
>
>  If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**wheni** *variable* **in** *procedure* **if** *expression* {*command-list*}
>  Tests both conditions at every machine instruction within a given procedure. If they are true, executes the command list.

See also "Linking With DSOs" on page 114 for a description on using the *wheni in* and *wheni at* commands with DSOs.

## Stepping Through Machine Code

The *stepi* and *nexti* commands allow you to step through machine code in much the same way you can with the *step* and *next* commands while debugging source code. The *stepi* and *nexti* commands step in units of machine instructions instead of in lines of code.

The formats of the *nexti* and *stepi* commands are:

**nexti** [**integer**] Executes the specified number of machine instructions, stepping over procedures. If you do not provide an argument, *nexti* executes one instruction. If *nexti* encounters any breakpoints, even in procedures that it steps over, it immediately stops execution.

**stepi** Single steps one machine instruction, stepping into procedures (as called by *jal* and *jalr*). If *stepi* encounters any breakpoints, it immediately stops execution.

**stepi** [**n**] Executes the specified number of machine instructions, stepping into procedures (as called by **jal** and **jalr**).

The value of the *dbx* variable *$stepintoall* affects the *stepi* and *nexti* commands just as it does the *step* and *next* commands. See the section "Stepping Through Your Program" in Chapter 6 for a discussion.

If your program has DSOs, set the environment variable *LD_BIND_NOW* to 1 before you run your program. This forces complete run-time linking when your program starts. Otherwise, you could accidentally step into the run-time linker, **rld(1)**, which becomes part of your program at run time.

# Multiple Process Debugging

This chapter explains multiprocess debugging procedures, including:

- "Listing Available Processes"
- "Adding a Process to the Process Pool"
- "Deleting a Process From the Process Pool"
- "Selecting a Process"
- "Suspending a Process"
- "Resuming a Suspended Process"
- "Waiting for a Resumed Process"
- "Waiting for Any Running Process"
- "Killing a Process"
- "Handling fork System Calls"
- "Handling exec System Calls"
- "Handling sproc System Calls and Process Group Debugging"

## Processes and Threads

*dbx* supports debugging multiprocess applications, including processes spawned with either the **fork**(2) or **sproc**(2) system calls. You can attach child processes automatically to *dbx*. You also can perform process control operations on a single process or on all processes in a group.

*dbx* provides commands specifically for seizing, stopping, and debugging currently running processes. When *dbx* seizes a process, it adds it to a pool of processes available for debugging. Once you select a process from the pool of available processes, you can use all the *dbx* commands normally available.

Once you are finished with the process, you can terminate it, return it to the pool, or return it to the operating system.

dbx also provides limited support for the IRIX pthreads library. You can obtain information about threads, but cannot specify threads in program-control commands.

## Using the **pid** Clause

Many *dbx* commands allow you to append the clause **pid** *pid* (where *pid* is a numeric process ID or a debugger variable holding a process ID). Using the **pid** *pid* clause means you can apply a command to any process in the process pool even though it is not the active process.

For example, to set a breakpoint at line 97 of the process whose ID is 12745, enter:

```
(dbx) stop at 97 pid 12745
Process 12745: [3] stop at "/usr/demo/test.c":97
```

Commands that accept the **pid** *pid* clause include:

```
active         edit        resume         wait
addproc        file        return         whatis
assign         func        showpoc        when, when[i]
catch          goto        status         where
cont, cont[i]  ignore      step, step[i}  whereis
delete         kill        stop, stop[i]  which
delproc        next        suspend
directory      print       trace, trace[i]
down           printf      up
dump           printregs   use
```

## Using the `pgrp` Clause

Many *dbx* commands allow the *pgrp* clause as a way to apply a command to several processes. For information, see "Handling sproc System Calls and Process Group Debugging" on page 132.

## Using the `thread` Clause

You can append the clause **thread** *tid* (where *tid* is a numeric thread ID, a debugger variable holding a thread ID, or the qualifier **all**) to some *dbx* commands that provide program information. You cannot use the **thread** *tid* clause with program-control commands such as *stop*, *trace*, *when* or *continue*. Using the **thread** *tid* clause means you can apply a command to any thread even if it is not current or in the current process. The current thread is defined to be the thread that is running in the current process. Examples of the **thread** *tid* clause are:

(dbx) **where thread**

(dbx) **where thread** *$no*

(dbx) **print x thread all**

The outputs of these commands are respectively: a stack trace of the current thread, a stack trace of the thread whose ID is stored in *$no*, and the values of all instances of the program variable *x* in all threads.

The *showthread* command provides status information about the threads in your program. In one *dbx* session, you cannot debug more than one program that uses threads.

The syntax of the *showthread* command is:

**showthread** [**full**]
> Prints brief status information about the current thread. If the **full** qualifier is included, prints full status information.

**showthread** [**full**] [**thread**] {*number* | *$no* | **all**}
> Prints brief status information about the thread identified by *number* or the value of *$no*, or all threads associated with the debug session. If the **full** qualifier is included, prints full status information. The **thread** qualifier does not affect the output, but it is allowed so the syntax can be the same as that for other commands that use the **thread** clause.

### Using Scripts

Additionally, *dbx* provides two variables that you can use when writing scripts to debug multiprocess programs:

*$lastchild*    Always set to the process ID of the last child process created by a **fork** or **sproc**.

*$pid0*       Always set to the process ID of the process started by the *run* command.

See the *dbx* online help file section on *hint_mp_debug* for sample multiprocessing debugging scripts.

## Listing Available Processes

Use the *showproc* command to list the available processes:

**showproc**     Shows processes already in the *dbx* process pool or processes that *dbx* can control. Without any arguments, *dbx* lists the processes it already controls.

**showproc all** Lists all the processes it controls as well as all those processes it could control but that are not yet added to the process pool.

**showproc** *pid*  Shows the status of the process ID.

For example, to display all processes in the process pool, enter:

```
(dbx) showproc
Process 12711 (test) Trace/BPT trap [main:14 ,0x40028c]
Process 12712 (test) Trace/BPT trap [main:18 ,0x4002b4]
```

To display only process 12712, enter:

```
(dbx) showproc 12712
Process 12712 (test) Trace/BPT trap [main:18 ,0x4002b4]
```

To display all processes that *dbx* can control, enter:

```
(dbx) showproc all
Process 12711 (test) Trace/BPT trap [main:14 ,0x40028c]
Process 12055 (tcsh)
Process 12006 (clock)
Process 12054 (tcsh)
Process 12673 (zipxgizmo)
Process 12672 (zip)
Process 11974 (4Dwm)
Process 12712 (test) Trace/BPT trap [main:18 ,0x4002b4]
Process 12708 (dbx)
Process 12034 (xlock)
```

## Adding a Process to the Process Pool

The *addproc* command adds one or more specified processes to the *dbx* process pool. This allows you to debug a program that is already running. The syntax of the *addproc* command is:

**addproc** *pid* [ ... ]
**addproc** *var*

For example:

```
(dbx) addproc 12924
Reading symbolic information of Process 12924 . . .
Process 12924 (loop_test) added to pool
Process 12924 (loop_test) running
```

Equivalently, you can enter either of the following:

```
(dbx) set $foo = 12924
(dbx) addproc $foo
```

**125**

## Deleting a Process From the Process Pool

The *delproc* command removes a process or variable from the process pool, freeing it from *dbx* control. When you delete a process from the process pool, *dbx* automatically returns the process to normal operation. The syntax of the *delproc* command is:

**delproc** *pid* [ ... ]
**delproc** *var*

For example:

```
(dbx) delproc 12924
Process 12924 (loop_test) deleted from pool
```

Equivalently, you can enter either of the following:

```
(dbx) set $foo = 12924
(dbx) delproc $foo
```

## Selecting a Process

The *dbx* command has the ability to control multiple processes. However, *dbx* commands (by default) apply to only one process at a time, the *active process*. To select a process from the process pool to be the active process, use the *active* command:

**active** [*pid*]     Selects a process, *pid*, from *dbx* process pool as the active process. If you do not provide a process ID, *dbx* prints the currently active process ID.

For example, to determine which process is currently active, enter:

```
(dbx) active
Process 12976 (test1) is active
```

To then select process 12977 as the active process, enter:

```
(dbx) active 12977
Process 12977 (test1) after fork [.fork.fork:15 +0x8,0x4005e8]
```

## Suspending a Process

The *suspend* command allows you to stop a process in the *dbx* process pool:

**suspend**      Suspends the active process if it is running. If it is not running, this command does nothing.

**suspend all**

      Suspends all the processes.

**suspend pid** *pid*

      Suspends the process *pid* if it is in the *dbx* process pool. If it is not running, this command does nothing.

**suspend** *pgrp*

      Suspends all the processes in the *pgrp*.

For example, to stop the active process, enter:

```
(dbx) suspend
Process 12987 (loop_test) requested stop [main:10 +0x8,0x400244]
  10  i = i % 10;
```

Then to stop process 12988, enter:

```
(dbx) suspend pid 12988
Process 12988 (test3) requested stop [main:29 +0x4,0x400424]
  10  j = k / 10.0;
```

## Resuming a Suspended Process

To resume execution of a suspended *dbx* controlled process, you can use either the *cont* command or the *resume* command. If you use *cont*, you do not return to the *dbx* command interpreter until the program encounters an event (for example, a breakpoint). On the other hand, the *resume* command returns immediately to the *dbx* command interpreter.

The syntax of the *resume* command is:

**resume**      Resumes execution of the program, and returns immediately to the *dbx* command interpreter.

**resume** [*signal*]  Resumes execution of the process, sending it the specified signal, and returns immediately to the *dbx* command interpreter.

Because the *resume* command returns you to the *dbx* command interpreter after restarting the process, it is more useful than using the *cont* command when you're debugging

multiple processes. With *resume*, you are free to select and debug a process while another process is running.

If any *resume*d process modifies the terminal modes (for example if it uses *curses*(3X)), *dbx* can't correctly control the modes. Intercept programs using *curses* by typing **dbx –p** (or **dbx –P**).

For example, if you are debugging multiple processes and want to resume the active process, enter:

```
(dbx) resume
```

*dbx* restarts the active process and returns the *dbx* prompt. You can then continue debugging, for example by switching to another process.

To resume all the processes in **pgrp 2** and send a **SIGINT** signal to the process when *dbx* resumes, enter:

```
(dbx) resume SIGINT 2
```

## Waiting for a Resumed Process

To wait for a process to stop for an event (such as a breakpoint), use the *wait* command. This is useful after a *resume* command. Also refer to the description of the *waitall* command, described in "Waiting for Any Running Process" on page 129.

The syntax of the *wait* command is:

**wait**          Waits for the active process to stop for an event.

**wait pid** *pid*   Waits for the process *pid* to stop for an event.

For example, assume that you want to wait until process 14280 stops, perhaps at a breakpoint you have set. To do so, enter:

```
(dbx) wait pid 14280
```

After you enter this command, *dbx* waits until process 14280 stops, at which point it displays the *dbx* prompt.

## Waiting for Any Running Process

To wait for any process currently running to breakpoint or stop for any reason, use the *waitall* command. It causes *dbx* to wait until a running process in the process list stops, at which point it returns you to the *dbx* command interpreter.

**Note:** When you return to the *dbx* command interpreter after a *waitall* command, *dbx* does *not* make the process that stopped the active process. You must use the *active* command to change the active process.

For example, to wait until one of your processes under *dbx* control stops, enter:

```
(dbx) waitall
```

After you enter this command, *dbx* waits until a process stops, at which point it indicates which process stopped and displays the *dbx* prompt. For example:

```
Process 14281 (loop_test) Terminated [main:10 +0x8,0x400244]
  10  i = i % 10;
(dbx)
```

## Killing a Process

To kill a process in the process pool while running *dbx*, use the *kill* command:

**kill**          Kills the active process.

**kill** *pid* [ ... ]    Kills the specified process(es).

For example, to kill process 14257, enter:

```
(dbx) kill 14257
Process 14257 (fork_test) terminated
Process 14257 (fork_test) deleted from pool
```

## Handling *fork* System Calls

When a program executes a **fork** system call and starts another process, *dbx* allows you to add that process to the process pool. (See also "Stopping at System Calls" on page 96.)

The *dbx* variable *$promptonfork* determines how *dbx* treats forks. Table 8-1 summarizes its effects.

**Table 8-1**        How the *$promptonfork* Variable Affects *dbx*'s Treatment of Forks

| $promptonfork Value | Effect on *dbx*'s Treatment of Forks |
|---|---|
| 0 (default) | *dbx* does not add the child process to the process pool. Both the child process and the parent process continue to run. |
| 1 | *dbx* stops the parent process and asks if you want to add the child process to the process pool. If you answer yes, then *dbx* adds the child process to the pool and stops the child process; if you answer no, *dbx* allows the child process to run and does not place it in the process pool. |
| 2 | *dbx* automatically stops both the parent and child processes and adds the child process to the process pool. |

**Note:** "Handling sproc System Calls and Process Group Debugging" on page 132 provides additional information on debugging multiprocessing programs; some of the material in that section can apply also to programs that use the **fork** system call.

Consider a program named *fork* that contains these lines:

```
main(argc, argv)
int argc;
char *argv;
```

```
{
   int pid;
   if ((pid = fork()) == -1)
      perror("fork");
   else if (pid == 0)
      printf("child\n");
   else { printf("parent\n");
}
```

If you set *$promptonfork* to 1 and run the program, *dbx* prompts you whether it should add the child process to the process pool:

```
(dbx) set $promptonfork = 1
(dbx) run
Process 22661 (fork) started
Process 22662 (fork) has executed the "fork" system call

Add child to process pool (n if no)?y
Process 22662 (fork) added to pool
Process 22662 (fork) stopped on sysexit fork [_fork:28 ,0x40643a4]
Process 22661 (fork) stopped on sysexit fork [_fork:28 ,0x40643a4]
        Source (of /shamu/lib/libc/libc_64/proc/fork.s) not available
for Process 22661
```

## Handling *exec* System Calls

The **exec** system call executes another program. During an **exec**, the first program gives up its process number to the program it executes. When a program using DSOs executes an *exec()* call, *dbx* runs the new program to **main**. When a program linked with a non-shared library executes an *exec()* call, *dbx* reads the symbolic information for the new program and then stops program execution. In either case, you can continue by entering a *cont* or *resume* command.

For example, consider the programs *exec1.c* and *exec2.c*:

```
/* exec1.c */
main()
{
   printf("in exec1\n");

/* Invoke the "exec2" program */

   execl("exec2", "exec2", 0);

   /* We'll only get here if execl() fails */

   perror("execl");
}
/* exec2.c */
main()
{
   printf("in exec2\n");
}
```

You can enter *cont* to continue executing *exec2*. For example:

```
(dbx) cont
in exec2
Process 14409 (exec2) finished
```

## Handling *sproc* System Calls and Process Group Debugging

The process group facility allows a group of processes to be operated on simultaneously by a single *dbx* command. This is more convenient to use when dealing with processes created with the **sproc** system call than issuing individual *resume*, *suspend*, or breakpoint setting commands. This facility was created to deal more conveniently with parallel programs created, for example, by the Power Fortran Accelerator (PFA).

The *dbx* variable *$mp_program* determines how *dbx* treats **sproc** system calls. Table 8-2 summarizes its effects.

**Table 8-2**      How the *$mp_program* Variable Affects *dbx*'s Treatment of *sproc*s

| $mp_program Value | Effect on *dbx*'s Treatment of sproc |
|---|---|
| 0 (default) | *dbx* treats calls to **sproc** in the same way as it treats calls to **fork**. |
| 1 | Child processes created by calls to **sproc** are allowed to run; they block on multiprocessor synchronization code emitted by mp Fortran or C code. When you set *$mp_program* to 1, multiprocess Fortran or C code is easier to debug. |

Whenever a process executes a **sproc**, if *dbx* adds the child to the process pool, *dbx* also adds the parent and child to the *group list*. The group list is simply a list of processes. If you set the *dbx* variable *$groupforktoo* to 1, then forked processes are added to the group list automatically just as **sproc**ed processes are. (By default, *$groupforktoo* is 0.)

You can explicitly add one or more processes to the group list with the *addpgrp* command (you can add only processes in the process pool to the group list):

**addpgrp** *pid* [ ... ]

You can remove processes from the group list with the *delpgrp* command:

**delpgrp** *pid* [ ... ]

The *showpgrp* command displays information about the group list. The *showpgrp* command shows the process group numbers and all the *stop*, *trace*, or *when* events in each. These events are created by *stop[i]*, *when[i]* ... **pgrp** (which create multiple *stop*, *trace*, or *when* events) and by *delete* **pgrp** commands, which delete them.

The following example shows the output of the *showpgrp* command with two processes in the group list:

```
(dbx) showpgrp
2 processes in group:
 14559 14558
```

Once you add processes to the group list (by adding the keyword **pgrp** to the end of certain *dbx* commands), you can apply that command to all processes in the group. The commands to which you can append *pgrp* are: *delete, list, next[i], resume, status, stop[i], suspend, trace[i]*, and *when*.

The breakpoints and traces set by the *stop[i], trace[i]*, and *when* commands, when used with the **pgrp** keyword, are also added to the *group history*. This group history is displayed as a numbered list when you execute *showpgrp*.

To delete breakpoints from multiple processes with a single command, use the group history number with the *delete* command. For example, to delete the history entry 7 for the process group, enter:

```
(dbx) delete 7 pgrp
```

The *dbx* variable *$newpgrpevent* stores the group history number of the most recent **pgrp** event. This can be useful when writing a script, for example:

```
set $myevent = $newpgrpevent
....
delete $myevent pgrp
```

Breakpoints set on the process group are recorded both in the group and in each process. Deleting breakpoints individually (even if set by a group command) is allowed.

For example, the following command sets a breakpoint at line 10 in all processes in the group list:

```
(dbx) stop at 10 pgrp
Process 14558: [6] stop at "/usr/demo/pgrp_test.c":10
Process 14559: [7] stop at "/usr/demo/pgrp_test.c":10
```

If you now enter a *status* command, only those breakpoints associated with the active process are displayed:

```
(dbx) status
Process 14559: [7] {pgrp 269011340} stop at "/usr/demo/pgrp_test.c":10
```

By appending the keyword **pgrp**, you can display the breakpoints for all processes in the group list:

```
(dbx) status pgrp
Process 14558: [6] {pgrp 269011276} stop at "/usr/demo/pgrp_test.c":10
Process 14559: [7] {pgrp 269011340} stop at "/usr/demo/pgrp_test.c":10
```

Use the *showpgrp* command to display the group history:

```
(dbx) showpgrp
2 processes in group:
 14559 14558
Group history number: 10
        Process 14558 Process 14558: [6] stop at "/usr/demo/pgrp_test.c":10
        Process 14559 Process 14559: [7] stop at "/usr/demo/pgrp_test.c":10
```

You can delete the breakpoints in both processes by deleting the associated group history entry. For example, enter:

```
(dbx) delete 10 pgrp
(dbx) showpgrp
2 processes in group:
 14559 14558
```

**135**

# *dbx* Commands

All *dbx* commands are listed below along with a brief description. For more information about a command, refer to its description in the main text of this guide.

**;**                 Use the semicolon (*;*) as a separator to include multiple commands on the same command line.

\                 Use the backslash (\) at the end of a line of input to *dbx* to indicate that the command is continued on the next line.

**./**                 Repeats the previous examine command by incrementing the address.

**/**[*reg_exp*]                 Searches forward through the current source file from the current line for the regular expression *reg_exp*. If *dbx* reaches the end of the file without finding the regular expression, it wraps around to the beginning of the file. *dbx* prints the first source line containing a match of the search expression.

                 If you do not supply **reg_exp**, *dbx* searches forward, based on the last regular expression you searched for.

**.?**                 Repeats the previous examine command by decrementing the address.

**?**[*reg_exp*]                 Searches backward through the current source file from the current line for the regular expression *reg_exp*. If *dbx* reaches the beginning of the file without finding the regular expression, it wraps around to the end of the file. *dbx* prints the first source line containing a match of the search expression.

                 If you do not supply a regular expression, *dbx* searches backward, based on the last regular expression you searched for.

**!!**                 Repeats the previous command. If the value of the *dbx* variable *$repeatmode* is set to 1, then entering a carriage return at an empty line is equivalent to executing **!!**. By default, *$repeatmode* is set to 0.

**!***string*                 Repeats the most recent command that starts with the specified *string*.

**!***integer*                 Repeats the command associated with the specified *integer* in the history list.

**!-***integer*        Repeats the command that occurred *integer* times before the most recent command. Entering **!-1** executes the previous command, **!-2** the command before that, and so forth.

**active** [*pid*]        Selects a process, *pid*, from *dbx* process pool as the active process. If you do not provide a process ID, *dbx* prints the currently active process ID.

**addpgrp** *pid* [ ... ]
                Adds the process IDs specified to the group list. Only processes in the process pool can be added to the group list.

**addproc** *pid* [ ... ]
                Adds the specified process(es) to the pool of *dbx* controlled processes.

*address* **/** *count format*
                Prints the contents of the specified address or disassembles the code for the machine instruction at the specified address. Repeats for a total of *count* addresses in increasing address—in other words, an *examine forward* command. The format codes are listed in Table 7-2.

*address* **?** *count format*
                Prints the contents of the specified address or disassembles the code for the machine instruction at the specified address. Repeats for a total of *count* addresses in decreasing address—in other words, an "examine backward" command. The format codes are listed in Table 7-2.

*address* **/** *count* **L** *value mask*
                Examines *count* 32-bit words in increasing address and print those 32-bit words which, when ORed with *mask*, equal *value*. This command searches memory for specific patterns.

**alias**          Lists all existing aliases.

**alias** *name*      Lists the alias definition for *name*.

**alias** *name command*
                Defines *name* as an alias for *command*.

**alias** *name* "*string*"
                Defines *name* as an alias for *string*. With this form of the *alias* command, you can provide command arguments in the alias definition.

**alias** *name***(**arg1 **[, ...** argN**])** *"string"*
> Defines *name* as an alias for *string*. *arg1* through *argN* are arguments to the alias, appearing in the *string* definition. When you use the alias, you must provide values for the arguments, which *dbx* then substitutes in *string*.

**assign** *register* = *expression*
> Assigns the value of *expression* to *register*. You must precede the name of the register with a dollar sign ($).

**assign** *variable* = *expression*
> Assigns the value of *expression* to the program variable, *variable*.

**catch**          Prints a list of all signals caught.

**catch** {**signal** | **all**}
> Instructs *dbx* to stop your program whenever it receives the specified signal. If you use the keyword **all** rather than giving a specific signal, *dbx* catches all signals.

**ccall** *func***(**arg1, arg2, ... , argn**)**
> Calls a function with the given arguments.

**clearcalls**     Clears all stopped interactive calls.

**cont**           Continues execution with the current line.

**cont** {**at** | **to**} **line**
> Sets a temporary breakpoint at the specified source line, then resumes execution with the current line. When your program reaches the breakpoint at **line**, *dbx* stops your program and deletes the temporary breakpoint. The keywords **at** and **to** are equivalent.

**cont in procedure**
> Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current line. When your program reaches the breakpoint in **procedure**, *dbx* stops your program and deletes the temporary breakpoint.

**cont [signal]**  Continues execution with the current line and sends the specified signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.

**`cont` [signal] {`at` | `to`} line**
> Sets a temporary breakpoint at the specified source line, then resumes execution with the current line and sends the specified signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.

**`cont` [signal] `in` procedure**
> Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current line and sends the specified signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.

**`conti` [signal]**  Continues execution with the current machine instruction. If you specify a signal, *dbx* sends the signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.

**`conti` [signal] {`at` | `to`} address**
> Sets a temporary breakpoint at the specified address, then resumes execution with the current machine instruction. When your program reaches the breakpoint at **address**, *dbx* stops your program and deletes the temporary breakpoint.
>
> If you specify a signal, then *dbx* sends the signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.

**`conti` [signal] `in` procedure**
> Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current machine instruction. When your program reaches the breakpoint in **procedure**, *dbx* stops your program and deletes the temporary breakpoint.
>
> If you specify a signal, then *dbx* sends the signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.

**140**

**corefile** [*file*]     If you provide a filename, *dbx* uses the program data stored in the core dump *file*.

If you do not provide a filename, *dbx* displays the name of the current core file.

**delete** {**item** [, **item** ...] | **all**}
Deletes the item(s) specified. If you use the keyword **all** instead of listing individual items, *dbx* deletes all breakpoints, traces, and conditional commands.

**delpgrp** *pid* [ ... ]
Deletes the process IDs specified from the group list.

**delproc** *pid* [ ... ]
Deletes the specified process(es) from the pool of *dbx* controlled processes.

**dir** [**dir** ...]     If you provide one or more directories, *dbx* adds those directories to the end of the source directory list.

If you do not provide any directories, *dbx* displays the current source directory list.

**disable item** [, **item** ... ]
Disables the item(s) listed. The specified breakpoint(s), trace(s), or conditional command(s) no longer affect program execution. This command has no effect if the item you specify is already disabled.

**down** [*num*]     Moves down the specified number of activation levels in the stack. The default is one level.

**duel**     Invokes *duel*, the high-level debugging tool.

**duel alias**     Shows are current *duel* aliases.

**duel clear**     Deletes all *duel* aliases.

**dump**     Prints information about the variables in the current procedure.

**dump** *procedure*     Prints information about the variables in the specified procedure. The procedure must be active.

**dump .**     Prints information about the variables in all procedures currently active.

**edit** [*file | procedure*]

Edits a file. If you set the *dbx* variable *$editor* to the name of an editor, the *edit* command invokes that editor on the source file. If you do not set the *dbx* variable *$editor*, *dbx* checks whether you have set the environment variable *EDITOR* and, if so, invokes that editor. If you do not set either the *dbx* variable or the environment variable, *dbx* invokes the *vi* editor. When you exit the editor, you return to the *dbx* prompt.

If you supply a filename, *edit* invokes the editor on that file. If you supply the name of a procedure, *edit* invokes the editor on the file that contains the source for that procedure. If you do not supply a filename or a procedure name, *edit* invokes the editor on the current source file.

**edit pid** *pid*

Edits the process ID *pid* clause.

**enable item** [, **item** ... ]

Enables the item(s) specified. This command activates the specified breakpoint(s), trace(s), or conditional command(s), reversing the effects of a *disable* command, so that they affect program execution.

**file** [*file*]

Changes the current source file to *file*. The new file becomes the current source file, on which you can search, list, and perform other operations.

**func**

Displays the name of the procedure corresponding to the current activation level.

**func** {*activation_level | procedure*}

Changes the current activation level. If you specify an activation level by number, *dbx* changes to that activation level. If you specify **procedure**, *dbx* changes to the activation level of that procedure. If you specify a procedure name and that procedure has called itself recursively, *dbx* changes to the most recently called instance of that procedure. If you specify **procedure**, *dbx* changes the current source file to be that procedure, even if the procedure is not active.

**givenfile** [*file*]

If you provide a filename, *dbx* kills the currently running processes and loads the executable code and debugging information found in *file*.

If you do not provide a filename, *dbx* displays the name of the program that it is currently debugging.

**hed**

Edits only the last line of the history list (the last command executed).

**hed** *num1*

Edits line *num1* in the history list.

**hed** *num1,num2*
Edits the lines in the history list from *num1* to *num2*.

**hed all**  Edits the entire history list.

**help**  Shows the list of available help sections.

**help all**  Displays the entire *dbx* help file. *dbx* displays the file using the command name given by the *dbx $pager* variable. The *dbx* help file is large and can be difficult to use if you use a simple paging program like *more*(1). A useful technique is to set the *$pager* variable to a text editor like *vi*(1).

**help help**  Explains how to display the help file in your favorite editor.

**help** *section*  Shows this help section. *dbx* displays the file using the command name given by the *dbx $pager* variable. (By default, it uses *more*.) A useful technique is to set the *$pager* variable to a text editor like *vi*(1).

**history**  Prints the commands in the history list.

**ignore**  Prints a list of all signals ignored.

**intercept** {**all** | *item*}
Stops on all C++ exceptions, or exceptions that throw the base type *item*.

**intercept unexpected** {[**all**] | [*item* [, *item*] ]}
Stops on all C++ exceptions that have either no handler or are caught by an "unexpected" handler. You may omit **all**. If you specify *item*, stops on exceptions that throw the base type *item*.

**intercept ... if expression**
You can append the **if** clause to all *intercept* commands. Your program stops only if **expression** is non-zero. Note that the context for evaluation of **expression** is the C++ runtime library, not that of the throw, so use global variables or fully qualified names in **expression**.

**ignore** {**signal** | **all**}
Instructs *dbx* to ignore the specified signal. All ignored signals are passed to your program normally. If you use the keyword **all** rather than giving a specific signal, *dbx* ignores all signals.

**kill**  Kills the active process.

**kill** *pid* ...  Kills the active process(es) whose PIDs are specified.

**list** *exp*  Lists *$listwindow* lines starting with the line number given by the expression *exp*. The expression may be any valid expression that evaluates to an integer value.

**list** *exp1***:***exp2*  Lists *exp2* lines, beginning at line *exp1*.

**list** *exp1***,***exp2*  Lists all source between line *exp1* and line *exp2* inclusive.

**list** *func*  Lists *$listwindow* lines starting at procedure *func*.

**list** *func***:***exp*  Lists *exp2* lines, beginning at *func*.

**list** *func***,***exp*  Lists all source between *func* and *exp*, inclusive.

**listclones**  Lists all the root functions and their derived clones.

**listclones** *func*
> Lists the root and all derived clones for *func*.

**listinlines**  Lists all of the inlined routines with their start and end addresses.

**listinlines** *func*
> Lists all of the inlined versions of *func* with their start and end addresses.

**listobj**  Lists dynamic shared objects being used. The base application is first in the list.

**listregions**  Lists all the memory regions being used by the application. Any object region with debugging information is marked with a "Y."

**next** [**n**]  Executes the specified number of lines of source code, stepping over procedures. If you do not provide an argument, *next* executes one line. If *next* encounters any breakpoints, even in procedures that it steps over, it immediately stops execution.

**nexti** [**n**]  Executes the specified number of machine instructions, stepping over procedures. If you do not provide an argument, *nexti* executes one line. If *nexti* encounters any breakpoints, even in procedures which it steps over, it immediately stops execution.

**pixie clear**  Clears the basic block counts for the current execution.

**pixie write**  Writes the counts file with the current basic block counts. The counts reflect the exectuion of the program since the *run* command or since the last *pixie clear* command, whichever is more recent.

**playback input** [*file*]
> Executes the commands from *file*. The default file is the current temporary file created for the *record input* command. If the *dbx* variable *$pimode* is nonzero, the commands are printed out as they are played back.

**playback output** [*file*]
> Prints the commands from *file*. The default file is the current temporary file created for the *record output* command.

**print** [*exp1* [**,** *exp2***,** ... ] ]
> Prints the value(s) of the specified expression(s).

**printd** [*exp1* [**,** *exp2***,** ... ] ]
> Prints the value(s) of the specified expression(s) in decimal.

**printenv**
> Prints the list of environment variables affecting the program being debugged.

**printf** *string* [**,** *exp1* [**,** *exp2***,** ... ] ]
> Prints the value(s) of the specified expression(s) in the format specified by the string, *string*. The *printf* command supports all formats except "%s". For a list of formats, see the **printf**(3S) reference page.

**printo** [*exp1* [**,** *exp2***,** ... ] ]
> Prints the value(s) of the specified expression(s) in octal.

**printregs**
> Prints all register values.

**printx** [*exp1* [**,** *exp2***,** ... ] ]
> Prints the value(s) of the specified expression(s) in hexadecimal.

**quit**
> Quits *dbx*.

**record**
> Displays the current input and output recording sessions.

**record input** [*file*]
> Records everything you type to *dbx* in *file*. The default file is a temporary *dbx* file in the */tmp* directory. The name of the temporary file is stored in the *dbx* variable *$defaultin*.

**record output** [*file*]
> Records all *dbx* output in *file*. The default file is a temporary *dbx* file in the */tmp* directory. The name of the temporary file is stored in the *dbx* variable *$defaultout*. If the *dbx* variable *$rimode* is nonzero, *dbx* also records the commands you enter.

**rerun run-arguments**
> Without any arguments, repeats the last *run* command, if applicable. Otherwise, *rerun* is equivalent to the *run* command without any arguments.

**resume**    Resumes execution of the program, and returns immediately to the *dbx* command interpreter.

**resume** [*signal*]  Resumes execution of the process, sending it the specified signal, and returns immediately to the *dbx* command interpreter.

**return**    Continues execution until control returns to the next procedure on the stack.

**return** *proc*    Continues execution until control returns to the named procedure.

**run** *run-arguments*

Starts your program and passes to it any arguments that you provide. All shell processing is accepted, such as unglobbing of * and ? in filenames. Redirection of the program's standard input and standard output, and/or standard error is also done by the shell. In other words, the *run* command does exactly what typing **target run-arguments** does. You can specify a target, either on *dbx* invocation or in a prior *givenfile* command. *dbx* passes *./target* as **argv[0]** to *target* when you specify it as a relative pathname. You can specify *target* either on *dbx* invocation or in a prior *givenfile* command. *dbx* passes *./target* as **argv[0]** to *target* when you specify it as a relative pathname.

A *run* command must appear on a line by itself and cannot be followed by another *dbx* command. Terminate the command line with a return (new-line). Note that you cannot include a *run* command in the command list of a *when* command.

**set**    Displays a list of predefined and user defined variables.

**set** *var* **=** *exp*    Defines (or redefines) the specified *dbx* variable, setting its value to that of the expression you provide.

| | |
|---|---|
| **setenv** | Prints the list of environment variables for the program being debugged. |
| **setenv** *VAR* | Sets the environment variable *VAR* to an empty value. |

**setenv** *VAR value*
Sets the environment variable VAR to *value*, where *value* is not a dbx variable.

**setenv** *VAR $var*
Sets the environment variable *VAR* to *$var*, where *$var* is a *dbx* variable.

**setenv** *VAR "charstring"*
Sets the environment variable VAR to charstring.

| | |
|---|---|
| **sh** | Invokes a subshell. To return to *dbx* from the subshell, enter **exit** at the command line, or otherwise terminate the subshell. |
| **sh** *com* | Executes the specified shell command. *dbx* interprets the remainder of the line as a command to pass to the spawned shell process, unless you enclose the command in double-quotes or you terminate your shell command with a semicolon (;). |
| **showpgrp** | Shows the group process list and the group history. |

**showproc** [*pid* | **all**]
Shows processes already in the *dbx* process pool or processes that *dbx* can control. If you provide no arguments, *dbx* lists the processes it already controls. If you provide a *pid*, *dbx* displays the status of the specified process. If you use argument "all," *dbx* lists all the processes it controls as well as all those processes it could control but that are not yet added to the process pool.

**showthread** [**full**]
Prints brief status information about the current thread. If the **full** qualifier is included, prints full status information.

**showthread** [**full**] [**thread**] {*number* | *$no* | **all**}
Prints brief status information about the thread identified by *number* or the value of *$no*, or all threads associated with the debug session. If the **full** qualifier is included, prints full status information. The **thread** qualifier does not affect the output, but it is allowed so the syntax can be the same as that for other commands that use the **thread** clause.

| | |
|---|---|
| **source** [*file*] | Executes *dbx* commands from *file*. |
| **status** | Displays all breakpoints, traces, and conditional commands. |

**step** [**n**]        Executes the specified number of lines of source code, stepping into procedures. If you do not provide an argument, *step* executes one line. If *step* encounters any breakpoints, it immediately stops execution.

**stepi**        Single steps one machine instruction, stepping into procedures (as called by *jal* and *jalr*). If *stepi* encounters any breakpoints, it immediately stops execution.

**stepi** [**n**]        Executes the specified number of machine instructions, stepping into procedures (as called by **jal** and **jalr**).

**stop at**        Set a breakpoint at the current source line.

**stop at line**        Sets a breakpoint at the specified source line.

**stop expression**

Inspects the expression. If the expression is type pointer, checks the data being pointed at. Otherwise, checks the 32 bits at the address given by the expression.

**stop in procedure**

Sets a breakpoint to stop execution upon entering the specified procedure. Execution will stop in all inlined or cloned instances of the procedure.

**stop** [**expression** | **variable**]

Inspects the value before executing each source line. If the expression is of type pointer, look at the data pointed to and watch until it changes.

If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stop** [**expression** | **variable**] **at line**

Inspects the value at the given source line. Stops if the value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stop** [**expression** | **variable**] **in procedure**

Inspects the value at every source line within a given procedure. Stops if the value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes.

**148**

If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stop `if` expression**

Evaluates the expression before executing each source line. Stops if the expression is true.

**stop `at` line `if` expression**

Evaluates the expression at the given source line. Stops if the expression is true.

**stop `in` procedure `if` expression**

Evaluates the expression at every source line within a given procedure. Stops if the expression is true.

**stop [expression1 | variable] `if` expression2**

Tests both conditions before executing each source line. Stops if both conditions are true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stop [expression1 | variable] `at` line `if` expression2**

Tests both conditions at the given source line. Stops if both conditions are true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stop [expression1 | variable] `in` procedure `if` expression2**

Tests both conditions at every source line within a given procedure. Stops if both conditions are true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stopi at**    Sets an unconditional breakpoint at the current machine instruction.

**stopi at address**

Sets an unconditional breakpoint at the specified address (for machine-level debugging).

**`stopi` `in` procedure**

Sets an unconditional breakpoint to stop execution upon entering the specified procedure (for machine-level debugging).

**`stopi` [expression | variable]**

Inspects the value before executing each machine instruction and stops if the value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**`stopi` [expression | variable] `at` `address`**

Inspects the value when the program is at the given address and stops if the value has changed (for machine-level debugging).

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**`stopi` [expression | variable] `in` procedure**

Inspects the value at every machine instruction within a given procedure and stops if the value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**`stopi` `if` expression**

Evaluates the expression before executing each machine instruction and stops if the expression is true.

**`stopi` `at` address `if` expression**

Evaluates the expression at the given address and stops if the expression is true (for machine-level debugging).

**`stopi` `in` procedure `if` expression**

Evaluates the expression at every machine instruction within a given procedure and stops if the expression is true.

**stopi** [**expression1** | **variable**] **if expression2**

Tests both conditions before executing each machine instruction. Stops if both conditions are true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stopi** [**expression1** | **variable**] **at address if expression2**

Tests both conditions at the given address (for machine-level debugging). Stops if both conditions are true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**stopi** [**expression1** | **variable**] **in procedure if expression2**

Tests the expression each time that the given variable changes within the given procedure.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**suspend**

Suspends the active process if it is running. If it is not running, this command does nothing. If you use the keyword **all**, suspends all active processes.

**suspend** *pgrp*

Suspends all the processes in *pgrp*.

**suspend pid** *pid*

Suspends the process *pid* if it is in the *dbx* process pool. If it is not running, this command does nothing.

**syscall**

Prints a summary of the catch and ignore status of all system calls. The summary is divided into four sections: 1) caught at call, 2) caught at return, 3) ignored at call, and 4) ignored at return.

**syscall catch** [{**call** | **return**}]

Prints a list of all system calls caught upon entry (**call**) or return (**return**). If you provide neither the **call** nor **return** keyword, *dbx* lists all system calls that are caught.

**syscall ignore** [{**call** | **return**}]

> Prints a list of all system calls not caught upon entry (**call**) or return (**return**). If you provide neither the **call** nor **return** keyword, *dbx* lists all system calls that are ignored.

**syscall catch** {**call** | **return**} {**system_call** | **all**}

> Sets a breakpoint to stop execution upon entering (**call**) or returning from (**return**) the specified system call. Note that you can set *dbx* to catch both the call and the return of a system call.
>
> If you use the keyword **all** rather than giving a specific system call, *dbx* catches all system calls.

**syscall ignore** {**call** | **return**} {**system_call** | **all**}

> Clears the breakpoint to stop execution upon entering (**call**) or returning from (**return**) the specified system call.
>
> If you use the keyword **all** rather than giving a specific system call, *dbx* clears the breakpoints to stop execution upon entering (*call*) or returning from (**return**) all system calls.

**tag** *procedure*    Searches the tag file for the given procedure.

**trace variable**    Whenever the specified variable changes, *dbx* prints the old and new values of that variable.

**trace procedure**Prints the values of the parameters passed to the specified procedure whenever your program calls it. Upon return, *dbx* prints the return value.

**trace** [**expression**|**variable**] **at line**

> Whenever your program reaches the specified line, *dbx* prints the value of the variable if its value has changed.
>
> If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**trace** [**expression**|**variable**] **in procedure**

> Whenever the variable changes within the procedure, *dbx* prints the old and new values of that variable.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**trace** [**expression1** | **variable**] **at line if expression2**

Prints the value of the variable (if changed) whenever your program reaches the specified line and the given expression is true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**trace** [**expression1** | **variable**] **in procedure if expression2**

Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable, if the given expression is true.

If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**tracei** [**expression** | **variable**]

Whenever the specified variable changes, *dbx* prints the old and new values of that variable. (For machine-level debugging.)

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**tracei procedure**

This command is equivalent to entering **trace procedure**. (For machine-level debugging.)

**tracei** [**expression** | **variable**] **at address**

Prints the value of the variable whenever your program reaches the specified address. (For machine-level debugging.)

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**tracei** [**expression** | **variable**] **in procedure**

> Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable. (For machine-level debugging.)

> If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**tracei** [**expression1** | **variable**] **at address if expression2**

> Prints the value of the variable whenever your program reaches the specified address and the given expression is true. (For machine-level debugging.)

> If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**tracei** [**expression1** | **variable**] **in procedure if expression2**

> Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable, if the given expression is true. (For machine-level debugging.)

> If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**unalias** *alias*    Removes the specified alias.

**unrecord** *session1* [**,** *session2* ... ]

> Turns off the specified recording session(s) and closes the file(s) involved.

**unrecord all**    Turns off all recording sessions and closes all files involved.

**unset** *var*    Removes the specified *dbx* variable.

**unsetenv** *VAR*    Removes the specified environment variable.

**up** [*num*]    Moves up the specified number of activation levels in the stack. The default is one level.

**use** [**dir** ...]    If you provide one or more directories, *dbx* replaces the source directory
                        list with the directories that you provide.

                        If you do not provide any directories, *dbx* displays the current source
                        directory list.

**wait**                Waits for the active process to stop for an event.

**wait pid** *pid*      Waits for the process *pid* to stop for an event.

**waitall**             Waits for any process currently running to breakpoint or stop for any
                        reason.

**whatis** *name*       Prints the type declaration for *name*.

**when** [**expression** | **variable**] **{command-list}**
                        Inspects the value before executing each source line. If it has changed,
                        executes the command list.

                        If the expression is of type pointer, look at the data pointed to and
                        watch until it changes. If the expression is not of type pointer, look at
                        the 32 bits at that address (assume the expression evaluates to an
                        address).

**when** [**expression** | **variable**] **at line {command-list}**
                        Inspects the value at the given source line. If it has changed, executes the
                        command list.

                        If the expression is of type pointer, look at the data pointed to and
                        watch until it changes. If the expression is not of type pointer, look at
                        the 32 bits at that address (assume the expression evaluates to an
                        address).

**when** [**expression** | **variable**] **in procedure {command-list}**
                        Inspects the value at every source line within a given procedure. If it has
                        changed, executes the command list.

                        If the expression is of type pointer, look at the data pointed to and
                        watch until it changes. If the expression is not of type pointer, look at
                        the 32 bits at that address (assume the expression evaluates to an
                        address).

**when if expression {command-list}**
                        Evaluates the expression before executing each source line. If it is true,
                        executes the command list.

**when at line if expression {command-list}**
> Evaluates the expression at the given source line. If it is true, executes the command list.

**when in procedure if expression {command-list}**
> Evaluates the expression at every source line within a given procedure. If it is true, executes the command list.

**when [expression1 | variable] if expression2 {command-list}**
> Checks if the value of the variable has changed. If it has changed and the expression is true, executes the command list. If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**when [expression1 | variable] at line if expression2 {command-list}**
> Checks if the value of the variable has changed each time the line is executed. If the value has changed and the expression is true, executes the command list. If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**when [expression1 | variable] in procedure if expression2 {command-list}**
> Checks if the value of variable has changed at each source line of the given procedure. If the value has changed and the expression is true, executes the command list. If **expression1** is of type pointer, look at the data pointed to and watch until it changes. If **expression1** is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**wheni if** *expression* {*command-list*}
> Evaluates the expression before executing each machine instruction. If the expression is true, executes the command list.

**wheni at** *address* **if** *expression* {*command-list*}
> Evaluates the expression at the given address. If the expression is true, executes the command list. (For machine-level debugging.)

**wheni in** *procedure* **if** *expression* {*command-list*}
> Evaluates the expression in the given procedure. If the expression is true, executes the command list. (For machine-level debugging.)

**156**

**wheni** *variable* **at** *address* **if** *expression* {*command-list*}

Tests both conditions at the given address. If the conditions are true, executes the command list. (For machine-level debugging.) If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

**wheni** *variable* **in** *procedure* **if** *expression* {*command-list*}

Tests both conditions at every machine instruction within a given procedure. If they are true, executes the command list.

**where**          Print a stack trace. **t** is an alias for the **where** command.

**whereis** *name*  Prints the fully qualified names of all versions of *name*. The range of objects examined is determined by *$whereisdsolimit.*

**which** *name*     Prints the fully qualified name of the active version of *name*.

**whichobj** *variable*

Lists the dynamic shared objects that contain *variable*.

# Predefined Aliases

Table B-1 lists all predefined *dbx* aliases. You can override any predefined alias by redefining it with the *alias* command or by removing it with the *unalias* command.

**Table B-1**      Predefined Aliases

| Alias | Definition | Description |
|-------|-----------|-------------|
| a | assign | Assigns the specified expression to the specified program variable or register. |
| b | stop at | Sets a breakpoint at the specified line. |
| bp | stop in | Sets a breakpoint in the specified procedure. |
| c | cont | Continues program execution after a breakpoint. |
| d | delete | Deletes the specified item from the status list. |
| dir | directory | Displays the current source directory list. If you specify one or more directories, those directories are added to the end of the source directory list. |
| e | file | Displays the name of the currently selected source file. If you specify a file, this command makes the specified file the currently selected source file. |
| f | func | Moves to the specified procedure (activation level) on the stack. If you specify no procedure or expression, *dbx* prints the current activation level. |
| g | goto | Goes to the specified source line. |
| h | history | Lists all the items currently in the history list. |

**Table B-1 (continued)**     Predefined Aliases

| Alias | Definition | Description |
|-------|------------|-------------|
| j | status | Lists all the currently set *stop, trace*, and *when* commands. |
| l | list | Lists the next *$listwindow* lines of source code beginning at the current line. |
| li | $curpc/10i; \ <br> set $curpc=$curpc+40 | Lists the next 40 bytes of machine instructions (approximately 10 instructions). |
| n | next | Executes the specified number of lines of source code, stepping over procedures. If you do not provide an argument, *dbx* executes only one line. |
| ni | nexti | Executes the specified number of lines of machine code, stepping over procedures. If you do not provide an argument, *dbx* executes only one instruction. |
| p | print | Prints the value of the specified variable or expression. |
| pd | printd | Prints the value of the specified variable in decimal. |
| pi | playback input | Replays *dbx* commands saved in the specified file. If you do not specify a file, *dbx* uses the temporary file specified by *$defaultin*. |
| po | printo | Prints the value of the specified variable or expression in octal. |
| pr | printregs | Prints values contained in all registers. |
| px | printx | Prints the value of the specified variable or expression in hexadecimal. |
| q | quit | Quits *dbx*. |
| r | rerun | Runs the program again using the arguments specified for the last *run* command executed. |

**Table B-1 (continued)**    Predefined Aliases

| Alias | Definition | Description |
|---|---|---|
| ri | record input | Records to the specified file all the input you give to *dbx*. If you do not specify a file, *dbx* creates a temporary file. The name of the file is specified by *$defaultin*. |
| ro | record output | Records all *dbx* output to the specified file. If no file is specified, records output to a temporary file. The name of the file is specified by *$defaultout*. |
| s | step | Executes the specified number of lines of source code, stepping into procedures. If you do not provide an argument, *dbx* executes only one line. |
| S | next | Executes the specified number of lines of source code, stepping over procedures. If you do not provide an argument, *dbx* executes only one line. |
| si | stepi | Executes the specified number of lines of machine code, stepping into procedures. If you do not provide an argument, *dbx* executes only one instruction. |
| Si | nexti | Executes the specified number of lines of machine code, stepping over procedures. If you do not provide an argument, *dbx* executes only one instruction. |
| source | playback input (pi) | Replays *dbx* commands saved in the specified file. If no file is specified, *dbx* uses the temporary file specified by *$defaultin*. |
| t | where | Does a stack trace to show the current activation levels. |
| u | list $curline-9:10 | Lists a window of source code showing the nine lines before the current code line and the current code line. This command does not change the current code line. |

**Table B-1 (continued)**     Predefined Aliases

| Alias | Definition | Description |
|-------|-----------|-------------|
| w | list $curline-5:10 | Lists a window of source code around the current line. This command shows the four lines before the current code line, the current code line, and five lines after the current code line. This command does not change the current code line. |
| W | list $curline-10:20 | Lists a window of source code around the current line. This command shows the nine lines before the current code line, the current code line, and 10 lines after the current code line. This command does not change the current code line. |
| wi | $curpc-20/10i | Lists a window of assembly code around the program counter. |

# Predefined *dbx* Variables

Predefined *dbx* variables are listed in Table C-1. The predefined variable names begin with "$" so that they do not conflict with variable, command, or alias names.

**Table C-1**     Predefined *dbx* Variables

| Variable | Default | Description |
| --- | --- | --- |
| *$addrfmt* | "0x%x" | Specifies the format for addresses. This can be set to any format valid for the C language **printf**(3S) function. |
| *$addrfmt64* | "0x%llx" | Specifies the format for 64-bit addresses. This can be set to any format valid for the C language **printf**(3S) function. |
| *$assignverify* | 1 | If nonzero, the new value of a program variable will be displayed after the *assign* command. |
| *$casesense* | 2 | If 0, symbol names are case sensitive. If 1, symbol names are not case sensitive. If 2, the case sensitivity of symbol names depends on the case sensitivity of the language in which the symbol was defined. |
| *$ctypenames* | 1 | If 1, the words "unsigned," "short," "long," "int," "char," "struct," "union," and "enum" are keywords usable only in type casts. If 0, "struct," "union," and "enum" are ordinary words with no predefined meaning (in C modules, the others are still known as C types). |
| *$curevent* | | The last event number as seen by the *status* command. |
| *$curline* | | The current line in the source code being executed. |

**Table C-1 (continued)**    Predefined *dbx* Variables

| Variable | Default | Description |
|---|---|---|
| *$curpc* | | The current program counter. |
| *$cursrcline* | | The current source listing line plus one. |
| *$defaultin* | | The name of the file that *dbx* uses when the *record input* or the *playback input* command is executed with no argument. |
| *$defaultout* | | The name of the file that *dbx* uses when the *record output* or the *playback output* command is executed with no argument. |
| *$editor* | vi | The name of the editor to invoke (with the *edit* command). Default value is set to the value of the *EDITOR* environment variable. If *EDITOR* missing, it defaults to *vi*. |
| *$fp_precise* | 0 | When nonzero, *dbx* runs programs on R8000 processors in floating point precise mode, allowing accurate floating point exceptions. By default, R8000 floating point interrupts are asynchronous and reported program counter values are useless for debugging. For more information about floating point precise mode, see the **syssgi**(2) reference page section on SGI_SET_FP_PRECISE. |
| *$framereg* | 1 | If 1, all references to registers are to the registers of the current activation level. If 0, all references are to the hardware registers (the registers of activation level 0). |
| *$groupforktoo* | 0 | If 0, adds only processes created with the **sproc**(2) system call to the process group list automatically. If 1, then adds processes created with either the **fork**(2) or **sproc** system calls to process group list. |
| *$hexchars* | 0 | If nonzero, outputs characters in hexadecimal, using C format "%x". This affects char type variables, including those in structures. It does not affect arrays of characters, which are printed using the "%.*s" format. |

**Table C-1 (continued)**     Predefined *dbx* Variables

| Variable | Default | Description |
| --- | --- | --- |
| *$hexdoubles* | 0 | If nonzero, *dbx* displays floating point and double-precision variables both as literals and as hexadecimal representations of the bit pattern. |
| *$hexin* | 0 | If nonzero, input constants are assumed to be in hexadecimal. This overrides *$octin*. |
| *$hexints* | 0 | If nonzero, outputs integers in hexadecimal format. This overrides *$octints*. |
| *$hexstrings* | 0 | If nonzero, outputs strings and arrays in hexadecimal. For character arrays, if nonzero, the null byte is not taken as a terminator. Instead, prints the entire array (or *$maxlen* values, whichever is less). If 0, then a null byte in a C or C++ character array is taken as the end of the array (the length of the array and *$maxstrlen* can terminate the array printing before a null byte is found). |
| *$historyevent* | | The current history line number. |
| *$lastchild* | | The process ID of the last child process created by a **fork** or **sproc** system call. |
| *$lines* | 100 | The number of lines in the history list. |
| *$listwindow* | 10 | Specifies how many lines the *list* command lists. |
| *$maxstrlen* | 128 | Maximum length printed for zero-terminated char strings and arrays. Prints char arrays for array-length, *$maxstrlen* bytes, or up to a null byte, whichever comes first (see *$hexstrings*). |
| *$mp_program* | 0 | If 0, *dbx* treats calls to **sproc** in the same way as it treats calls to **fork**. If 1, child processes created by calls to **sproc** are allowed to run; they block on multiprocessor synchronization code emitted by mp Fortran code. When you set *$mp_program* to 1, mp Fortran code is easier to debug. |

**Table C-1 (continued)**     Predefined *dbx* Variables

| Variable | Default | Description |
| --- | --- | --- |
| *$newevent* | 0 | After every command creating an event, this variable is set to the event's number. The *$newevent* variable is useful in writing scripts that do not use hard-coded event numbers. |
| *$newpgrpevent* | 0 | Stores the number of the latest **pgrp** event created by *stop[i]*, *trace[i]*, and *when[i]*... **pgrp**. Useful when writing scripts . |
| *$nonstop* | 0 | Only used with *addproc* or with *dbx* options **-p** and  **-P**. If 0, the process that is the argument of the command is stopped; if 1, the process is not stopped. In either case the process state is not changed. If the you start dbx with the –**N** option, then $*nonstop* = 1. |
| *$octin* | 0 | If nonzero, assumes input constants are in octal (*$hexin* overrides *$octin*). |
| *$octints* | 0 | If nonzero, outputs integers in octal format (*$hexints* takes precedence). |
| *$page* | 1 | Specifies whether or not to page when *dbx* output scrolls information off the current screen. A nonzero value turns on paging; a 0 turns it off. |
| *$pager* | more | The name of the program used to display output from *dbx*. |
| *$pagewidth* | 80 | The width of the window in characters (assumes a fixed-width font). Used by *dbx* to calculate how many screen lines are output. *dbx* never inserts newlines; the window software wraps the lines. |
| *$pagewindow* | 23 | Specifies how many lines print when information is longer than one screen. This can be changed to match the number of lines on any terminal. If set to 0, 1 is used. |

**166**

**Table C-1 (continued)**     Predefined *dbx* Variables

| Variable | Default | Description |
|---|---|---|
| *$pendingtraps* | 0 | If nonzero, allows traps that cannot be satisfied immediately to wait until they can be satisfied. This is useful for debugging programs that use DSOs, as it allows setting breakpoints before the **dlopen()** call. When set to nonzero, mistyped procedure names are not flagged and cause a pending trap to be set. |
| *$piaddtohist* | 1 | If 1, adds commands read from files using the *playback input* command to the command history. If 0, does not add the commands to the history. |
| *$pid* | | The current process for kernel debugging (**–k**). |
| *$pid0* | | Set by *dbx* to the process ID of the running process (also called the object file). |
| *$pimode* | 0 | If 1, *dbx* prints the commands read from files using the *playback input* command. If 0, *dbx* does not print the commands. In either case, *dbx* prints the output resulting from such commands. |
| *$printdata* | 0 | Used when disassembling. If 1, prints register contents alongside disassembled instructions. If 0, just prints disassembled instructions. |
| *$print_exception_frame* | 0 | If nonzero, the display of a kernel exception frame by the *dump* or *where* commands includes information that you can use to find the contents of the kernel registers at the time of the fault. |
| *$printwhilestep* | 0 | If 0, prints only the next line to be executed. If nonzero, prints each line that is executed while it single steps. |
| *$printwide* | 0 | If 0, prints arrays, unions, structures and classes one element per line. If nonzero, prints arrays compactly (wide). |

**Table C-1 (continued)**     Predefined *dbx* Variables

| Variable | Default | Description |
|---|---|---|
| *$procaddr* | | This variable applies only if you invoke *dbx* with the **–k** option (that is, it is not available unless you are doing kernel debugging). Whenever *$pid* is set, *dbx* sets *$procaddr* to the address of the process table entry for that process. |
| *$prompt* | dbx | The prompt for *dbx*. |
| *$promptonfork* | 0 | If 0, *dbx* does not add the child process to the process pool. Both the child process and the parent process continue to run. |
| | | If 1, *dbx* stops the parent process and asks if you want to add the child process to the process pool. If you answer yes, then *dbx* adds the child process to the pool and stops the child process; if you answer no, *dbx* allows the child process to run and does not place it in the process pool. |
| | | If 2, *dbx* automatically stops both the parent and child processes and adds the child process to the process pool. |
| *$regstyle* | 0 | If 0, *dbx* uses the alternate form of the register name (for example, "zero" instead of "r0" and "t1" instead of "r9"). If nonzero, *dbx* uses the machine name ("r0" through "r31"). |
| *$repeatmode* | 0 | If nonzero, entering a null line (entering a newline on an empty line) repeats the last command. If 0, *dbx* performs no action. |
| *$rimode* | 0 | If 1, *dbx* records commands you enter in addition to output when using the *record output* command. If 0, *dbx* does not copy the commands. |
| *$shellparameters* | " " | A string that is added by *run* to the command line it passes to the command interpreter, *SHELL*. Use *$shellparameters* to disable spawning of subshells by the initialization file of a non-standard shell. |

**Table C-1 (continued)**     Predefined *dbx* Variables

| Variable | Default | Description |
|---|---|---|
| *$showbreakaddrs* | 0 | If nonzero, show the address of each breakpoint placed in the code each time it is placed. Removal of the breakpoints is not shown. If multiple breakpoints are placed at one location, only one of the placements is shown. Since breakpoints are frequently placed and removed by *dbx*, the volume of output can be annoying when tracing. |
| *$showfilename* | 0 | If 0, *step*, *next*, and so on. do not show the source file name in the *dbx* message describing the stopped state. |
| | | If 1, prints just the base file name. |
| | | If 2, prints the full path. |
| | | If $stopformat is 1, $showfilename = 0 is treated as if $showfilename were 2. |
| *$sourcepathrule* | 0 | If 0, search for a source file by: a) using the pathname in the object file's debugging information; if the file is not found, then b) examine pathnames remapped by the *dir* or *use* command; if the file is still not found, then c) reduce full pathnames to base file names and search the list of directories created by the *dir* or *use* command. |
| | | If 1, permute the default source-file search sequence to: step b, step c, then step a. |
| | | If 2, use only steps b and c of the default source-file search sequence. |
| *$stacktracelimit* | 100 | Sets the maximum number of frames that will be examined by the *dump, func,* and *where* commands. |

**Table C-1 (continued)**     Predefined *dbx* Variables

| Variable | Default | Description |
|---|---|---|
| *$stdc* | 0 | If nonzero, attempts in *dbx* expressions to model exactly the promotion rules of ANSI C and ISO/IEC 9899 C (even to the point of matching float to float rather than converting all floating points to doubles). |
| | | If 0, promotes variables more like traditional pcc C (but promotions of 16-bit and 8-bit unsigned is to int, not unsigned int). |
| *$stepintoall* | 0 | If 0, *step* steps into all procedures that are compiled with debugging options **−g**, **−g2**, or **−g3** for which line numbers are available in the symbol table. Note that standard library routines are excluded. |
| | | If 1, in addition to the procedures above, steps into any procedures for which a source file can be found. Note that when you debug a source file compiled without symbols or compiled with optimization, the line numbers may jump erratically. |
| | | If 2, steps into all procedures. Note that if *dbx* cannot locate a source file, then it cannot display source lines as you step through a procedure. |
| *$stopformat* | 0 | If 0, stopping messages appear in the traditional IRIX *dbx* format, for example: |
| | | ```
stopped at [main:32 , 0x400000
main.c]
``` |
| | | If 1, messages appear in a more standard BSD *dbx* format: |
| | | ```
stopped in main at line 32 in file
"main.c"
``` |
| | | See affect on *$showfilename* also. |

**Table C-1 (continued)**  Predefined *dbx* Variables

| Variable | Default | Description |
| --- | --- | --- |
| *$tagfile* | tags | The name of a file of tags, as created by *ctags*(1). Used by the *tag* command. |
| *$whereisdsolimit* | 1 | If 1, *whereis* looks only in main object. If 0, *whereis* checks all objects. If *n*, *whereis* checks first *n* objects. |

# Index

wait for, 128, 129, 155

process group list
  adding processes, 133, 138
  deleting processes, 133, 141
  showing processes, 133, 147

process groups, 132
  breakpoints, 134
  group history, 134
  tracing, 134

process identification number (PID), 122

process pool, 122
  add processes, 125, 138
  delete processes, 126, 141
  display active process, 126, 138
  display processes, 124, 147
  kill active process, 129, 144
  kill processes, 129, 144
  resume active process, 127, 146
  select active process, 126, 138
  suspend active process, 127
  suspend processes, 127, 151

program stack. *See* stack

program variables. *See* variables, program

*$prompt*, 8, 168

prompt, 8, 168

*$promptonfork*, 97, 130, 168

*px*, 41, 47

## Q

qualifying program variable names, 43, 65, 87

quick start *duel*, 52

*quit* command, 13, 145

quitting *dbx*, 13, 145

quotation marks, 41, 46, 103

## R

*record* command, 35, 145

recording, displaying sessions, 35, 145

recording, ending, 33, 154

recording input, 32, 145

recording output, 32, 34, 146

*record input* command, 32, 33, 145, 164

*record output* command, 34, 146, 163

register names, 105, 168

registers, 105
  changing values, 109, 139
  displaying values, 66
  printing values, 66, 107, 145
  using values in expressions, 108

*$regstyle*, 107, 168

removing *dbx* variables, 25, 154

repeating commands, 25, 26, 137, 138, 168

*$repeatmode*, 26, 137, 168

*rerun* command, 3, 9, 10, 146

resume active process, 127, 146

*resume* command, 85, 127, 146

*return* command, 100, 146

*-R* flag, 8

*-r* flag, 8

*$rimode*, 34, 146, 168

*run* command, 3, 9, 146

running process, wait for, 129, 155

running programs, 8, 9, 10, 146

## S

scope of program variables, 46, 65, 68, 69

scripts, 35

search backward (?) command, 21, 137

search forward (/) command, 21, 137

*whereis*  command,  43, 45, 63, 157
*$whereisdsolimit*,  64, 171
*which*  command,  43, 45, 63, 157
*whichobj* command,  6, 157

## Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-0906-110.

Thank you!

## Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
    - On the Internet: techpubs@sgi.com
    - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

Technical Publications
Silicon Graphics, Inc.
2011 North Shoreline Boulevard, M/S 535
Mountain View, California  94043-1389