dbx Reference Manual

IRIS-4D Series

SiliconGraphics
Computer Systems

Document number: 007-0906-030

dbx Reference Manual

Version 3.0

Document Number 007-0906-030

12/90

Technical Publications:

Wendy Ferguson
Beth Fryer
Claudia Lohnes
Robert Reimann

Engineering:

David Anderson
Greg Boyd
Jeff Doughty
Jim Terhorst

dbx Reference Manual
Version 3.0
Document Number 007-0906-030

Silicon Graphics, Inc.
Mountain View, California

Contents

List of Tables

1. Introduction

This manual explains how to use the source level debugger, *dbx*, and provides both user and reference information about operating the debugger. The debugger works for C, FORTRAN 77, Pascal, assembly language, and machine code.

1.1 Using This Manual

This book is divided into the chapters listed below. If you're new to *dbx*, you might want to read the entire book. Chapter 2, "Getting Started," contains a section describing the few basic commands necessary for debugging a simple program.

For more experienced users looking for specific information, there is an overview of what's discussed at the beginning of each chapter. Also, there is a list of each command covered in the chapter, its syntax, and a brief description of what the command does. For more detailed information, just look in that chapter.

Overview of This Manual

The following paragraphs briefly describe the contents of each chapter and appendix in this manual.

Chapter 2, "Getting Started," introduces new users to the debugger. It contains a general discussion of the debugging process. This chapter also offers tips for people who are new to source-level debugging. If you're experienced at using debuggers, you might want to skip this part.

Chapter 3, "Running *dbx*," explains how to run the debugger. In particular, it explains how to compile a program for debugging, use the *corefile* and *givenfile* commands, and how to invoke and quit *dbx*.

Chapter 4, "Commands and Expressions," describes *dbx* commands, expression precedence, data types, constants, and registers.

Chapter 5, "The *dbx* Monitor," explains how to use *history*, edit the command line, and type multiple commands.

Chapter 6, "Controlling *dbx*," explains how to work with variables, how to create command aliases, record and playback input and output, invoke a shell from *dbx*, and use the *dbx* status feature.

Chapter 7, "Examining Source Programs," explains how to specify source directories, move to a specified procedure or source file, list source code, search through the source code, call an editor from *dbx*, print symbolic names, and print type declarations.

Chapter 8, "Controlling Your Program," explains how to run and rerun a program, execute single lines of code, return from procedure calls, start at a specified line, continue after a breakpoint, and assign values to program variables.

Chapter 9, "Setting Breakpoints," explains how to set and remove breakpoints and continue executing a program after a breakpoint.

Chapter 10, "Examining Program State," explains how to print stack traces, move up and down the activation levels of the stack, print register and variable values, and print information about the activation levels in the stack.

Chapter 11, "Debugging at the Machine Level," describes the *dbx* commands for debugging machine code. It also explains how to examine memory addresses and disassemble source code.

Chapter 12, "Multi-Process Debugging," describes the *dbx* commands for seizing control of and debugging currently active processes.

Appendix A, "*dbx* Command Summary," lists commands, aliases, syntax, and gives a brief description of the commands.

Appendix B, "Sample Program," shows an example of a *dbx* program.

Appendix C, "Questions and Problems," has solutions to some *dbx* questions and problems.

1.2 Conventions

This document uses the standard UNIX convention for referring to entries in the IRIX™ documentation. The entry name is followed by a section number in parentheses. For example, *cc*(1) refers to the *cc* manual entry in Section 1 of the *IRIX User's Reference Manual.*

In command syntax descriptions and examples, words in *italics* represent variable parameters, which you replace with the string or value appropriate for the application. Square brackets ([]) around an argument indicate that the argument is optional.

In text descriptions, file names and IRIX commands are printed in *italics*.

1.3 Relevant Documentation

You may find useful information to help you plan and set up your network in these documents:

- *Learning to Debug With edge*, (*edge* is Silicon Graphics Inc.'s graphical interface to *dbx*)
- *IRIX User's Reference Manual*, *dbx*(1) man page

2. Getting Started

This chapter introduces *dbx*. It offers a general discussion of basic debugging commands, as well as some tips about how to approach a debugging session. The first section explains the basic commands and procedures for debugging a simple program. Experienced users might want to skip this chapter.

2.1 Debugging a Simple Program

It's easy to debug a simple program. For example, assume that you want to debug a simple program such as the one in Appendix B, *anthrax.c*. To use *anthrax.c* as an example, first change the call to *printline* from:

```
printline(&line1);
```

to

```
printline(line1);
```

Now the *printline* call is incorrect, so the program will write to a *core* file (called a "coredump") when you run it.

Here's how to debug this new, broken, version of *anthrax.c*:

1. Compile the program:

   ```
   cc -g anthrax.c
   ```

2. Run the program (it will coredump):

   ```
   a.out anthrax.c
   ```

3. Start *dbx*:

   ```
   dbx a.out core
   ```

4. Look at the stack trace:

   ```
   t
   ```

5. Now you know the problem is in *printline*. Look at the source where it coredumped:

   ```
   W
   ```

6. Print the pointer:

   ```
   print pline
   ```

   The *pline* looks like a list of characters -- something's obviously wrong.

7. Move up the stack trace to look at the caller:

   ```
   up
   ```

8. Look at the source where it coredumped:

```
W
```

9. Print the contents of line1:

```
print line1
```

10. Stop at the start-up point:

```
stop in printline
```

11. Run the program:

```
run
```

12. It stops in *printline*. What is the pointer?

```
print pline
```

    It looks just like when it coredumped. Obviously, it needs a pointer.

13. Now you can quit *dbx*:

```
q
```

2.2 Source-Level Debuggers

dbx is a source-level debugger. Source-level debuggers let you trace problems in your program object 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.

2.3 Activation Levels

Activation levels define the currently active scopes (usually procedures) on the stack. The activation stack is a list of procedure calls. The most recently called procedure or block is numbered 0. The next procedure called is numbered 1. The last activation level is always the main program.

Activation levels can also consist of blocks that define local variables within procedures. You see activation levels when you do stack traces (see the *where* command) and when you move around the activation stack (see the *up*, *down*, and *func* commands).

2.4 Locating the Failure Point

Even if your program compiles successfully, it still might 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 running program.

The first step in figuring out why a program fails is to look at the terminating signal. In particular, it's often useful to know which line generates the signal.

To determine which line generates the terminating signal, follow these steps:

1. Copy the core file to the directory containing the failed program.
2. Invoke *dbx* for the failed program. *dbx* automatically reads in the local core file.
3. Do a stack trace using the *where* command to locate the failure point.

For example, suppose you're debugging a program called *madmax*. If the program fails on line 83, the *where* command returns this message:

```
0 main(argc = 4, argv = 0x7fffc794) ["madmax.c":83, 0x400228]
```

Note: If you don't strip symbol table information from your program object, you can do a stack trace on a program that wasn't compiled using the debug flag, **-g**.

2.5 Debugging Your Programs

Whether you are debugging a crashed or a normally terminating program, you need to know the value of program variables at various points throughout the code. To do this, you should set several *breakpoints* in your program. A breakpoint stops program execution and lets you examine the state of the program at that point. Chapter 9, "Setting Breakpoints," discusses this topic in more detail.

Set breakpoints throughout your program. Look at your program carefully to determine where there are likely to be problems, and be sure to set breakpoints in these problem areas. If the program crashes, first determine which line caused it to crash, then set a breakpoint just before that line.

There are many *dbx* commands you can use to trace a variable's value. This is the simplest method for tracing a program variable:

1. Use *stop* to set breakpoints in the program at locations where you want to print the value of a variable. See Chapter 9 for more details.
2. Use *run* or *rerun* to run the program under *dbx*. The program pauses at any breakpoint you set.
3. When the program pauses at a breakpoint, use *print* to print the value of the program variables you want to follow.
4. Use *cont* to continue execution past a breakpoint. However, you cannot continue execution past a line that crashes the program.

Chapter 4, "Commands and Expressions," explains these commands in more detail. Appendix A, "*dbx* Command Summary," lists all the *dbx* commands.

2.6 Studying a New Program

dbx is a useful tool for examining 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.* This is the procedure:

1. Use *stop* to set breakpoints in the program. When you execute the program under *dbx*, it stops execution at the set 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 *run* and *rerun* to run the program under *dbx*. The program stops at each breakpoint.
3. Use *print* to print the value of a program variable at a breakpoint.

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

   - *next* executes the next line; if it is a procedure, *next* executes it but does not step down into it.
   - *step* executes the next line of the program. If the next line is a procedure, *step* steps down into the procedure.
   - *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.

   For more information about these commands, see Chapter 4, "Commands and Expressions," or Appendix A, "*dbx* Command Summary."

2.7 Avoiding Common Pitfalls

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

- If the debugger won't display variables, recompile the program with the `-g` compiler flag. If your program is a FORTRAN program and you started *dbx* with `-F`, you must use the *readsyms* command to make variables in common visible.
- If the debugger's listing seems confused, try separating the lines of source code into logical units. The debugger might get confused if there's more than one source statement 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 *include* files and macros as single lines. To debug code from an *include* file, remove the source from the *include* file and compile the source as part of your source program. To debug a macro, expand the macro in the source code.

3. Running *dbx*

This chapter explains how to run *dbx*—specifically how to:

- compile your program for debugging
- create a *.dbxinit* command file (optional)
- invoke *dbx* from the shell
- use the *corefile* and *givenfile* commands
- get on-line help
- end your debugging session

3.1 Compiling Your Program

Before using *dbx*, compile the program using the **-g** option. The **-g** compile option inserts symbol table information into your program object so the local variables are visible to *dbx*. *dbx* uses the symbol table information to list local variables and to find source lines.

If you use *dbx* to debug code that was not compiled using the **-g** flag, local variables are invisible to *dbx*, and source lines may appear to jump around oddly as a result of various optimizations. Since it's harder to debug code without reliable references to lines of source code, always recompile a program you want to debug using the **-g** flag, if possible.

3.2 Building a *.dbxinit* Command File

You can use your system's editor to create a *.dbxinit* command file. This is a file in which you list various *dbx* commands, which are then automatically executed when you invoke *dbx*. You can put any *dbx* command in the *.dbxinit* file. If any of these commands require input, the system prompts you for it.

When you invoke *dbx*, it 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 you set the IRIX system HOME environment variable.)

Here's an example of a *.dbxinit* file:

```
set $page = 5
set $lines = 20
set $prompt = "DBX>"
alias du dump
```

Note: The *$page*, *$lines*, and *$prompt* are all *dbx* system variables. For complete description of these and other *dbx* system variables, see "Predefined *dbx* Variables" in Chapter 6.

3.3 Invoking *dbx*

To invoke *dbx* from the shell command line, type **dbx**.

The syntax is:

dbx [*options*] [*program* [*corefile*]]

Examples

```
dbx a.out
dbx a.out /usr/tmp/core
dbx -P myprog
dbx -p 3409
```

There are several optional parameters listed on the following page. After invocation, *dbx* sets the current function to the first procedure of the program. If you specify *corefile*, *dbx* lists the point of program failure. For core files, you can do stack traces and look at the code; however, you cannot force the program to execute past the line that caused it to crash.

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

Use the environment variable *DBXINIT* (see the previous section) to hold *dbx* command line options. The debugger inserts the contents of *DBXINIT* before the command line options. This is most useful for options not recognized by *edge*(1), since it provides a way to pass options to *dbx* even if *edge* does not recognize them. Currently the options **-e** and **-F** are recognized by *dbx* but not by *edge*.

Since *program* is given on the command line, the program file named there can be referred to as the *givenfile*. (See the *givenfile* and *corefile* commands for more information.)

The following table lists specifications for *dbx* options.

Option	Select this option to...
`-I` *dir*	Tell *dbx* to look in the specified 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*, change the directories searched for source files with the *use* and *dir* commands.
`-c` *file*	Select a command file other than *dbxinit*.
`-i`	Use interactive mode. This option does not treat #s as comments in a file. It also prompts for source even when it reads from a file and has extra formatting, as if for a terminal.
`-r` *program* [*arg*]	Run the named program upon entering *dbx*, using the specified arguments. You cannot specify a core file with `-r`.
`-p` *PID #*	Debug the process specified by the *PID* number.
`-P` *name*	Debug the running process with the specified *name* (*name* as described in *ps*(1)).
`-e` *num*	Choose a larger size for the evaluation stack (as large as you want). The default stack size is 20,000 bytes. *num* = number of bytes.

Table 3-1. dbx Options

Option	Select this option to...
-F	Fastpath load the symbol table of the program to be debugged. This only helps programs using FORTRAN common -- for large FORTRAN programs, start-up time can be long. This option keeps *dbx* from loading the internal fields of *common*, making *dbx* startup much faster. After startup, the variables in *common* are invisible to *dbx*. Use the *readsyms* command to read in *common* symbols when and where you need them visible.
-k	Turn on kernel debugging. When debugging a running system, specify */dev/kmem* as the *core file*.
-C	Suppress the automatic truncation of C++ variable names. This option causes the full long names output by *cfront* to *cc* to be visible. This is useful when you think *dbx* has truncated a name improperly, but it makes C++ code more difficult to work with.

Table 3-1. (continued) dbx Options

3.4 Using the *corefile* and *givenfile* Commands

After you invoke *dbx*, you can debug a core file or another program from within the debugger by using the *corefile* or *givenfile* commands, respectively. You can use these commands without an argument or with a specified *file*.

3.4.1 The *corefile* Command

The command:

corefile

displays the name of the core file. If the corefile is currently used by *dbx*, this command displays program data. If you type:

corefile *file*

dbx uses the core file, *file* for program data. This command does the same thing as if you typed *dbx file corefile*.

3.4.2 The *givenfile* Command

The command:

givenfile

displays the name of the program being debugged. If you type:

givenfile *file*

dbx kills the current running processes and reads in *file*'s symbol table. This command does the same thing as if you typed *dbx file*.

3.5 On-line Help

The *dbx* command *help* shows the *dbx help* file. *dbx* displays the file using the command name given by the *$pager* environment variable. For ease of use, it is best to put the following command in your *.dbxinit* file:

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

If you don't like *vi*, just substitute the name of your favorite editor.

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

Under *edge*, the *help* file appears in a separate window, which you can leave on the screen for easy access. The format and colors of this window are controlled by the *dbx* debugger variable *$helpwshformat*. The *wsh*(1) options `-H` and `-c` are automatically added by *dbx*. See the *wsh*(1) for more information on its options. Use the window-menu "quit" entry on the *help* window to quit.

3.6 Quitting *dbx*

Use the *quit* command to end a debugging session, just type:

```
quit
```

4. Commands and Expressions

This chapter describes *dbx* commands and expressions and includes:

- strings
- expression and precedence
- data types and constants
- comments
- registers
- keywords
- C type-casts
- pointers and structures
- C++ function and variable names

4.1 Strings

In general, *dbx* recognizes the following escape sequences in quoted strings (per the standard C language usage):

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

dbx strips the escapes (\) and the surrounding quotes while creating the internal representation of the string. You can use the double-quote character (") to quote strings; *dbx* also recognizes the single-quote character (').

4.2 Qualifying Variable Names

dbx qualifies variables with the file, the procedure, a block, or a structure. You can manually specify the full scope of a variable by separating scopes with periods. For example, look at this expression:

```
mrx.main.i
```

Here, *mrx* is the current file, *main* is a procedure, and *i* is a variable.

A leading dot (a period at the beginning of the identifier) tells *dbx* that the identifier is a module (file). For example:

```
.mrx.main.i
```

Here, the period at the beginning of the line indicates that *mrx* is a file. In this example, *main* is a procedure and *i* is a variable.

The leading dot is very 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* contains a global variable called *mi* and a local variable, also called *mi*. Then, for example:

```
.mrx.mi          (refers to the global)
.mrx.mrx.mi      (refers to the local)
```

4.3 Expressions and Precedence

dbx recognizes expression operators from C and FORTRAN 77. Operators follow the C language precedence. The tables that follow show debugger operations, C language and FORTRAN operators, and data types.

Syntax	Description
"file" #exp	Uses specified line number *exp* in the specified file, *file*, and returns the address of that line.
proc #exp	Uses the specified line number, *exp*, in the specified procedure, *proc*, and returns the address of that line.
#exp	Takes the specified line number, *exp*, and returns the address of that line.

Table 4-1. Debugger Operations

Note: The table above describes the interactive case. In a script, use two pound (##) signs (for example, *##exp*).

Type	Operators
Unary	& + - * sizeof() ~ (type) (type*)
Binary	<< >> " ! == != <= >= < > & && | || + - * % [] ->

Table 4-2. C Language Operators

Note: The *sizeof* operator specifies the number of bytes retrieved to get an element, not (number_of_bits+7)/8.

Type	Operators
Unary	-
Binary	+ - * //

Table 4-3. FORTRAN Operators

Note: Use `//` (instead of `/`) for divide.

FORTRAN array subscripting must use square brackets, `[]`, instead of parentheses, `()`.

4.4 Data Types and Constants

dbx commands can use the built-in data types listed in the following table.

Data Types	Description
$address	pointer
$unsigned	unsigned pointer
$char	character
$boolean	boolean
$real	double precision real
$integer	signed integer
$float	single precision real
$double	double precision real
$uchar	unsigned character
$short	16-bit integer

Table 4-4. Data Types

You can also use the built-in data types for type coercion. For example, use them to make a variable a type that isn't supported in the language you're using.

Constant	Description
false	zero
true	nonzero
nil	zero
0x*number*	hexadecimal
0t*number*	decimal
number	decimal
number.[number] [eIE][+I-] [*exp*]	float

Table 4-5. Input Constants

When using data types and constants, remember:

- Overflow on non-float uses the right-most digits.
- Overflow on float uses the left-most of the mantissa and the highest or lowest exponent possible.
- Setting the *$octin dbx* variable changes the default input type to octal.
- Setting the *$hexin* variable changes the default input type to hexadecimal. If both variables are set, *$hexin* takes precedence over *$octin*. See "Predefined *dbx* Variables" in Chapter 6.
- Setting the *$octints dbx* variable changes the default output type to octal.
- Setting the *$hexints* variable changes the default output type to hexadecimal. If both variables are set, *$hexints* takes precedence over *$octints* (see Chapter 6).

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.

In interactive mode, the pound sign is not a comment character (you can't type a comment interactively). Instead, it identifies an expression as a line number. In interactive use, when *dbx* encounters a pound sign, it interprets the characters between the pound sign and the end of the current line as a line number, and it calculates the address of that line number.

To indicate a line number in a script, use two pound signs (##). For example, suppose you want to tell *dbx* to print the address of line 27 of the current file. In a script, type:

```
print ##27
```

In interactive session, type:

```
print #27
```

4.5 Registers

Table 4-6 contains a list of *dbx* hardware registers with brief descriptions. Table 4-7 shows alternate names (aliases) for some of the registers. These are useful for debugging machine code.

The *$regstyle* variable controls which name is shown by *dbx* (for those registers that have alternate names).

Alias	Description
$pc	current user pc
$sp	current value of stack pointer
$rn	register n
$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
$f0, $f2..$f30	single precision floating point registers

Table 4-6. Hardware Registers

Alias	Alternate	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
$r8..$r15	$t0..t7	temporaries
$r16-$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

Table 4-7. Hardware Registers with Aliases

4.6 Keywords

A list of *dbx* keywords appears below. When naming variables in your program, it's best not to use these keywords. If one of the variables is identical to a *dbx* keyword, there will be some minor problems.

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

Here are some additional keywords that are used as C casts:

```
signed          struct
unsigned        union
short           enum
long            double
int             float
char
```

You can turn off these C cast keywords by typing:

```
set $ctypenames=0
```

This prevents these keywords from getting in the way of non-C programs. For more information about C casts, see the following section, "C Type-Casts."

To print a variable name that is also a *dbx* keyword, put the variable in parentheses when you type it. For example, suppose you have a variable called *and*, and you type this:

```
print and
```

You will get a `syntax error` message. Instead, you should type:

```
print (and)
```

which tells *dbx* to print the value of the variable *and.*

4.7 C Type-Casts

Most C casts now work properly. The exceptions are casting values to or from *float* or *double.*

To turn off the C cast keywords see the previous section, "Keywords."

4.8 Pointers and Structures

You can use `->`, `^`, and `.` almost interchangeably for structure data references and pointer dereferences. In the future, this may be changed to reflect language usage precisely, so that `x^.b` and `x.b` are different (similarly for C). It is therefore a good idea to write scripts with pointers and structure references as much like your language as *dbx* allows. That way, future changes won't break your scripts.

4.9 Case Sensitivity in Variable Names

When *dbx* searches its tables for variable names, it matches the name you type against names in its tables according to the *$casesense* variable.

If *$casesense* is 0, case is ignored. If *$casesense* is 1, case is always checked.

If *$casesense* is 2, (the default), then the language in which the variable was defined is taken into account (e.g., C and C++ are case sensitive while Pascal and FORTRAN are not).

For example, to make *dbx* always distinguish between upper and lower case (e.g., A is not the same as a):

```
set $casesense=1
```

4.10 C++ (2.0) Function Names

To make debugging of C++ 2.0 programs easy, use the following special syntax for C++ names. This syntax does not work with C++ 1.2.

Refer to C++ functions with their source name. For example, with class A:

```
A::func
```

For globals:

```
::func
```

For special functions:

`A::new`	
`A::delete`	
`A::class`	(constructor)
`A::~class`	(destructor)
`A::+`	(example of overloaded operator)

5. The *dbx* Monitor

This chapter explains:

- the history feature and the history editor
- how to use *dbx* command line editing
- how to type multiple commands

Examples:

```
history

hed 1,9

!-3

!!

hed all
```

Syntax	Select this command to:
`history`	Print the items in the history list.
`!` [*string*]	Repeat the most recent command that starts with the specified *string*.
`!` [*integer*]	Repeat the command associated with the specified *integer*.
`!` [`-`*integer*]	Repeat the command that occurred *integer* times before the most recent command.
`!!`	Repeat the last command.
[*command*]`;` [*command*]	Type multiple commands on the same line.
`hed`	Edit only the last line of *history*.
`hed` *num1*`,`*num2*	Edit the range of line numbers from *num1* to *num2*.
`hed all`	Edit the entire current *history*.
`set $repeatmode=1`	Set *dbx* so that a carriage return on an empty line works like a double exclamation point (`!!`).

Table 5-1. History Commands

5.1 History Commands (*history* and *!*)

The *dbx history* feature is similar to the C shell's *history* feature. However, *dbx history* works only at the beginning of the line. Don't use *history* in the middle of the line or with *!$*.

Set the number of lines of history by using the *$lines* variable. The default is 20. To reset this variable, use the *set* command (see "Setting *dbx* Variables (*set*)" in Chapter 6). For example:

```
set $lines=200
```

To see a list of the commands in your history list, type:

```
history
```

To repeat a previous command, use one of the exclamation point (*!*) commands. For example:

```
!pr
```

executes the last command that began with the letters pr.

5.2 The History Editor (*hed*)

The history editor, *hed* is available with *dbx* (not with *edge*). The *hed* editor lets you use your favorite editor on any or all of the commands in the current *dbx* history.

When you use the *hed* command, *dbx* puts you in a temporary file, which you can edit. When you quit the editor, any commands left in this temporary file are automatically executed.

hed uses the editor named in the *$editor* variable. The command:

```
set $pimode=1
```

causes the commands to be displayed as they are executed.

Here are some examples. To edit the last line of history, type:

```
hed
```

To edit lines 3 through 9, type:

```
hed 3,9
```

To edit the entire current history, type:

```
hed all
```

5.3 Multiple Commands

Use the semicolon (;) as a separator to type multiple commands on the same command line. This can be useful when you use the *when* command. See "Writing Conditional Code" in Chapter 9.

Example:

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

6. Controlling *dbx*

This chapter describes *dbx* predefined variables and aliases and also explains how to control *dbx* by:

- creating and changing *dbx* variables
- creating and removing command aliases
- recording and playing back input and output
- invoking a shell from *dbx*
- checking and deleting *dbx* status items

Examples:

```
set $repeatmode=1

set $lines=200

alias fv "printf \"%20.3f,\\n\" "

status

delete 3

sh cat small.C

set $rimode=1

record

record output my_journal

unrecord all
```

Syntax	Select this command to...
alias	List all existing aliases.
alias *name*	List the alias string for *name*. The alias value is inserted in quotes with escape characters to show how the alias "*string*" was typed. See the help file (*/usr/lib/dbx.help*) EXPRESSIONS section for additional information.
alias *name* "*string*"	Define a new alias. See the help file (*/usr/lib/dbx.help*) EXPRESSIONS section for additional information.
alias *name name2*	Define a new alias.
alias *name*(*arg1*,...*argN*) "*string*"	Define a new alias. When using the alias, the actual arguments are substituted for "string".
delete *item1...itemN*	Deletes the specified items.
delete all	Delete all the status items.
playback input [*file*]	Execute the commands from the specified file, *file*. The default file is the current temporary file created for the *record input* command.
playback output [*file*]	Print the commands from the specified file, *file*. The default file is the current temporary file created for the *record output* command.
record	Show a list of the current recording sessions. Each is assigned a number.

Table 6-1. Commands to Control *dbx*

Syntax	Select this command to...
record input [*file*]	Record everything you type to *dbx* in the specified file, *file*. The default file is a temporary *dbx** file in the */tmp* directory.
record output [*file*]	Record all *dbx* output in the specified file, *file*. The default file is a temporary *dbx* file in the */tmp* directory.
set	Display a list of predefined and user defined variables.
set *var* = *exp*	Define (or redefines) the specified variable, *var*.
sh [*com*]	Call a shell. Execute the specified shell command, *com*.
status	Check the status of commands.
source [*file*]	Execute *dbx* command from *file*.
unalias *alias*	Remove the specified alias.
unrecord [*N*]	Turn off recording session *N* and close the file involved.
unrecord all	Turn off all recording sessions and close all files involved.
unset *var*	Unset the value of the specified variable, *var* (it disappears from the list).

Table 6-1. (continued) Commands to Control *dbx*

6.1 Predefined *dbx* Variables

Predefined *dbx* variables are listed in the table that follows. The predefined variable names begin with "$" so they don't conflict with variable, command, or alias names.

Variable	Default	Select this variable to...
`$addrfmt`	"0x%x"	C format for address printing ($pc, $pc of source line, $pc offset of instruction in procedure).
`$casesense`	1	If 1(0), case (in)sensitive; if 2, depends on the language.
`$charisunsigned`	1	If 1, a (char) cast is taken as unsigned;if 0, a (char) cast is taken as signed.
`$ctypenames`	1	If 1, words *unsigned, short, long, int, char, struct, union, enum* are keywords usable only in type-casts. If 0, *struct, union, enum* are ordinary words with no predefined meaning (in C modules, the others are still known as C types).
`$curevent`		Current event id for *trace/stop/when*.
`$curline`		The current line number for execution.
`$curpc`		Current program counter.
`$cursrcline`		Current source listing line.

Table 6-2. Predefined Variables

Variable	Default	Select this variable to...
`$datacache`	1	If 0, then cache multiple data accesses while stopped. Setting to 0 is a good idea. No visible loss in performance.
`$defaultin`		Default record input file name if none specified in playback input or record input.
`$defaultout`		Default record output file name if none specified in playback output or record output.
`$editor`		The name of the editor to invoke (with *edit* command). Default is EDITOR environment variable. If EDITOR missing, defaults to *vi*.
`$funcentrybylines`	0	Only applies to Pascal, C, and C++. If 0, *dbx* uses disassembly of the code to estimate the location of the first line of each function. If 1, *dbx* uses the line numbers in the line table; this does not work well if the first line of code is on the same line as the function opening brace "{" as it often is in short C++ functions.

Table 6-2. (continued) Predefined Variables

Variable	Default	Select this variable to...
`$groupforktoo`	0	If 0, only sproc'd procs are added to grouplist automatically. If 1, then fork'd & sproc'd procs added to grouplist.
`$hexchars`	0	If 1, output chars in hex, using C format "%x".
`$hexin`	0	If 1, input constants are assumed in hex. This overrides *$octin*.
`$hexints`	0	If 1, output integers in hex and override *$octints*.
`$hexstrings`	0	If 1, output strings and arrays in hex. For char arrays, if 1, the null byte is not taken as a terminator. Instead, the whole array (or *$maxlen* values, whichever is less) is printed. If 0, then a null byte in an array is taken as the end of the array (the length of the array and *$maxstrlen* can terminate the array print before a null byte is found).

Table 6-2. (continued) Predefined Variables

Variable	Default	Select this variable to...
`$hide_anonymous_blocks`	1	Anonymous inner blocks ({ } in C) are not shown in stack traces or counted in *up/down* commands. If 0, these blocks are shown, counted. Set to 0 if you have a local variable in an inner block hiding a variable with the same name in an outer block and want to see the value of the outer variable. If 0, the *up* command will take you to the outer scope where a *print* command will show the variable visible in that local scope.
`$lastchild`		The process id of the last child *forked/sproced.*
`$lines`	20	Number of lines in history list.
`$listwindow`		List command size.

Table 6-2. (continued) Predefined Variables

Variable	Default	Select this variable to...
`$main`	main	At startup, *dbx* sets source file and line to the function named in this string variable. It can be any procedure. It is only usefully set by *dbx* when it reads the process' symbol table, since it is used only once by *dbx*, before any commands are read.
`$maxstrlen`	128	Maximum length printed for zero-terminated char strings and arrays. Char arrays are printed for array-length, *$maxstrlen* bytes, or up to a null byte, whichever comes first (see *$hexstrings*).
`$mp_program`	0	If 0, *sproc* is treated like *fork*. If 1, *sproc* is treated specially. The children are allowed to run (they will block on multi-processor synchronization code emitted by mp FORTRAN). Set to 1 only if mp FORTRAN code. Set to 1, mp FORTRAN code is easier to work with.
`$naptime`		Trace (no operands) and *step n*, *next n* will delay *$naptime* 100ths of a second after every instruction. See *sginap*(2).

Table 6-2. (continued) Predefined Variables

Variable	Default	Select this variable to...
`$nextbreak`	2	If 0, *$stepintoall* controls whether *next* will behave as if *$nextbreak* were 1 or 2. If 1, *next* will single-step through calls and will get back to the next statement. (Currently, *stop if*, *trace*, or *when* commands testing variables may fail with "not active," stopping execution.) This is slow but guarantees arrival back to the right place even if part of the code is compiled without symbols and is recursive. Also causes tracing of function values to be done in lower level functions (slow), but handy if a lower-level function uses a wild pointer or if data is passed by reference or is visible in upper levels, as in Pascal. If 2, a *next* will quickly execute calls with tracing and breakpointing. If the function calls itself, the *next* may stop at a later return from the function, not precisely the point of call (the stack trace will be deeper than expected).

Table 6-2. (continued) Predefined Variables

Variable	Default	Select this variable to...
`$octin`	0	If 1, input constants are assumed in octal (*$hexin* overrides *$octin*).
`$octints`		If 1, output integers in octal (*$hexints=1* takes precedence).
`$page`	1	If 0, then page screen output.
`$pager`		The name of the program used to display help information.
`$pagewindow`		The size (length in lines) of page for paging. This controls when the "n?" or "More" prompt is displayed.
`$pagewidth`		Width of window in characters (assumes fixed-width font). Used by dbx to calculate how many screen lines are output. *dbx* never inserts newlines; the window software wraps the lines.
`$pid`		Set current process for kernel debugging (-k).
`$pid0`		The process id of the given process.
`$pimode`	0	If 0, *playback input* prints commands as they are played back.

Table 6-2. (continued) Predefined Variables

Variable	Default	Select this variable to...
`$printdata`	0	If 0, print register contents when disassembling.
`$print_exception_frame`		Only available to *dbx -k* (not available unless you are doing kernel debugging). If 0, no special output is done. If 0, when doing *where*, *up*, or *down*, exception frames and struct sigcontext blocks are output in hex when the stack frame encounters one of these on the stack. This may be of some use.
`$printwhilestep`	0	If 0, acts as if *step[i] n* was *step[i] n* times. With *edge*, green line steps through as program executes. If 0, *step n* steps *n* then displays (notifies *edge*).
`$printwide`	0	If 0, print compactly (wide). If 0, print arrays one element per line.
`$prompt`	(dbx)	Prompt string.

Table 6-2. (continued) Predefined Variables

Variable	Default	Select this variable to...
`$promptonfork`	0	If 1, prompt user to add child process to pool on fork. The reply is taken from the current input file (which may be the screen). If 0, child not added to pool. If 0, the running process does not stop on a *fork* or *sproc*. If 2, child added to pool. An *exec* always stops the running process.
`$regstyle`	1	If 0, use hardware names for registers when disassembling.
`$repeatmode`	0	If 1, null line repeats last command.
`$showbreakaddrs`	0	If 1, 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.

Table 6-2. (continued) Predefined Variables

Variable	Default	Select this variable to...
`$stepintoall`	0	If 0, *step[i]* will step into all procedures compiled -g or -g2 or -g3 for which line numbers are available in the symbol table. This does not include standard library routines since they are not compiled -g[23]. If 1, *step[i]* will step into those plus procedures for which dbx can find a source file. If 2, *step[i]* will step into all procedures.
`$tagfile`	"tags"	String with name of file searched for tags. Defaults to *tags*. See *ctags*(1).
`$visiblemangled`	0	Applicable to C++ 2.0 only. If not 0, mangled names of functions print along with demangled form.

Table 6-2. (continued) Predefined Variables

6.2 Setting *dbx* Variables (*set*)

The *set* command defines a *dbx* variable, sets an existing *dbx* variable to a different type, or displays a list of existing *dbx* defined variables. You cannot define a debugger variable that has the same name as a program variable. You can see the setting for a single variable by using the *print* command. (Recall that predefined variable names begin with "$" so they don't conflict with variable, command, or alias names.)

For example:

```
set
set $promptonfork = 2
set $prompt = "::>"
set set $myvar = (int) $myvar+1
```

The *dbx* predefined variables are listed in the table at the beginning of this chapter.

6.3 Removing Variables (*unset*)

Use the *unset* command to remove the specified *dbx* variable from the list. To see a full list of *dbx* variables, use the *set* command, or see the beginning of this chapter.

For example:

```
unset $myvar
```

6.4 Creating Command Aliases (*alias*)

Use the *alias* command to see a list of all current aliases or to define a new alias.

dbx lets you create an alias for any debugger command. Enclose multi-word command names within double or single quotation marks.

dbx has a group of predefined aliases that you can modify or delete. In addition, you can add your own aliases. To do so, use the following syntax:

alias *name(arg1,...argN) "string"*

When using the alias, the actual arguments are substituted in "string." For example:

```
alias m(a,b) "print alf.a.b"
m(mystruct,i)
#the command is: print alf.mystruct.i
```

You can include aliases in the *.dbxinit* file if you want to use them in future debugging sections.

For a complete list of predefined aliases, see the section titled "Predefined Aliases" that follows, or type to *dbx*:

```
alias
```

Also see the section "Alias Examples."

6.5 Removing Command Aliases (*unalias*)

The *unalias* command removes the specified alias from the current debugger session. All predefined aliases are restored the next time you start a debugging session. For example:

```
unalias pd
```

6.6 Predefined Aliases

To list current aliases, use the *alias* command. You can override any predefined alias by redefining it with the *alias* command or by removing it from the list with the *unalias* command. The following table shows the debugger predefined aliases. For example:

```
alias
```

Alias	Command	Select this alias to...
a	assign	Assign the specified expression to the specified program variable.
b	stop at	Set a breakpoint at the specified line.
bp	stop in	Stop in the specified procedure.
c	cont	Continue program execution after a breakpoint.
d	delete	Delete the specified item from the status list.
e	file	Display 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	Move to the specified procedure (activation level) on the stack. If no procedure or expression is specified, *dbx* prints the current activation level.
g	goto	Go to the specified source line.
h	history	List all the items currently in the history list.

Table 6-3. Predefined Aliases

Alias	Command	Select this alias to...
j	status	List all the currently set stop and trace commands.
li	$curp/10i; set $curp=$curp+40	List the next 40 bytes of machine instructions (approximately 10 instructions).
n	next	Execute the next *n* lines of source code. The default value is one line. This command does not step down into procedures.
ni	nexti	Execute the next *n* lines of machine code. The default value is one line. This command does not step down into procedures.
p	print	Print the value of the specified variable or expression.
pd	printf "%d\n",	Print the value of the specified variable in decimal.
pi	playback input	Replay *dbx* commands saved in the specified file. If no file is specified, *dbx* uses the temporary file specified by *$default*.

Table 6-3. (continued) Predefined Aliases

Alias	Command	Select this alias to...
po	printf "0%o\n",	Print the value of the specified variable or expression in octal.
pr	printregs	Print values contained in all registers.
px	printf "0x%x\n",	Print the value of the specified variable or expression in hexadecimal.
q	quit	Exit *dbx*.
r	rerun	Run the program again using the values specified in the last used run command.
ri	record input	Record to the specified file all the commands you give to *dbx*. If you do not specify a file, *dbx* creates a temporary file. The name of the file is specified by *$default*.
ro	record output	Record all the debugger output to the specified file. If no file is specified, output is recorded to a temporary file. The name of the file is specified by *$default*.

Table 6-3. (continued) Predefined Aliases

Alias	Command	Select this alias to...
s	step	Execute the next *n* number of lines. If a line contains a procedure, this command steps down into that procedure. The default is one line.
S	next	Execute the next *n* number of lines. If a line contains a procedure, this command does not step down into that procedure. The default is one line.
si	stepi	Execute the next *n* lines of assembly code. The default is one line. If a line contains a procedure call, this command steps down into the procedure.
Si	nexti	Execute the next *n* lines of assembly code. The default is one line. If a line contains a procedure call, this command does not step down into the procedure.

Table 6-3. (continued) Predefined Aliases

Alias	Command	Select this alias to...
source	playback input	Replay *dbx* commands saved in the specified file. If no file is specified, *dbx* uses the temporary file specified by *$default*.
t	where	Do a stack trace to show the current activation levels.
w	list $curline-4:10	List a window of 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-9:20	List a window of 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	List a window of assembly code around the program counter.

Table 6-3. (continued) Predefined Aliases

6.7 Alias Examples

The following examples show various ways that you can use the *alias* command with the debugger. You must use escapes when using aliases with internal quotes.

```
# printing a float/double with your own choice of size
alias mypflt(v)  "printf \"%6.18f\\n\",v"
mypflt(44.551234567)     # example using the alias

alias pf2  "printf \"%27.3f\\n\","
pf2  32.3
```

Some quoted strings (" ") in the following examples are split into two lines so they can be printed here. However, when you type strings (or any command), the entire string/command must appear on one line for *dbx* to recognize it.

An easy way to follow linked lists is to use aliases and casts. The following example shows a linked list with *next* pointers and a pointer to the contents of the linked list:

```
  struct list { struct list *next; int *elt ; } *mylist;

set $pimode=1
# set up aliases for following the list:
alias foll(p) "px ((struct list *) p)->next ;
   px ((struct list *)p)->elt"
alias show(t,p) "print *(t *) (p)"
# then, an initial list element is pointed to by mylist:
# use the following to print the first element and its
# contents where I assume one knows the contents somehow
foll(mylist)
# use the "elt" address printed (assume it is 0x123)
# and assume the element is a pointer to "struct something":
show(struct something,0x123)
# using the "next" address printed (assume it was 0x345)
foll(0x345)
# which shows the next list structure.
# And so on.
```

The following example has a similar scheme. The alias remembers the last pointer.

```
# Assume we know the address of an element of
# the list is at 0x1234
set $p = 0x1234
# Aliases, including "folly" below, must be defined on
# one line; what looks like two lines is just one line
# wrapped around. The cast to int in the "set $p = " is
# essential! If left off,dbx will leave the $p reference
# symbolic and dbx will get into an infinite
# loop (use ^C or your interrupt key) to get out of trouble
# if you get into the infinite loop).
# Of course $p is an arbitrary choice-use any name desired.
# The name is not required to start with $, though starting
# with $ is a good idea as the name won't conflict
# with program variable names.
alias folly "print *(struct list *)$p ;  set $p =
                  (int)((struct list *)($p))->next"
#
# then each time you type "folly," you see the next
# element printed
folly
folly
set $repeatmode=1
# now after the initial "folly," simply pressing return
# will print the next list entry
folly
```

6.8 Showing Record State (*record*)

Use *record* to show *record input* or *record output* session currently active. For example:

```
record
```

6.9 Recording Input (*record 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. You can start and run as many simultaneous *dbx* input recording sessions as you need.

After you end the input recording session, use the command file with the *source* or *playback* input commands to execute again all the commands saved to the file. See "Playing Back Input" later in this chapter. The syntax of *record input* is shown below. *dbx* saves the recorded input in the specified file.
For example, to save the recorded input in a file called *keving*, type:

```
record input keving
```

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 in the system variable *$defaultin*. You can display the temporary file name using the print command as follows:

```
print $defaultin
```

Because the */tmp/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 file name when you invoke *record input*. If the specified file already exists, the new input is appended to the file.

6.10 Ending a Record Session (*unrecord*)

To end a *dbx* recording session, use the *unrecord all* or the *unrecord [session #]* command. Thus, to stop recording session 3, you would enter the *dbx* command:

```
unrecord 3
```

To stop all recording sessions, use:

```
unrecord all
```

Note: The *dbx status* command does not report on recording sessions. To see whether there are any active recording sessions, use the *record* command. For example:

```
record
```

6.11 Recording Output (*record 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. By default, the commands you enter are not copied to the output file. However, if you set *$rimode* to a non-zero value, *dbx* will also copy the commands you enter.

Examples:

```
set $rimode=1
record output  my_output
```

The *record output* command is very useful when the screen output is too large for a single screen (e.g., printing a very large structure). Within *dbx*, you can use the *playback output* command 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*).

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

```
record output gaffa
```

To record both the commands and the output, type:

```
set $rimode=1
record output gaffa
```

If you omit the file name, *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 after the *dbx* session, you must specify the file name when giving the *record output* command. The name of the temporary file is in the system variable *$defaultout*. To display the temporary file name, type:

```
print $defaultin
```

To end a record output session, use the *unrecord all* or specify a session number. For example, to end recording session 3, you would type:

```
unrecord 3
```

Note: The *dbx* status command does not report on recording sessions. To see whether there are any active recording sessions, use the *record* command without arguments.

6.12 Playing Back Input (*source* or *playback input*)

Use these commands to replay the commands that you recorded with the *record input* command. If you don't specify a file name, *dbx* uses the current temporary file that it created for the record input command. If you set the *dbx* variable *$pimode* to non-zero, the commands are printed out as they are played back. By default, *$pimode* is set to zero.

Examples:

```
playback input script_file
pi script2
```

6.13 Playing Back Output (*playback output*)

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

Example:

```
playback output ascript
```

6.14 Invoking a Shell (*sh*)

To invoke a sub-shell, type **sh** at the *dbx* prompt, or type **sh** and a shell command at the *dbx* prompt. If you invoke a sub-shell, type **exit** or press **<ctrl-d>** to return to *dbx*. For example:

```
sh tail mydata
sh
```

6.15 Checking the Status (*status*)

Use the *status* command to check which, if any, of these commands are currently set:

- *stop* or *stopi* commands for breakpoints
- *trace* or *tracei* commands for line-by-line variable tracing
- *when* command

For example:

```
status
```

6.16 Deleting Status Items (*delete*)

Use the *delete* command to remove items (e.g., breakpoints, conditionals) from the status list. For example, to remove all items from the status list, type:

```
delete all
```

To remove item number 3 from the status list, type:

```
delete 3
```

7. Examining Source Programs

This chapter explains how to:

- specify source directories
- move to a procedure
- change source files
- list source code
- search for strings in source code
- call an editor from *dbx*
- print symbol names
- print type declarations for variables

Examples:

```
dir  x/b

func  somefunc

list  1,30

/abcd

edit  "special.c"

whatis  x

whereis  y
```

Syntax	Select this command to...
/exp	Search ahead in the current source file for the specified regular expression, *exp*.
?*exp*	Search back in the current source file for the specified regular expression, *exp*.
dir	List current directory.
dir *dir1...dirn*	Add *dir1...dirn* to the current list of directories.
edit [*file*] [*function*]	Call an editor to edit the specified source file or function. Default is the current file or function.
file [*file*]	Change the current source file to specified file, *file*. The default is the current file.
func [*proc*] [*exp*]	Move to the activation level specified by the procedure or expression. Default is the current activation level(s).
list [*exp*] [*proc*]	List the specified line(s) for *$listwindow*. Default is the current line.
list *exp*:*integer*	Lists the specified number of lines, *integer*, starting at the specified line, *exp*.
use	List the current directories.
use *dir1...dirN*	Specifies different directories, *dir1*, *dir2*, etc.

Table 7-1. Commands to Examine Source Programs

Syntax	Select this command to...
whatis *variable*	Print the type declaration for the specified variable or procedure in your program.
whereis *variable*	Print all versions of the specified variable.
which *variable*	Print the currently active version of the specified variable.

Table 7-1. (continued) Commands to Examine Source Programs

7.1 Specifying Source Directories (*use* and *dir*)

Unless you specify the **-I** option at invocation, *dbx* looks for source files in the current directory or in the object file's directory. The *use* command lets you change the directory list and list the directories currently in use. The command recognizes absolute and relative pathnames (for example, *./*); however, it doesn't recognize the C shell tilde (~) syntax (e.g., ~ *john/src*) or environment variables (e.g., $HOME/src).

For example:

```
use dir1
```

makes *dir1* the only directory. The example:

```
dir dir1
```

adds *dir1* to the list of directories.

To print the current directory list, you can type either:

```
use
```

or

```
dir
```

7.2 Moving to a Specified Procedure (*func*)

The *func* command moves you up or down the activation stack. The function can be a procedure name or an activation level number. To find the name or activation number for a specific procedure, do a stack trace with the *where* command. You can also move through the activation stack by using the *up* and *down* commands. For a definition of activation levels, see Chapter 2, "Getting Started."

The *func* command changes the current line, the current file, and the current procedure if the named function is currently on the activation stack.

This changes the scope of the variables you can access. You can use the *func* command when a program isn't executing and the function is not on the activation stack (when you want only to examine source code).

For example:

```
func x
```

changes the source view to function *x*. If function *x* is currently on the activation stack, the command (*func x*) also changes the scope of the variables you can access to those variables visible in *x*.

7.3 Specifying Source Files (*file*)

The *file* command changes the current source file to a file you specify. The new file becomes the current file, which you can search, list, and perform other operations on. For example:

```
file whizzy.c
```

7.4 Listing Your Source Code (*list*)

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 current file, procedure, and line unless otherwise specified. It moves the current line forward.

7.5 Searching Through Source Code (/ and ?)

The / and ? commands search through the current file for regular expressions in source code. The slash (/) searches forward; the question mark (?) searches back from the current line. Both commands search the entire file. They wrap around to the beginning of the file and end at the point where you invoked the search command.

To search forward in the code for a specified regular expression, type:

/exp

For example:

```
/a.*xy
```

To search backward in the code for a specified regular expression, type:

?exp

7.6 Calling an Editor (*edit*)

The *edit* command lets you make changes to your source code from within *dbx*. For example, to edit a file named soar.c from within *dbx*, type:

```
edit soar.c
```

For the changes to become effective, you must recompile and rerun your program. The *edit* command loads the editor that you set as an environment variable editor. If you don't set the environment variable, *dbx* assumes the *vi* editor. When you exit the editor, it returns you to the *dbx* prompt.

7.7 Printing Symbolic Names (*which* and *whereis*)

The *which* and *whereis* commands print program variables. These commands are useful for programs that have multiple variables with the same name occurring in different scopes. The commands follow the rules described in "Qualifying Variable Names" in Chapter 4.

Examples:

```
whereis a
which b
```

7.8 Printing Type Declarations (*whatis*)

The *whatis* command lists the type declaration for variables and procedures in your program. For example, to list the type declaration for the variable, *x*, type:

```
whatis x
```

For a procedure called quahog, type:

```
whatis quahog
```

Since the same name sometimes applies to source files as well as variable names, the example:

```
whatis carpet
```

may display:

```
source file "carpet.c"
(int) carpet
```

8. Controlling Your Program

This chapter explains how to control a program by:

- running and rerunning the program
- stepping through the program one line at a time
- returning from a procedure call
- starting at a specified line
- continuing after a breakpoint
- assigning values to program variables

Examples:

```
run

run -f data2 <indata >outdata

step

return

cont

assign c = 27
```

Syntax	Select this command to...
assign *exp1* = *exp2*	Assign a new value to a program variable.
cont	Resume execution from the current line and wait for a break or other event.
cont *signal*	Send signal *signal* to the process, resume execution, and wait for a break or other event.
cont to *line*	Set a breakpoint at line, *line*, resume execution, and wait for a break or other event.
cont to *proc*	Set a breakpoint at the first line of procedure *proc*, resume execution, and wait for a break or other event.
cont *signal* **to** *line*	Set a breakpoint at line, *line*, send signal *signal* to the process, resume execution, and wait for a break or other event.
cont *signal* **to** *proc*	Set a breakpoint at the first line of procedure *proc*, send signal *signal* to the process, resume execution, and wait for a break or other event.
goto *line*	Start execution at the specified line, *line*, when execution is resumed.
next [*integer*]	Step over the specified number of lines (default is 1). This command does not step into procedures. Breakpoints in procedures stepped over are honored.

Table 8-1. Commands to Control a Program

Syntax	Select this command to...
rerun [*arg1...argN*] [<|>|>&*file1*]	Rerun your program with the arguments you specified to the *run* command or with new arguments. The <>*file* arguments redirect program input and output. The >&*file* argument redirects *stderr* and *stdout* output to the specified file.
return	Continue execution until control returns to the next procedure up on the activation stack.
return [*proc*]	Continue execution until control returns to the named procedure.
run [*arg1...argN*] [<|>|>&*file1*]	Run your program with the specified arguments. The <>*file* arguments allow you to redirect program input and output. The >&*file* argument redirects *stderr* and *stdout* output to the specified file.
step [*integer*]	Execute the specified number of lines, *integer*, source code. *integer* refers to the number of lines to be executed in the current procedure, as well as any called procedures. Default is 1 source code line.

Table 8-1. (continued) Commands to Control a Program

8.1 Running Your Program (*run* and *rerun*)

The *run* and *rerun* commands start program execution. You can specify arguments to either command. Arguments to these commands override previous arguments. If you don't specify arguments to the *run* or *rerun* command, it uses the last set of arguments.

Use these commands to redirect program input and output (works like redirection in the C shell). The optional `<FILE1` parameter redirects input to your program from the specified file. `<FILE2` redirects output from the program to the specified file. The optional parameter `>&FILE2` redirects *stderr* and *stdout* output to the specified file.

Note: This output differs from the output you save with the *record output* command, which saves debugger (not program) output in a file. See "Recording Output (*record output*)" in Chapter 6.

The arguments to the *run* command specify any program arguments that your program might have. For example, suppose you have a program called *sik.c* compiled as:

```
cc -g sik.c -o sik
```

Then, to run the program you would type:

```
run sik
```

To run the program with the specified arguments, use the following format:

run [*arg1...argN*] [>*file1*] [>*file2*]

Rerun the program, using the same arguments that were specified to the *run* command. If you specify new arguments, *rerun* uses those arguments:

rerun [*arg1...argN*] [<*file1*] [<*file2*]

8.2 Executing Single Lines (*step* and *next*)

The *step* and *next* commands execute a fixed number of source code lines as specified by *exp*. If you don't specify *exp* for *step* and *next*, *dbx* executes one source code line. If you specify *exp*, *dbx* executes the source code lines as follows:

- For *step* and *next*, *dbx* does not take comment lines into consideration in interpreting *exp*. The program executes *exp* source code lines, regardless of the number of comment lines interspersed among them.
- For *step*, *dbx* considers *exp* to apply to both the current procedure and to called procedures. The program stops after executing *exp* source lines in the current procedure and any called procedures.
- For *next*, *dbx* considers *exp* to apply to only the current procedure. The program stops after executing *exp* source lines in the current procedure, regardless of the number of source lines executed in any called procedures.

8.2.1 *step [exp]*

This command steps only into procedures that have line numbers and were compiled with options `-g`, `-g2`, or `-g3`. The *step* command honors breakpoints in any procedures it steps over.

The *step* command counts source lines in called procedures as well as the current procedure. The program stops after executing the specified number of source lines in the current procedure and any called procedures.

You can force *step* to step into all procedures for which *dbx* can find a source file, even if the procedure was not compiled with symbols. To do this, type:

```
set $stepintoall=2
```

When you debug a source file compiled without symbols or compiled with optimization, the line numbers sometimes jump erratically, which may be quite confusing.

If the file for a procedure has the same name as a file *dbx* can find, *dbx* will step into the procedure while reporting the source file it finds. The source

file found may not be the source file the procedure was compiled from; this will also be very confusing.

You can also use *$stepintoall* to control *step* in other ways as shown below.

- If =0 (the default), *step*[*i*] will step into all procedures compiled with **-g**, **-g2**, or **-g3** for which line numbers are available in the symbol table. This does not include standard library routines since they are not compiled **-g**[*23*].
- If =1, *step*[*i*] will step into the above plus procedures for which *dbx* can find a source file.
- If =2, *step*[*i*] will step into all procedures.

You can also use the variable *printwhilestep* to control *step*.

- If =0, *dbx* acts as if the command:

  step[i] n

  was *step*[*i*] *n* times. With *edge*, green line steps through as the program executes.

- If =0, (the default):

  stepn

  steps *n* then displays (notifies *edge*):

  ```
  -trace (no operands) and step, next will delay
  $naptime 100ths of a second after every instruction.
  ```

  For more information, see *sginap*(2).

8.2.2 *next [integer]*

The *next* command steps over the specified number of lines (default is 1). This command does not step into procedures. Breakpoints in procedures stepped over are honored. See the previous description of *$stepintoall*, and the subsequent description of *$nextbreak* for information on modifying the behavior of *next*.

$nextbreak does the following:

- If =0, *$stepintoall* controls whether *next* will behave as if *$nextbreak* were 1 or 2. If 0, *step[i]* will step into all procedures compiled **-g**, **-g2**, or **-g3**, for which line numbers are available in the symbol table. This does not include standard library routines since they are not compiled -g[23]. If 1, *step[i]* will step into those plus procedures for which dbx can find a source file. If 2, *step[i]* will step into all procedures.
- If =1, a *next* command will single-step through calls and will get back to exactly the next statement. (In the current implementation, *stop if* or *trace* or *when* commands testing variables may fail with the message "not active," stopping the execution.) This is slow but guarantees arrival back at the right place even if part of your code is compiled without symbols and is recursive. It also causes tracing of function values to be done in lower-level functions, (which may be slow but handy) if a lower-level function uses a wild pointer or if data is passed by reference or is visible in upper levels, as in Pascal.
- If =2 (the default), a *next* command will execute calls at full speed. If nexting from within a function that recurses, *next* may stop at a deeper level of recursion rather than at the next source. To avoid this, set *$nextbreak=1*.

8.3 Starting at a Specified Line (*goto*)

The *goto* command shifts program execution to a line you specify. This command is useful in a *when* statement, e.g., to skip a line that you know has problems. For example:

```
goto 136
```

Note: The *goto* command cannot be used to go to a line outside the currently active procedure.

8.4 Continuing after a Breakpoint (*cont*)

The *cont* command resumes program execution after a breakpoint. These commands, *cont to* and *cont in*, go into effect when you issue them. When your program again reaches a breakpoint, you can reissue either *cont to* or *cont in* to continue, if desired. If **SIGNAL** is specified as a parameter, *dbx* sends the specified signal to the program and continues.

Note: You can also use the *dbx* command, *resume*, to execute past a breakpoint.

For example:

```
cont
cont int
cont sigint
```

8.5 Variables and Registers

The *assign* command changes the value of existing program variables or registers. For example:

```
assign  x = 27
assign  y = 37.5
```

If the "incompatible types" message appears when you try to assign a value to a pointer, use casts to make the assignment work. For example:

```
# c definition might be struct x *y;
# to set y to the null pointer;
assign *(int *) (&y) = 0
```

9. Setting Breakpoints

Topics covered in this chapter include:

- setting breakpoints at lines
- setting breakpoints in procedures
- running your program after a breakpoint
- stopping for signals
- stopping for system calls

Examples:

```
stop at 37

stop at "compute.f":29

stop in myfunc

cont

cont sigfpe

syscall catch call exit

catch USR1

ignore INT
```

Syntax	Select this command to...
`catch`	List all the signals that *dbx* catches.
`catch` [*signal*]	Add a new signal to the catch list. A *signal* can be a name or number.
`ignore`	List of the signals that *dbx* does not catch.
`ignore` [*signal*]	Add a signal, *signal*, to the ignore list.
`stop` [*var*] `at` *line*	Set a breakpoint at the specified source line. The breakpoint is conditional if the variable, *var*, is specified.
`stop` [*var*] `at` *line* `if` *exp*	Set a conditional breakpoint at the specified source line.
`stop` *proc*	Set up to stop execution when the specified procedure, *proc*, is entered.
`stop` [*var*] `in` *proc*	Set a breakpoint in the specified procedure. The breakpoint is conditional if the variable, *var*, is specified.
`stop at`	Set a breakpoint at the current source line.
`stop` [var] `in` *proc* if *exp*	Set a conditional breakpoint in the specified procedure.

Table 9-1. Commands for Setting Breakpoints

Syntax	Select this command to...
`stop if` *exp*	Set up program execution tracing, test the expression at each program line, and stop when it is true. (Execution will be very slow.)
`stop` *var*	Set up to stop execution when the variable, *var*, changes. (Execution will be very slow.)
`syscall`	Print the list of system calls in 4 sections.

`syscall catch call`
`syscall ignore call`
`syscall catch return`
`syscall ignore return`
	These four commands print individual sections of the 4-section list that *syscall* prints.

`syscall catch call` *syscall...*	Make the named system calls breakpoint at the entry of the system call.
`syscall ignore call` *syscall...*	Make the named system calls not breakpoint at the entry of the system call.
`syscall catch return` *syscall...*	Make the named system calls breakpoint at the return from the system call.
`syscall ignore return` *syscall...*	Make the named system calls not breakpoint at the entry from the system call.

Table 9-1. (continued) Commands for Setting Breakpoints

Syntax	Select this command to...
`syscall catch call all`	Make all system calls breakpoint at the entry to the system call.
`syscall ignore call all`	Make all system calls not breakpoint at the entry to the system call.
`syscall catch return all`	Make all system calls breakpoint at the return from the system call.
`syscall ignore return all`	Make all system calls not breakpoint at the return from the system call.
`trace`	Step through the program a line at a time without stopping.
`trace` *var*	When the variable, *var*, changes, print its old and new values. (Execution will be very slow.)
`trace` *proc*	When the procedure, *proc*, is entered, print its arguments and its caller's name.
`trace` *var* `in` *proc*	Print the variable, *var*, when it changes in procedure, *proc*.
`trace` *var* `at` *line*	Print the variable, *var*'s, old and new values when the source line is reached.
`trace` *var* `at` *line* `if` *exp*	If the expression is true when the source line is reached, and the variable, *var*, has changed value, print the old and new values.

Table 9-1. (continued) Commands for Setting Breakpoints

Syntax	Select this command to...
trace *var* **in** *proc* **if** *exp*	Print the variable, *var*, when it changes in the procedure *proc*, if the expression, *exp*, is true.
when if *exp* {*command-list*}	Execute *command-list* on every line executed for which the expression, *exp*, is true. (Execution will be very slow.)
when at *line* [**if** *exp*] {*command-list*}	Execute the specified *command-list* when the conditions are met.
when *var* [**at** *line*] [**if** *expr*] {*command-list*}	Execute the specified *command-list* when the conditions are met. (Execution will be very slow unless **at** *line* is specified.)
when *var* [**in** *proc*] [**if** *expr*] {*command-list*}	Execute the specified *command-list* when the conditions are met. (Execution will be very slow.)
when in *proc* [**if** *exp*] {*command-list*}	Execute the specified *command-list* when the conditions are met.

Table 9-1. (continued) Commands for Setting Breakpoints

9.1 Introduction

When a program stops at a breakpoint, the debugger displays an informational message. However, before setting a breakpoint in a program that has multiple files, be sure that you're setting the breakpoint in the right file.

To select the right procedure, follow these steps:

1. Use the *func* command and specify a procedure name. This command moves you to the file that contains the specified procedure (see Chapter 8, "Controlling Your Program").
2. List the lines of the procedure. Use the *list* command (see Chapter 8).
3. When you see the procedure or line you want, use a *stop* command to set a breakpoint.

You can use the *dbx* variable *$showbreakaddrs* to verify exact breakpoint placement. For example:

```
set $showbreakaddrs=1
```

shows 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.

This chapter describes the *stop*, *trace*, and *when* commands, which set up breakpoints and tracing, but do not begin or continue program execution. Each command takes several optional arguments (clauses); two that appear repeatedly are the *variable* and **`if`** *expression* clauses. They are explained in the following paragraphs.

9.1.1 The *variable* Clause

The *variable* clause turns the command into a conditional command. The condition is "has the variable value changed?". When *variable* is used, it may be either a variable or an expression.

If a variable is given, that variable is inspected at "appropriate" points. If an expression is given, that expression is assumed to be a pointer to a 32-bit value and the value-pointed-at is inspected at the appropriate points. An appropriate point is either:

- at specific locations in the program (if **`at`** *line*)
- at every instruction in a given function (if **`in`** *procedure*)
- at every instruction (execution will be very slow).

If *variable* has changed, then the result of the condition test "has the variable's value changed?" is true. Otherwise, the result of the test is false. The old and the new values are printed.

9.1.2 The *if expression* Clause

The **`if`** *expression* clause turns the command into a conditional command. The *expression* is evaluated at the same points as mentioned for *variable* and evaluates to true and false.

9.1.3 Combining the *variable* and *if expression* Clauses

If you use both *variable* and **`if`** *expression*, the overall test evaluates to true only if both evaluate true.

The *stop* command, for example, stops execution of the process when the **`if`** clause is true (if present) and the *variable* has changed (if present).

9.2 *stop (breakpointing)*

The syntax for the *stop* command is at the beginning of this chapter. Examples include:

```
stop in funcf
stop at "file.c":27
stop at 38
b 38
stop in funcz if x==0
```

Interactive function calls provide a powerful way to set breakpoints.

For example, function foo has a C string argument *s* and an integer argument *i*. Then to stop when *s* is "abc" and *i* is 24, issue the command:

```
stop in foo if i == 24 && strcmp("abc",s) == 0
```

This only works if the function *strcmp*(3C) is linked into the program you are debugging.

9.3 Tracing (*trace*)

The *trace* commands print information about the process when the trace conditions are satisfied, but program execution continues. The syntax for tracing is given at the beginning of this chapter.

The *trace* command steps through the program a line at a time without stopping. This command is useful with *edge*(1), as *edge* shows the source code with a green bar on the line executing as *dbx* steps through it.

Examples include:

```
trace  funcf
trace  z
```

9.4 Writing Conditional Code (*when*)

The *when* command is similar to *stop* except that rather than stopping when the conditions are met, the *command-list* (*dbx* commands separated by semi-colons) is executed. If one of the commands in the list is *stop* (with no operands), then the process will stop when the *command-list* is executed.

Examples are:

```
when in funcx  { trace z }
when at "file.c":27  if z=2  {print y}
```

9.5 Stopping at Signals (*catch* and *ignore*)

The catch command lists the signals that *dbx* catches or specifies a signal for *dbx* to catch. If a child in the program encounters a specified signal, *dbx* stops the process and gives you control.

You can use signal names and numbers as listed on the *signal*(2) man page. You can abbreviate signal names by omitting the "SIG" portion. *dbx* ignores case on the signal names.

Note: *dbx* ignores SIGSTOP, SIGTSTP, SIGCONT, SIGTTIN, and SIGTTOU in this version since handling them would lead to problems in *dbx* as presently coded (they are used for ^Z handling in *csh*). Using *dbx* to debug a program (such as a shell) that manipulates the above signals will not work well.

The syntax:

catch [*signal*]

adds a new signal to the catch list. A *signal* is specified as a name or a number. For example, the interrupt signal is named INT, SIGINT, or 2. A process does not see this signal directed at it until 1) the signal comes to *dbx* and the process is stopped, and 2) the process is continued. If the process has not declared a signal handler for a signal, the process does not see the signal when it is continued.

The syntax:

ignore [*signal*]

adds a signal, *signal*, to the ignore list. A *signal* is specified as a name or a number. For example, the interrupt signal is named INT, SIGINT, or 2. A process sees this signal when directed at it by itself or by another process. The process responds to the signal just as if *dbx* were not present. A SIGINT signal at the keyboard is seen by *dbx* and it interrupts *dbx* (it is also sent to the process(s) being debugged). Keyboard-generated signals are seen by the whole process group (i.e., the IRIX process group, not the *dbx* process group).

Debugging a program that attempts to catch signals can be awkward if you catch the signal in *dbx*. For example, if program *P* wants to catch SIGFPEs and you issue the command:

```
catch sigfpe
```

then you must, after *dbx* sees the signal *ignore sigfpe* allow program *P* to see the signal when you issue the command:

```
cont sigfpe
```

Having ignored the signal, you have to get control in *dbx* again (the best way would be to set breakpoints *before* doing the *cont*) to re-do the *catch sigfpe* if you wish to catch floating-point exceptions.

Examples are:

```
catch 2
ignore INT
```

9.6 Stopping at System Calls (*syscall*)

The *syscall* command prints the list of system calls. The syntax is listed in the beginning of this chapter.

The list of system calls is printed in four sections. System calls may be caught (breakpointed) at the time the call is made, or when it is about to return. The *syscall* command prints the system calls in the four sections:

1. caught at call
2. ignored at call
3. caught at return
4. ignored at return

The system calls are all listed in */usr/include/sys.s*. In all *syscall* commands, case is ignored when checking system call names. Thus you can use lower case in these commands.

The **`syscall catch call`** *syscall...* command makes the named system calls breakpoint at the entry of the system call. A particularly useful setting is:

```
syscall catch call exit
```

which will breakpoint the entry to *exit()*. Thus, if the program is about to terminate, you can do a stacktrace before the termination to see why *exit()* was called. Examples include:

```
syscall
syscall catch call exit
syscall catch return read
```

10. Examining Program State

This chapter describes how to examine a program's state by:

- printing stack traces
- moving up and down the activation levels of the stack
- printing variable values
- printing register values
- printing information about the stack trace activation levels
- using interactive function calls

Examples:

```
where

up 4

down

printf "8.1f %d\n",fv2,ival

print $pc

dump .

ccall func(y,3)
```

Syntax	Select this command to...
`ccall` *func(arg1,arg2,...,argn)*	Call a function with the given arguments.
`down` [*num*]	Move down the specified number of activation levels in the stack. The default is one level.
`dump`	Print variable information about the current procedure.
`dump` [*proc*]	Print variable information about the procedure, *proc*, which must be active.
`dump .`	Print variable information for all procedures currently active.
`print` [*exp1...expN*]	Print the value of the specified expressions.
`print "`*string*`",` [*exp1*, ..., *expN*]	Print the value of the specified expressions in the format specified by the string, *string*.
`printregs`	Print the current values of all current registers.
`up` [*num*]	Move up the specified number of activation levels in the stack. The default is one level.
`where`	Print a stack trace.

Table 10-1. Commands to Examine a Program's State

10.1 Doing Stack Traces (*where*)

The *where* command prints stack traces. Stack traces show the current activation levels (procedures) of a program. This command doesn't trace variables. For example:

```
where
t
```

The *t* is an alias for *where*.

10.2 Moving In the Stack (*up* and *down*)

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. For a definition of activation levels, see "Activation Levels" in Chapter 2. For example:

```
up
up 2
down 3
down
```

10.3 Printing (*print* and *printf*)

The *print* command lists the value of one or more expressions. You can also use *print* to display the program counter and the current value of registers (see the following section, "Printing Register Values," for details).

The *printf* command lists information in a format you specify and supports all formats of the IRIX *printf* command except *%s*. For a list of formats, see the *printf*(3S) man page in the *IRIX Programmer's Reference Manual*. For example, you can use *printf* in *dbx* when you want to see a variable's value in a different number base.

The command alias list has some useful aliases for printing the value of variables in different bases: octal (*po*), decimal (*pd*), and hexadecimal (*px*). The default number base is decimal. See "Creating Command Aliases" in Chapter 6 for more information. Examples are:

```
print Zp->X[a].bval
printf "%8.1f\n",fval
printf "8.1f %d\n",fv2,ival
```

regs command"

10.4 Printing Register Values (*printregs*)

The *printregs* command prints register values, both the real machine register names and the software (from the include file *regdefs.h*) names. A prefix before the register number specifies the type of register. The prefixes used and their meanings are shown in the following table.

Prefix	Register Type
$r	machine register
$f	floating point
$d	double precision floating point
$pc	program counter value

Table 10-2. Register Prefixes

You can also specify prefixed registers in the print command to display a register value or the program counter. For example, typing:

```
print $r3
print $pc
```

prints the values of machine register 3 and the program counter, respectively. Set the *dbx* variables *$hexints* and *$hexouts* to 1 to specify that the listing uses hexadecimal.

10.5 Printing Activation Level Information (*dump*)

The *dump* command prints information about activation levels. For example, this command prints values for all variables local to a specified activation level. To see what activation levels you have in your program, use the *where* command to do a stack trace. Examples include:

```
dump
dump funcf
dump .
```

10.6 Interactive Function Calls (*ccall*)

The interactive function call, *ccall*, calls a function (with arguments, if given). 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. The *ccall* command is particularly suited to procedures or functions that do not return a value. For example:

```
ccall myfunc(x,2)
```

You can call functions that do return a value as normal expressions. For example, to call the function *f*, which returns an integer (taking integer, double, and string arguments), and shift the results by two bits, type:

```
print f(1,3.0,"a value") << 2
```

If there is a breakpoint in a function called interactively, the value returned by the function is lost and any computation (for example, in an expression) following the function call is ignored. You can debug a function by setting breakpoints and calling it interactively.

You can use string arguments with *ccall*; for example:

```
print strcmp("abcd",strp)
```

In addition, you can also have breakpoints in a function called interactively. It is up to you to eventually return. Any stack trace (*where* command) done while stopped in a routine executed interactively shows functions up to the interactive call with a line:

```
<stopped in interactive call>
```

as the end marker of the local interactive call stack.

Interactive calls nest properly. This means that 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. Use the *where* command to report on the depth of nesting, if applicable.

To unstack the calls, complete the call (*cont*, *return*, *next*, or *step*) as many times as necessary, or *rerun* the program being debugged. Unfortunately, there are no other ways to unstack the interactive call(s).

Your breakpoints in functions called interactively do not respect the nesting. This can cause confusion if you attempt to have various breakpoints at different nesting levels. Breakpoints are all effectively at one level, and are always active.

Only one level of activation stack is visible at a time; in an interactive call the stack trace of the hidden levels is invisible to you and to *dbx*. This can provoke:

```
<variable> is not visible
```

messages if you are actively tracing. For example, suppose you are in foo() and a "stop in foo if z==5" command is in effect. You do an interactive call. Delete such a "stop" to stop the messages, and re-enter the "stop" once out of the interactive call.

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

11. Debugging at the Machine Level

This chapter explains how to debug at machine level by:

- setting breakpoints
- executing single lines of code
- tracing variables
- printing the contents of memory addresses
- disassembling the source code

Additional information is in Chapter 4, "Expressions and Precedence."

Examples:

```
$pc-40/10i

&z/8x

$sp/20X

stopi at 0x400abc

nexti

stepi
```

Syntax	Select this command to...
addr/<*count*><*mode*>	Print the contents of the specified address, *addr*, for the specified count, *count*. The *modes* are listed at the end of this chapter.
addr/*count* `L` *val mask*	Print those words at *addr* that match *val* after ANDing with *mask*. Examine *count* words for a match.
`conti` *sig*	Send the specified signal, *sig*, and tell *dbx* to continue.
`conti in` *proc*	Tell *dbx* to continue until the beginning of the specified procedure, *proc*.
`conti to` *addr*	Tell *dbx* to continue until reaching the specified address, *raddr*.
`conti` *sig* `to` *addr*	Tell *dbx* to continue until reaching the specified address, *addr*, then send the specified signal, *sig*.
`conti` *sig* `in` *proc*	Tell *dbx* to continue until reaching the beginning of the specified procedure, *proc*, then send the signal, *sig*.
`nexti`[*integer*]	Step over the specified number of machine instructions. The default is one. This command does not step into procedures.

Table 11-1. Machine Level Debugging Commands

Syntax	Select this command to...
stepi [*integer*]	Step the specified number of machine instructions. This command steps into procedures even if no source, symbols, or line numbers are present. The default is one.
stopi at	Stop *dbx* at the current line.
stopi at *addr*	Stop *dbx* at the specified address, *addr*.
stopi at *addr***if** *exp*	Stop *dbx* at the specified address only if the expression, *exp*, is true.
stopi *var* **at**	Stop *dbx* at the current line and check to see if the specified variable, *var*, has changed. If so, *dbx* prints the old and new values of the variables.
stopi[*var*] **at** *addr*	Stop *dbx* at the specified address, *addr*, and check to see if the specified variable, *var*, has changed. If so, *dbx* prints the old and new values of the variables.
stopi [*var*] **at** *addr***if** *exp*	Stop *dbx* at the specified address only if the expression, *exp*, is true. If stopped, *dbx* checks to see if the specified variable, *var*, has changed. If so, *dbx* prints the old and new values of the variables.
stopi if *exp*	Stop *dbx* if the specified expression, *exp*, is true.

Table 11-1. (continued) Machine Level Debugging Commands

Syntax	Select this command to...
`stopi` *var* `if` *exp*	Stop *dbx* if the specified variable, *var*, changes and the specified expression, *exp*, is true.
`stopi in` *proc*	Stop *dbx* at the beginning of the specified procedure, *proc*.
`stopi` *var* `in` *proc*	Stop *dbx* in the specified procedure, *proc*, when the specified variable, *var*, changes.
`stopi in` *proc* `if` *exp*	Stop *dbx* in the specified procedure, *proc*, if the specified expression, *exp*, is true. *dbx* checks *exp* before *var*.
`stopi` *var* `in` *proc* `if` *exp*	Stop *dbx* in the specified procedure, *proc*, when the variable, *var*, changes and the expression, *exp*, is true. *dbx* checks *exp* before *var*.
`tracei` *var*	Trace the variable at each machine instruction. Execution will be very slow.
`tracei [`*var*`] at` *addr* `[if` *exp]*	Trace the variable in machine instructions.
`tracei [`*var*`] in` *proc* `[if` *exp]*	Trace the specified variable in machine instructions. Execution of *proc* will be very slow.

Table 11-1. (continued) Machine Level Debugging Commands

11.1 Setting Breakpoints (*stopi*)

The *stopi* commands set breakpoints in machine code. These commands work in the same way as the *stop at*, *stop i*, and *stop if* commands described in Chapter 9, "Setting Breakpoints." There are two exceptions. The *stopi* command steps in units of machine instructions instead of in lines of code. Also, the *stop at* command requires an address rather than a line number. See the earlier discussion of *stop* for details on these complex statements. For example:

```
stopi at 0x434500
stopi in funcx
```

The second example stops at the first machine instruction in function *funcx*. A *where* command at the point of stop may yield an incorrect stack trace since the stack for the function is not completely set up until several machine instructions have been executed. This version of *dbx* does not know how to correctly report in complete stack frames.

11.2 Continuing after Breakpoints (*conti*)

The *conti* commands continue executing assembly code after a breakpoint. Turn to the beginning of this chapter for the syntax of this command.

11.3 Executing Single Lines (*stepi* and *nexti*)

The *stepi* and *nexti* commands execute a fixed number of machine instructions, as specified by *exp*. If you don't specify *exp* for *stepi* and *nexti*, *dbx* executes one machine instruction. If you do specify *exp*, *dbx* executes the machine instructions according to the following rules.

- With *stepi* and *nexti* the program executes *exp* machine instructions, ignoring any comment lines interspersed among them.
- With *stepi*, *exp* applies to the current procedure as well as procedure calls (*jal* and *jalr*). The program stops after executing *exp* instructions.

- With *nexti*, *exp* applies only to the current procedure. The program stops after executing *exp* instructions in the current procedure, ignoring any instructions executed in procedure calls.
- Use *stepi* and *nexti* to execute source lines after a breakpoint.

You can use all *dbx* variables for *$nextbreak* and *$stepinto* with the *stepi* and *nexti* commands exactly as with the *step* and *next* commands.

11.4 Tracing Variables (*tracei*)

The *tracei* commands track changes to variables, one instruction at a time. The *tracei* commands work for machine instruction as the *trace* commands do for lines of source code. The *tracei* command traces in units of machine instructions instead of in lines of code. (See the discussion of *trace* in Chapter 9 for details.) For example:

```
tracei x
```

11.5 Printing the Contents of Memory

Entering values in the syntax shown below prints the contents of memory according to the specifications that follow.

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. This might be termed the "examine forward" command.

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 (the "examine backward" command).

address/count **L** *val mask*

Examines *count* 32-bit words in increasing address. Print those 32-bit words which, when ORed with *mask*, equals *val*. This command therefore searches memory for specific patterns.

. Repeats the previous examine command with increasing address.

.? Repeats the previous examine command with decreasing address.

Command	Examine command formats:
i	print machine instructions (disassemble)
d	print a 16-bit word in decimal
D	print a 32-bit word in decimal
o	print a 16-bit word in octal
O	print a 32-bit word in octal
x	print a 16-bit word in hexadecimal
X	print a 32-bit word in hexadecimal
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

Table 11-2. Disassemble Commands

For example, to print 20 disassembled machine instructions starting at the current pc-20, type:

```
$curpc-20/20i
```

To print 32-bit words starting at address 0x400200 whose least significant byte is hexadecimal ee (100 words are inspected), type:

```
0x400200/100L 0xee 0xff
```

12. Multi-Process Debugging

This chapter explains multi-process debugging procedures, including:

- listing available running processes
- adding a process to the available pool
- listing the available processes
- selecting processes
- suspending the active running process
- resuming suspended processes
- freeing processes from the *dbx* pool
- returning freed processes to the operating system
- terminating active processes
- using forks and execs
- debugging process groups

Examples:

```
addproc 1234

kill 1234

resume pid 3456

showproc

stop pgrp

suspend pid 4365

waitall
```

Syntax	Select this command to...
active [*pid*]	Tell *dbx* which is the active process in the pool of *dbx* controlled processes. *pid* is the process identification number (PID#) of the process you want to select as active. If *pid* is not specified, *dbx* prints the currently active process id.
addproc *pid*	Add the specified process to the pool of *dbx* controlled processes.
delproc *pid*	Delete the specified process from the pool of *dbx* controlled processes.
kill	Kill the active process.
kill *pid...*	Kill the active process(es) whose PIDs are specified.
resume	Resume execution of the program, and return immediately to the *dbx* command interpreter.
resume *signal*	Resume execution of the process, sending it signal, *signal*, and return immediately to the *dbx* command interpreter.
showproc [*pid*|**all**]	Show processes currently available for debugging under *dbx*. If you use no arguments, *dbx* lists the processes it already controls.
wait	Wait for the active process to stop for an event.

Table 12-1. Multi-process Debugging Commands

Syntax	Select this command to...
`waitall`	Wait for any process currently running to breakpoint or stop for any reason.
`suspend`	Suspend the active process if it is running. If it is not running, do nothing.
`suspend pid` *pid*	Suspend the process *pid* if it is running. If it is not running, do nothing.

Table 12-1. (continued) Multi-process Debugging Commands

12.1 Processes

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 done with the process, you can terminate it, return it to the pool, or return it to the operating system.

Many commands now take a clause, **`pid`** *pid* (where *pid* is a numeric process id or a debugger variable holding a process id) at the end to make them apply to process *pid*. Commands that do this include:

`active`	`edit`	`readsyms`	`use`
`addproc`	`file`	`resume`	`wait`
`assign`	`func`	`return`	`whatis`
`catch`	`goto`	`showproc`	`when`
`cont[i]`	`ignore`	`status`	`where`
`delete`	`kill`	`step[i]`	`whereis`
`delproc`	`next`	`stop[i]`	`which`
`directory`	`print`	`suspend`	
`down`	`printf`	`trace[i]`	
`dump`	`printregs`	`up`	

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.

Debugger variables help write multiple-process scripts independent of process id:

- *$lastchild* is always set to the process id of the last child *fork*ed or *sproc*ed.
- *$pid0* is always set to the process id of the given process.

12.2 Listing Available Processes (*showproc*)

Use the *showproc* command to list the available processes. The *showproc* command can take either of two optional arguments: a *pid* (process identification number, PID#), or the word *all*. If you specify a PID number with *showproc*, *dbx* lists the status of the specified process.

If you use *showproc* with the command 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.

If you use *showproc* without command arguments, *dbx* lists the processes it already controls (currently in the process pool).

For example:

```
showproc 2355
showproc all
```

12.3 Adding a Process (*addproc*)

To add a process to the process pool, use the *addproc* command. The argument for the *addproc* command is the PID # of the process that you want to add. Adding the process to the *dbx* process pool automatically stops the process.

For example:

```
addproc 345
```

12.4 Removing a Process (*delproc*)

To remove a process from the process pool, use the *delproc* command. The argument for the *delproc* command is the PID # of the process that you want to delete. Removing the process from the *dbx* process pool automatically stops the process.

For example:

```
delproc 134 385
delproc pid 37
delproc
```

12.5 Selecting a Process (*active*)

dbx allows you to seize control of a number of processes. By default, *dbx* commands apply only to the active process. To select a process from the process pool to be the active process, use the *active* command. The argument to the *active* command is the PID# of the process in question. If you do not specify an argument, *dbx* lists the currently active process.

For example:

```
active 2355
```

12.6 Suspending a Process (*suspend*)

Adding a process to the *dbx* pool of controlled processes does not automatically stop the process. You can stop a *dbx* controlled process only if it is the currently active process (use the *active* command). Use the *suspend* command to stop the currently active process.

For example:

```
suspend  pid 2355
suspend
```

12.7 Resuming a Suspended Process (*resume*)

To resume execution of a suspended *dbx* controlled process, use either the *cont* command or the *resume* command. *resume* returns immediately to the *dbx* command processor. If you use *cont*, you do not get the *dbx* command interpreter back until the program encounters an event (e.g., a breakpoint).

The *resume* command can be very useful if you are debugging more than one process. With *resume*, you are free to select and debug a process while another process is running. The argument for the *resume* command is *signal*.

For example:

```
resume pid 28
resume
```

12.8 Waiting for a Resumed Process (*wait*)

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 see *waitall*, later in this chapter.

For example:

```
wait
wait pid 347
```

12.9 Freeing a Process (*delproc*)

To free a process from the control of *dbx*, use the *delproc* command. The argument for this command is the PID # of the process. For example:

```
delproc 2355
```

12.10 Killing a Process (*kill*)

To kill a process in the process pool while in *dbx*, you can use the *kill* command. For example:

```
kill 2355
```

12.11 Forks

When a program forks and starts another process, *dbx* allows you to add that process to the process pool. You can set the variable *$promptonfork* to a 1 or 2.

If *$promptonfork=1*, when a program forks, *dbx* asks if you want to add the new process to the process pool. If you leave *$promptonfork* set to zero (the default), *dbx* ignores the process created by the fork.

If *$promptonfork=2*, new forked processes are automatically added to the process pool.

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("child0);
        } else {
                printf("parent0);
        }
}
```

If you set:

```
$promptonfork=1
```

and run *fork* under *dbx*, the system prompts you to add the new process to the process pool.

If you set:

```
$promptonfork=2
```

and run *fork* under *dbx*, the new process is automatically added to the pool, and both the parent and child processes stop executing at the *fork*.

When *$promptonfork* is zero, *dbx* doesn't stop at forks.

12.12 Execs

An *exec* is a call from within a program that executes another program. During an *exec*, the first program gives up its process number to the program it executes. You can use *dbx* to follow an *exec*.

Consider the programs *exec1.c* and *exec2.c*:

```
main()
{
        printf("in exec1\n");        /*
         * Invoke the "exec2" program
         */
        execl("exec2", "exec2", 0);        /*
         * We'll only get here if execl() fails
         */
        perror("execl");
}

main()
{
        printf("in exec2\n");
}
```

If you run *exec1* under *dbx*, the system pauses to reread the symbolic information. Enter the *dbx* command, *cont*, to continue executing, now executing *exec2*.

12.13 Process Group Debugging

The process group facility allows a group of processes to be operated on simultaneously by a single *dbx* command. This is far more convenient to use when dealing with *sproced* processes 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 Accellerator (PFA). When debugging such code and before running the program, be sure to:

```
set $mp_program=1
```

For *$mp_program*:

- if 0 (the default), *sproc* is treated like *fork*.
- if 1, *sproc* is treated specially. The children are allowed to run; they will block on multi-processor synchronization code emitted by mp FORTRAN code (if *$mp_program=1*, mp FORTRAN code is easier to work with).

Whenever a process *sprocs*, if the child is added to the process pool, the parent and child are added to the group list as well. The group list is simply a list of processes.

If the **pgrp** clause is added to the end of an applicable command (*delete, next*[*i*], *readsyms, resume, stop*[*i*], *status, suspend, trace*[*i*], or *when*), the command is applied to all the processes in the group.

If *$groupforktoo* is 1, then *forked* processes are added to the group automatically just as *sproced* processes are.

The commands:

stop (*operands*) **pgrp**

trace (*operands*) **pgrp**

when (*operands*) **pgrp**

each add to the *group history*. With these commands, you can add breakpoints to multiple processes with a single command. This group history is a numbered list that *showpgrp* shows.

The command:

delete *int* **pgrp**

deletes the history for group number *int*. Thus you can delete breakpoints from multiple processes with a single command.

Breakpoints set on the process group are recorded both in the group and in each process. Deleting breakpoints individually (even if set via a group command) is allowed.

The following commands are usable only on the processes in the group list.

addpgrp *pid...*
: Adds the process ids specified to the group list. Only processes in the process pool can be added to the group list.

delpgrp *pid...*
: Deletes the process ids specified from the group list.

showpgrp
: Shows the group process list and the group breakpoint list.

12.14 Waiting for Any Running Process (*waitall*)

To wait for any process currently running to breakpoint or stop for any reason, use the *waitall* command. It waits for all running processes in the process list, not just those in the group list. It does not make the process that stops first the active process.

Normally, you would use this command after *resume pgrp* or *resume*.

For example:

```
waitall
```

12.15 Multi-Process Debugging Examples

The following pages contain examples for:

- window process debugging
- complex multi-process debugging

12.15.1 Window Process Debugging

A process that calls *winopen()* forks unless *foreground()* is called. To debug such a process, try the following script.

```
set $pimode = 1
# ensure the child is added to the pool
set $promptonfork=2
# run up thru fork()
run
# assume we have stopped at the winopen() fork()
set $wpid = $lastchild
# set the interesting process as the one commands apply to
active $wpid
# now can "cont" the process, or set breakpoints, or ?
#
# A script is a good place to set breakpoints, as breakpoints
# set in the child are forgotten when you re-run the given
# process. Remember that control-c (and the edge INTERRUPT
# menu item) affect only dbx, not the child process we are
# continuing. Use "suspend" to halt the window process.
```

12.15.2 Complex Multiple Process Debugging

To debug a multiple process, try the following script. This script assumes you know ahead of time the sequence of operations of interest. You might try using the *record* and *unrecord* commands to save an interactive session which "got to the right point" as the basis for a script such as the one that follows.

You can "continue" only one process. You can "resume" several processes at a time with:

```
resume pgrp
```

Or, you can resume several processes to run simultaneously with the commands in sequence. For example:

```
resume pid 100
resume pid 101
```

You can see you are stopped at a breakpoint only when you "wait" for a process.

Assume that you want to control all children:

```
set $pimode = 1
# ensure that the child is added to the pool
set $promptonfork=2
# run up thru fork()
run
# assume we have stopped at the fork()
set $chld1 = $lastchild
# run to exec(), then assume fork is next
cont pid $chld1
cont pid $chld1
# on fork note process id
set $chld2 = $lastchild
# Set breakpoints in the program and so on.
# Now can "cont" the process, or set breakpoints, or ?
# Remember that control-c (and the edge INTERRUPT menu item)
# affect only dbx, not the child process we are continuing.
# Use "suspend pid <pid>" to halt process <pid>.
# The script ends here. Begin interactive debugging.
```

Now assume that you want to control only some children:

```
set $pimode = 1
# ensure that the child can be added to the pool
set $promptonfork=1
# run up thru fork()
run
n
# n answered "Add child to process pool (n if no)?"
# For example, if this process is really a popen and
# not interesting, continue executing the active process:
# up through "interesting" fork.
cont
y
# remember the pid
set $child1 = $lastchild
# run up to another fork()
cont pid $pid0
# remember another child
set $child2 = $lastchild
# Set breakpoints in various procs. Func names, variables,
# etc. are evaluated in context of the process named.
stop at func7 pid $child1
stop at func8 pid $child2
# and so on.
# and so on. Now can "cont" a process, or set breakpoints,
# or ? Remember that control-c (and the edge INTERRUPT menu
# item) affect only dbx, not the child process we are
# continuing. Use "suspend pid <pid>" to halt process <pid>.
# The script ends here. Begin interactive debugging.
```

Appendix A: *dbx* Command Summary

Table A-1 lists all commands (except for command line editing commands) and gives each command's alias, syntax, and brief description. For more information about a command, refer to the description of the command in the main part of this manual.

Command	Alias	Syntax	Select this command to:
/		*/regex*	Search ahead in the code for the specified string.
?		*?regex*	Search back in the code for the specified string.
!		*!string* *!int* *!-int*	Specify a command from history list.
active		active [*pid*]	List the active process; *pid* is the process id number. If no *pid*, list the currently active process.
addpgrp		addpgrp *pid...*	Add the process ids specified to the group list. Only processes in the process pool can be added to the group list.
addproc		addproc *pid*	Add specified process to the pool of *dbx* controlled processes.
alias		alias [*name* (*arg1,...argn*) "*string*"]	List all existing aliases, or, if *arg*, define a new alias.
assign	a	assign *exp1* = *exp2*	Assign the specified expression to a specified program variable.
catch		catch [*signal*]	List all signals that *dbx* catches, or, if *arg*, add a new signal to the catch list.
ccall		ccall *func* (*arg1,...argn*)	Call a function with given arguments.
cont	c	cont cont to *proc* cont to *line* cont *signal* to *line* cont *signal* in *proc*	Continue executing a program after a breakpoint.

Table A-1. Command Summary

Command	Alias	Syntax	Select this command to:
conti		conti *signal* conti to *addr* conti in *proc* conti *signal* to *addr* conti *signal* in *proc*	Continue executing assembly code after a breakpoint.
corefile		corefile corefile *core*	Display the name of the core file; if the corefile is currently used by *dbx*, display program data. If *core*, identify corefile name, as in that core file for program data.
delete	d	delete *exp1*,... *expn* delete all	Delete the specified item from the status list.
delpgrp		delpgrp *pid*...	Delete the process ids specified from the group list.
down		down [*exp*]	Move down the specified number of activation levels in the stack (default, one level).
dump		dump *proc* dump .	Print variable information about *proc*. Print global variable information for all procedures (.).
edit		edit [*file*]	Call an editor from *dbx*.
examine *addr*		*addr*/<*cnt*><*mode*>	Print the contents of the specified address or disassemble the code for the instruction at the specified address.
file	e	file [*file*]	Print the name of the current file or specified *file*.
func	f	func func *exp* func *proc*	Move to the specified procedure (activation level) or print the current activation level.

Table A-1. (continued) Command Summary

Command	Alias	Syntax	Select this command to:
givenfile		givenfile givenfile *name*	Set program to debug, as in running processes and read in *name*'s symbol table.
goto	g	goto *line*	Go to the specified *line*.
help	?	help	Print a list of *dbx* commands.
history	h	history	Print a list of the previous commands issued (default is 20).
ignore		ignore *signal*	List all signals that *dbx* does not catch, or add specified *signal* to ignore list.
kill		kill [*pid*...]	Kill the active process(es).
list	li	list list [*exp*:*int*] list [*exp*]	List the specified lines (default is 10).
next	n	next [*int*]	Step over the specified number of lines (default is 1). Does not step into procedures.
nexti	ni	nexti [*int*]	Step over the specified number of machine instructions (default is 1). Does not step into procedures.
playback input	pi	playback input *file*	Replay commands saved with the *record input* command in a text file.
playback output	po	playback output *file*	Replay *dbx* output saved with the *record output* command in a text file.
print	P	print *exp1*,... *expn*	Print the value of the specified expression.
printf	pd	printf "*string*" *exp1*,...*expn*	Print the value of the specified expression, using C string formatting.
printregs	pr	printregs	Print all register values.

Table A-1. (continued) Command Summary

Command	Alias	Syntax	Select this command to:
quit	q	quit	Exit *dbx*.
record input	ri	record input *file*	Record all commands typed to *dbx*.
record output	ro	record output *file*	Record all *dbx* commands.
resume		*resume* *resume signal*	Resume execution of the program (send it *signal*), and return immediately to *dbx*.
return		return [*proc*]	Continue executing until the procedure returns. If no procedure specified, *dbx* assumes the next procedure.
run		run [*arg1...argn*] [<*file1*] [>*file2*]	Run your program.
rerun	r	rerun [*arg1...argn*] [<*file1*] [>*file2*]	Run program again, using the same arguments specified to the *run* command.
set		set set *var* = *exp*	For the existing *dbx* variables and their values, assign a value to a variable, or define a new variable and assign a value to it.
sh		sh [sh *cmd*]	Call a shell from *dbx* or execute a shell command.
showpgrp		showpgrp	Show the group process list and the group breakpoint list.
showproc		showproc [*pid*|all]	Show processes currently available for debugging. With no arguments, list the processes under control.

Table A-1. (continued) Command Summary

Command	Alias	Syntax	Select this command to:
source		source [*file*]	Execute *dbx* commands from the specified *file*. If no file specified, *dbx* assumes that you want the file created with the *record input* command.
status	j	status	Print a list of currently set breakpoints, record commands and traces.
step	s	step [*int*]	Step the specified number of lines (default is 1). This command steps into procedures.
stepi	si	stepi [*int*]	Step the specified number of machine instructions (default is 1). This command steps into procedures.
stop	b bp	stop [*var*] at stop [*var*] at *line* stop [*var*] in *proc* stop [*var*] if *exp* stop [*var*] at *line* if *exp* stop [*var*] in *proc* if *exp*	Set a breakpoint at the specified point.
stopi		stopi [*var*] at *addr* stopi [*var*] in *proc* stopi [*var*] if *exp* stopi [*var*] at *addr* if *exp* stopi [*var*] in *proc* if *exp*	Set a breakpoint in machine code at the specified point.
suspend		suspend suspend *pid*	Suspend the active process or process *pid* if running. If it is not running, do nothing.

Table A-1. (continued) Command Summary

Command	Alias	Syntax	Select this command to:
syscall		syscall	Print list of system calls.
trace	tr	trace *var* trace *var* at *line* trace *var* in *proc* trace *var* at *line* if *exp* trace *var* in *proc* if *exp*	Trace the specified variable.
tracei		tracei *var* tracei *var* at *addr* tracei *var* in *proc* tracei *var* at *addr* if *exp* tracei *var* in *proc* if *exp*	Trace the specified variable in the machine instruction.
unalias		unalias *aliasname*	Remove the specified alias.
unset		unset *var*	Unset a *dbx* variable.
up		up [*exp*]	Move the specified number of activation levels up the stack (default is 1).
use		use [*dir1 dir2... dirn*]	Print a list of the source directories, or if directory name given, use new directories for the previous list.
wait		wait waitall	Wait for the process (any current process) to stop for an event.

Table A-1. (continued) Command Summary

Command	Alias	Syntax	Select this command to:
whatis		whatis *var*	Print the type declaration for the specified name.
when		when [*var*] [if *exp*] {*command-list*} when [*var*] at *line* [if *exp*] {*command-list*} when [*var*] in *proc* [if *exp*] {*command-list*}	Execute the specified *dbx* commands during execution.
where	t	where	Do a stack trace to show current activation levels.
whereis		whereis *var*	Print all qualifications of the specified variable name.
which		which *var*	Print the qualification of the variable name currently in use.

Table A-1. (continued) Command Summary

Appendix B: Sample Program

This program, *anthrax.c*, counts non-blank lines in a program.

■ C Program: anthrax

```
#include <stdio.h>

struct line {
        char    string[256];
        int     length;
        int     linenumber;

};typedef struct line LINETYPE;
void printline();

main(argc, argv)

int     argc;
char    **argv;{
        LINETYPE        line1;
        FILE            *fd;
        extern  FILE    *fopen();
        extern  char    *fgets();

        if (argc << 2) {
                fprintf (stderr, "Usage sam filename0);
                exit (1);
        } /* if */

        fd = fopen (argv[1], "r");

        if (fd == NULL) {
                fprintf (stderr, "cannot open %s0, argv[1]);
                exit(1);
```

```
        } /* if */
        /*loop through lines in a file and
    /*call a routine to print*/
        /*blank lines along with line numbers, line lengths*/

        while (fgets(line1.string, sizeof(line1.string), fd)
        != NULL)
{
              int     i;
              static  curlinenumber = 0;

              i = strlen(line1.string);
              if (i == 1 && line1.string[0] == '0)
          continue; /* don't count blank lines */

              line1.length = i;
              line1.linenumber = curlinenumber++;
              printline(&line1);
        } /* while */
} /* main */

void printline (pline)

LINETYPE        *pline;

{
        int     i;  /*dummy var entered to demo whereis cmd*/
        i = 0;
        fprintf (stdout, "%3d. (%3d) %s",
            pline->linenumber, pline->length, pline->string);
        fflush (stdout);
} /* printline */
```

Appendix C: Questions and Problems

Here are some of the most commonly asked *dbx* questions (and the answers):

Q: *How do I pass arguments to my program when using dbx?*

A: Use:

```
run <arg1> <argN>
rerun <arg1> <argN>
```

For example, if you are using the debugger, type:

```
run a 2.0 <data.in >res.out
```

To execute the same program in *csh* or the Bourne shell, type:

```
myprog 2 2.0 <data.in >res.out
```

dbx does not understand redirection other than the simple form presented in the example. It does not understand csh (>&) or Bourne shell (2>*file*) redirection.

Q: *What use is it to look at the registers? Is this for the user or for hackers?*

A: The registers are not usually of interest. Those interested in learning the machine instruction architecture might want to examine the registers.

However, you might be interested in the machine instructions at the point of a program fault. For this, you can do dissassembly. For example:

```
$pc-40/20i
```

prints 20 instructions disassembled, starting 10 instructions before the current program counter.

Q: *How do I add a child process to the process pool?*

A: Use *$promptonfork.*

It is most useful to control a child process at the point of a *fork* or *sproc*. Do this with:

```
$promptonfork=2
```

if the program does very few forks and you wish to control them all. Or set:

```
$promptonfork=1
```

if you only wish to control some of the process forks. Then rerun the given process.

Since it is somewhat tedious to rerun and respecify the necessary data to get to the "point of interest," try puting the needed commands in a script fine and executing that via a *playback input* (*pi*) command.

Note that in most cases, once a child process begins running on its own (assuming *$promptonfork=0*, for example), it is not very useful to intercept it. Nevertheless, you can intercept it and add it to the process pool by using *addproc*. For example:

```
addproc 12345
```

Q: *How do I look at the source of another function not in the current source file?*

A: Type:

file *filename*

For example:

```
file myfile.f
```

Then *list*, etc., will reference that file.

Caution: Many commands reset the current source file as a side effect. For example, after doing "up" or "down," the current file will change.

You can also use *func funcname*, for example:

```
func myfunc
```

If *myfunc* is active on the current activation stack, *myfunc* will become the current focus for the local variable and the current pc marker. Whether or not *myfunc* is active, the file containing *myfunc* source is the current source file for *list* commands, etc.

As explained previously, various commands change the source focus. The command:

```
file
```

will print the current source focus.

Q: *How do I display a value in hex, octal, binary?*

A: For hex, use *px*. For octal, use *po* (currently, there is no method to print binary).

For example:

```
px a
po b
printf ''%o0',b
printf ''%x0',a
```

Q: *I got the message* `segmentation violation core dumped`, *what should I do with core?*

A: *dbx* can tell you where you bombed out. For example, suppose you have a program named *a.out* that died in the library function *strcmp*.

Assuming the current directory contains *a.out*, *core*, and the source code of your program, type the IRIX command:

```
dbx a.out core
```

The *dbx* command:

```
t
```

will show the stack traceback. And the *dbx* command:

```
w
```

will say "source is not available." This is true, since you don't have the source to the library function *strcmp* available. The *dbx* command:

```
up
```

will move up one level in the activation stack. If you called *strcmp* directly with an invalid argument, the *dbx* command:

```
w
```

will then show your source code with the call to *strcmp* marked with ">".

If *strcmp* was called by some other library routine, you may need to repeat *up* until *w* prints source code from your source file.

Q: *How do I find the address of a variable, array, procedure, or function?*

A: You can use *print* or *px* to print the address. For example, suppose you have a variable named x1. Type:

```
print &x1
```

or

```
px &x1
```

which will print the address of x1.

Q: *How do I remove a breakpoint once it is set?*

A: Follow these steps:

1. Type **status** to get a list of *stop* (breakpoint), *trace*, and *when* commands.
2. Type **delete <***N***>**, where *N* is the entry number in the status list. For example:

```
delete 7
d 2
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Silicon Graphics, Inc.

Date _____

Your name _____

Title _____

Department _____

Company _____

Address _____

_____

_____

Phone _____

COMMENTS

Manual title and version _____

_____

Please list any errors, inaccuracies, or omissions you have found in this manual

_____

_____

_____

_____

Please list any suggestions you may have for improving this manual

_____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 45 MOUNTAIN VIEW, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Silicon Graphics, Inc.
Attention: Technical Publications
2011 Stierlin Road
Mountain View, CA 94043-1321