Debugging a Program With dbx

Sun™ Studio 10

Sun Microsystems, Inc.
www.sun.com

Part No 819-0489-10
January 2005, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Contents

Figures

Tables

Before You Begin

dbx is an interactive, source-level, command-line debugging tool. *Debugging a Program With* dbx is intended for programmers with a working knowledge of Fortran, C, or C++, and some understanding of the Solaris™ Operating System (Solaris OS), or the Linux operating system, and UNIX® commands, who want to debug an application using dbx commands.

How This Book Is Organized

Debugging a Program With dbx contains the following chapters and appendixes:

Chapter 1 gives you the basics of using dbx to debug an application.

Chapter 2 describes how to start a debugging session, discusses compilation options, and tells you how to save all or part of session and replay it later.

Chapter 3 describes how to set dbx environment variables to customize your debugging environment and how to use the initialization file, .dbxrc, to preserve changes and adjustments from session to session.

Chapter 4 tells you about visiting source files and functions; locating symbols; and looking up variables, members, types, and classes.

Chapter 5 describes how to run, attach to, detach from, continue execution of, stop, and rerun a program under dbx. It also tells you how to single-step through program code.

Chapter 6 describes how to set, clear, and list breakpoints and traces.

Chapter 7 tells you how to examine the call stack and how to debug a core file.

Chapter 8 shows you how to evaluate data; display the values of expressions, variables, and other data structures; and assign values to variables.

Chapter 9 describes how to use runtime checking to detect memory leak and memory access errors in your program automatically.

Chapter 10 describes the fix and continue feature of dbx that allows you to modify and recompile a source file and continue executing without rebuilding your entire program.

Chapter 11 tells you how to find information about threads.

Chapter 12 describes how to use `dbx` to debug OpenMP™ code.

Chapter 13 describes several `dbx` facilities that help you debug child processes.

Chapter 14 tells you how to use `dbx` to work with signals.

Chapter 15 describes `dbx` support of C++ templates, and the commands available for handling C++ exceptions and how `dbx` handles these exceptions.

Chapter 16 introduces some of the `dbx` facilities you can use to debug a Fortran program.

Chapter 17 describes how you can use `dbx` to debug an application that is a mixture of Java™ code and C JNI (Java™ Native Interface) code or C++ JNI code.

Chapter 18 tells you how to use event management and execution control command at the machine-instruction level, how to display the contents of memory at specific addresses, and how to display source code lines along with their corresponding machine instructions.

Chapter 19 explains the differences between ksh-88 and `dbx` commands.

Chapter 20 describes `dbx` support for program that use dynamically linked, shared libraries.

Appendix A focuses on `dbx` commands that change your program or its behavior when you run it under `dbx`.

Appendix B tells you how to manage events, and describes how `dbx` can perform specific actions when specific events occur in the program you are debugging.

Appendix C gives detailed syntax and functional descriptions of all of the `dbx` commands.

Typographic Conventions

TABLE P-1 Typeface Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type **`rm`** *filename*.

TABLE P-2 Code Conventions

Code Symbol	Meaning	Notation	Code Example
[ ]	Brackets contain arguments that are optional.	`O[`*`n`*`]`	`O4, O`
{ }	Braces contain a set of choices for a required option.	`d{y|n}`	`dy`
|	The "pipe" or "bar" symbol separates arguments, only one of which may be chosen.	`B{dynamic|static}`	`Bstatic`
:	The colon, like the comma, is sometimes used to separate arguments.	`R`*`dir`*`[:`*`dir`*`]`	`R/local/libs:/U/a`
…	The ellipsis indicates omission in a series.	`xinline=`*`f1`*`[,…`*`fn`*`]`	`xinline=alpha,dos`

Shell Prompts

Shell	Prompt
C shell	*machine-name*%
C shell superuser	*machine-name*#
Bourne shell, Korn shell, and GNU Bourne-Again shell	$
Superuser for Bourne shell, Korn shell, and GNU Bourne-Again shell	#

Supported Platforms

This Sun Studio release supports systems that use the SPARC® and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems for the version of the Solaris Operating System you are running are available in the hardware compatibility lists at `http://www.sun.com/bigadmin/hcl`. These documents cite any implementation differences between the platform types.

In this document, the term "x86" refers to 64-bit and 32-bit systems manufactured using processors compatible with the AMD64 or Intel Xeon/Pentium product families. For supported systems, see the hardware compatibility lists.

Accessing Sun Studio Software and Man Pages

The Sun Studio software and its man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the software, you must have your `PATH` environment variable set correctly (see "Accessing the Software" on page 29). To access the man pages, you must have the your `MANPATH` environment variable set correctly (see "Accessing the Man Pages" on page 30.).

For more information about the PATH variable, see the csh(1), sh(1), ksh(1), and bash(1) man pages. For more information about the MANPATH variable, see the man(1) man page. For more information about setting your PATH variable and MANPATH variables to access this release, see the installation guide or your system administrator.

Note – The information in this section assumes that your Sun Studio software is installed in the /opt directory on Solaris platforms and in the /opt/sun directory on Linux platforms. If your software is not installed in the default directory, ask your system administrator for the equivalent path on your system.

Accessing the Software

Use the steps below to determine whether you need to change your PATH variable to access the software.

To Determine Whether You Need to Set Your PATH Environment Variable

1. **Display the current value of the PATH variable by typing the following at a command prompt.**

```
% echo $PATH
```

2. **On Solaris platforms, review the output to find a string of paths that contain /opt/SUNWspro/bin. On Linux platforms, review the output to find a string of paths that contain /opt/sun/sunstudio10/bin.**

   If you find the path, your PATH variable is already set to access the software. If you do not find the path, set your PATH environment variable by following the instructions in the next procedure.

To Set Your PATH Environment Variable to Enable Access to the Software

- **On Solaris platforms, add the following to your PATH environment variable. If you have Forte Developer software, Sun ONE Studio software, or another release of Sun Studio software installed, add the following path before the paths to those installations.**

  /opt/SUNWspro/bin

- **On Linux platforms, add the following to your `PATH` environment variable.**

  `/opt/sun/sunstudio10/bin`

Accessing the Man Pages

Use the following steps to determine whether you need to change your `MANPATH` variable to access the man pages.

To Determine Whether You Need to Set Your `MANPATH` Environment Variable

1. **Request the `dbx` man page by typing the following at a command prompt.**

```
% man dbx
```

2. **Review the output, if any.**

   If the `dbx`(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in the next procedure for setting your `MANPATH` environment variable.

To Set Your `MANPATH` Environment Variable to Enable Access to the Man Pages

- **On Solaris platforms, add the following to your `MANPATH` environment variable.**

  `/opt/SUNWspro/`

- **On Linux platforms, add the following to your `MANPATH` environment variable.**

  `/opt/sun/sunstudio10`

Accessing the Integrated Development Environment

The Sun Studio integrated development environment (IDE) provides modules for creating, editing, building, debugging, and analyzing the performance of a C, C++, or Fortran application.

The command to start the IDE is `sunstudio`. For details on this command, see the `sunstudio`(1) man page.

The correct operation of the IDE depends on the IDE being able to find the core platform. The sunstudio command looks for the core platform in two locations:

- The command looks first in the default installation directory, `/opt/netbeans/3.5V` on Solaris platforms and `/opt/sun/netbeans/3.5V` on Linux platforms.
- If the command does not find the core platform in the default directory, it assumes that the directory that contains the IDE and the directory that contains the core platform are both installed in or mounted to the same location. For example, on Solaris platforms, if the path to the directory that contains the IDE is `/foo/SUNWspro`, the command looks for the core platform in `/foo/netbeans/3.5V`. On Linux platforms, if the path to the directory that contains the IDE is `/foo/sunstudio10`, the command looks for the core platform in `/foo/netbeans/3.5V`.

If the core platform is not installed or mounted to either of the locations where the `sunstudio` command looks for it, then each user on a client system must set the environment variable `SPRO_NETBEANS_HOME` to the location where the core platform is installed or mounted (*/installation_directory*`/netbeans/3.5V`).

On Solaris platforms, each user of the IDE also must add */installation_directory*`/SUNWspro/bin` to their `$PATH` in front of the path to any other release of Forte Developer software, Sun ONE Studio software, or Sun Studio software. On Linux platforms, each user of the IDE also must add */installation_directory*`/sunstudio10/bin` to their `$PATH` in front of the path to any other release of Sun Studio software.

The path */installation_directory*`/netbeans/3.5V/bin` should not be added to the user's `$PATH`.

Accessing Sun Studio Documentation

You can access the documentation at the following locations:

- The documentation is available from the documentation index that is installed with the software on your local system or network at `file:/opt/SUNWspro/docs/index.html` on Solaris platforms and at `file:/opt/sun/sunstudio10/docs/index.html` on Linux platforms.

  If your software is not installed in the `/opt` directory on a Solaris platform or the `/opt/sun` directory on a Linux platform, ask your system administrator for the equivalent path on your system.

- Most manuals are available from the `docs.sun.com`[sm] web site. The following titles are available through your installed software on Solaris platforms only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*
- The release notes for both Solaris platforms and Linux platforms are available from the `docs.sun.com` web site.
- Online help for all components of the IDE is available through the Help menu, as well as through Help buttons on many windows and dialogs, in the IDE.

The `docs.sun.com` web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index that is installed with the software on your local system or network.

Note – Sun is not responsible for the availability of third-party Web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: • *Standard C++ Library Class Reference* • *Standard C++ Library User's Guide* • *Tools.h++ Class Library Reference* • *Tools.h++ User's Guide*	HTML in the installed software on Solaris platforms through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes and man pages	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html` on Solaris platforms, and at `file:/opt/sun/sunstudio10/docs/index.html` on Linux platforms,
Online help	HTML available through the Help menu in the IDE
Release notes	HTML at `http://docs.sun.com`

Related Documentation

The following table describes related documentation that is available at `file:/opt/SUNWspro/docs/index.html` on Solaris platforms and at `http://docs.sun.com`. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Document Title	Description
dbx Readme	Lists new features, known problems, limitations, and incompatibilities of `dbx`.
dbx(1) man page	Describes the `dbx` command.
C User's Guide	Describes the Sun Studio 10 C programming language compiler along with ANSI C compiler-specific information.
C++ User's Guide	Instructs you in the use of the Sun Studio 10 C++ compiler and provides detailed information on command-line compiler options.
Fortran User's Guide	Describes the compile-time environment and command-line options for the Sun Studio 10 Fortran compiler.
OpenMP API User's Guide	Summarizes the OpenMP Fortran 95, C, and C++ application program interface (API) for building multiprocessing applications. Sun Studio compilers support the OpenMP API.
Performance Analyzer	Describes the performance analysis tools that are available with Sun Studio 10.

The following table describes related documentation that is available at `file:/opt/sun/sunstudio10/docs/index.html` on Linux platforms and at `http://docs.sun.com`. If your software is not installed in the `/opt/sun` directory, ask your system administrator for the equivalent path on your system.

Document Title	Description
dbx Readme	Lists new features, known problems, limitations, and incompatibilities of `dbx`.
dbx(1) man page	Describes the `dbx` command.
Performance Analyzer	Describes the performance analysis tools that are available with Sun Studio 10.

Accessing Related Solaris Documentation

The following table describes related documentation that is available through the `docs.sun.com` web site.

Document Collection	Document Title	Description
Solaris Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris operating system.
Solaris Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris Software Developer Collection	*Multithreaded Programming Guide*	Covers the POSIX and Solaris threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding tools for multithreaded programs.
Solaris Software Developer Collection	*SPARC Assembly Language Reference Manual*	Describes the assembler that runs on the SPARC® architecture and translates source files that are in assembly language format into object files in linking format.

Resources for Developers

Visit `http://developers.sun.com/prodtech/cc` to find these frequently updated resources:

- Articles on programming techniques and best practices
- A knowledge base of short programming tips
- Documentation of the software, as well as corrections to the documentation that is installed with your software
- Information on support levels
- User forums

- Downloadable code samples
- New technology previews

You can find additional resources for developers at http://developers.sun.com.

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

http://www.sun.com/service/contacting

Sending Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Submit your comments to Sun at this URL:

http://www.sun.com/hwdocs/feedback

Please include the part number (819-0489-10) of your document

CHAPTER 1

Getting Started With dbx

dbx is an interactive, source-level, command-line debugging tool. You can use it to run a program in a controlled manner and to inspect the state of a stopped program. dbx gives you complete control of the dynamic execution of a program, including collecting performance and memory usage data, monitoring memory access, and detecting memory leaks.

You can use dbx to debug an application written in C, C++, or Fortran. You can also, with some limitations (see "Limitations of dbx With Java Code" on page 230), debug an application that is a mixture of Java™ code and C JNI (Java™ Native Interface) code or C++ JNI code.

This chapter gives you the basics of using dbx to debug an application. It contains the following sections:

Compiling Your Code for Debugging

You must prepare your program for source-level debugging with dbx by compiling it with the -g option, which is accepted by the C compiler, C++ compiler, Fortran 95 compiler, and Java compiler. For more information, see "Compiling a Program for Debugging" on page 57.

Starting dbx and Loading Your Program

To start dbx, type the dbx command at a shell prompt:

```
$ dbx
```

To start dbx and load the program to be debugged:

```
$ dbx program_name
```

To start dbx and load a program that is a mixture of Java code and C JNI code or C++ JNI code:

```
$ dbx program_name{.class | .jar}
```

You can use the dbx command to start dbx and attach it to a running process by specifying the process ID.

```
$ dbx - process_id
```

If you don't know the process ID of the process, use the `ps` command to determine it, then use the `dbx` command to attach to the process. For example:

```
$ ps -def | grep Freeway
   fred 1855     1   1 16:21:36 ?         0:00 Freeway
   fred 1872  1865   0 16:22:33 pts/5     0:00 grep Freeway
$ dbx - 1855
Reading -
Reading ld.so.1
Reading libXm.so.4
Reading libgen.so.1
Reading libXt.so.4
Reading libX11.so.4
Reading libce.so.0
Reading libsocket.so.1
Reading libm.so.1
Reading libw.so.1
Reading libc.so.1
Reading libSM.so.6
Reading libICE.so.6
Reading libXext.so.0
Reading libnsl.so.1
Reading libdl.so.1
Reading libmp.so.2
Reading libc_psr.so.1
Attached to process 1855
stopped in _libc_poll at 0xfef9437c
0xfef9437c: _libc_poll+0x0004:ta      0x8
Current function is main
   48   XtAppMainLoop(app_context);
(dbx)
```

For more information on the `dbx` command and start-up options, see "dbx Command" on page 317 and the `dbx`(1) man page, or type `dbx -h`.

If you are already running `dbx`, you can load the program to be debugged, or switch from the program you are debugging to another program, with the `debug` command:

```
(dbx) debug program_name
```

To load or switch to a program that includes Java code and C JNI code or C++ JNI code:

```
(dbx> debug program_name{.class | .jar}
```

If you are already running dbx, you can also use the debug command to attach dbx to a running process:

```
(dbx) debug program_name process_id
```

To attach dbx to a running process that includes Java™ code and C JNI (Java™ Native Interface) code or C++ JNI code:

```
(dbx) debug program_name{.class | .jar} process_id
```

For more information on the debug command, see "debug Command" on page 320.

Running Your Program in dbx

To run your most recently loaded program in dbx, use the run command. If you type the run command initially without arguments, the program is run without arguments. To pass arguments or redirect the input or output of your program, use the following syntax:

```
run [ arguments ] [ < input_file ] [ > output_file ]
```

For example:

```
(dbx) run -h -p < input > output
Running: a.out
(process id 1234)
execution completed, exit code is 0
(dbx)
```

When you run an application that includes Java code, the run arguments are passed to the Java application, not to the JVM software. Do not include the main class name as an argument.

If you repeat the run command without arguments, the program restarts using the arguments or redirection from the previous run command. You can reset the options using the rerun command. For more information on the run command, see "run Command" on page 366. For more information on the rerun command, see "rerun Command" on page 364.

Your application may run to completion and terminate normally. If you have set breakpoints, it will probably stop at a breakpoint. If your application contains bugs, it may stop because of a memory fault or segmentation fault.

Debugging Your Program With dbx

You are likely to be debugging your program for one of the following reasons:

- To determine where and why it is crashing. Strategies for locating the cause of a crash include:
  - Running the program in `dbx`. `dbx` reports the location of the crash when it occurs.
  - Examining the core file and looking at a stack trace (see "Examining a Core File" on page 41 and "Looking at the Call Stack" on page 45).
- To determine why your program is giving incorrect results. Strategies include:
  - Setting breakpoints to stop execution so that you can check your program's state and look at the values of variables (see "Setting Breakpoints" on page 43 and "Examining Variables" on page 46).
  - Stepping through your code one source line at a time to monitor how the program state changes (see "Stepping Through Your Program" on page 44).
- To find a memory leak or memory management problem. Runtime checking lets you detect runtime errors such as memory access errors and memory leak errors and lets you monitor memory usage (see "Finding Memory Access Problems and Memory Leaks" on page 47).

Examining a Core File

To determine where your program is crashing, you may want to examine the core file, the memory image of your program when it crashed. You can use the `where` command (see "`where` Command" on page 399) to determine where the program was executing when it dumped core.

Note – `dbx` cannot tell you the state of a Java application from a core file as it can with native code.

To debug a core file, type:

```
$ dbx program_name core
```

or

```
$ dbx - core
```

In the following example, the program has crashed with a segmentation fault and dumped core. The user starts `dbx` and loads the core file. Then he uses the `where` command to display a stack trace, which shows that the crash occurred at line 9 of the file `foo.c`.

```
% dbx a.out core
Reading a.out
core file header read successfully
Reading ld.so.1
Reading libc.so.1
Reading libdl.so.1
Reading libc_psr.so.1
program terminated by signal SEGV (no mapping at the fault address)
Current function is main
    9       printf("string '%s' is %d characters long\n", msg,
strlen(msg));
(dbx) where
  [1] strlen(0x0, 0x0, 0xff337d24, 0x7efefeff, 0x81010100,
0xff0000), at
0xff2b6dec
=>[2] main(argc = 1, argv = 0xffbef39c), line 9 in "foo.c"
(dbx)
```

For more information on debugging core files, see "Debugging a Core File" on page 50. For more information on using the call stack, see "Looking at the Call Stack" on page 45.

Note – If your program is dynamically linked with any shared libraries, it is best to debug the core file in the same operating environment in which it was created. For information on debugging a core file that was created in a different operating environment, see "Debugging a Mismatched Core File" on page 51.

Setting Breakpoints

A breakpoint is a location in your program where you want the program to stop executing temporarily and give control to dbx. Set breakpoints in areas of your program where you suspect bugs. If your program crashes, determine where the crash occurs and set a breakpoint just before this part of your code.

When your program stops at a breakpoint, you can then examine the state of program and the values of variables. dbx allows you to set many types of breakpoints (see Chapter 6).

The simplest type of breakpoint is a stop breakpoint. You can set a stop breakpoint to stop in a function or procedure. For example, to stop when the main function is called:

```
(dbx) stop in main
(2) stop in main
```

For more information on the stop in command, see "Setting a stop Breakpoint in a Function" on page 103 and "stop Command" on page 375.

Or you can set a stop breakpoint to stop at a particular line of source code. For example, to stop at line 13 in the source file t.c:

```
(dbx) stop at t.c:13
(3) stop at "t.c":13
```

For more information on the stop at command, see "Setting a stop Breakpoint at a Line of Source Code" on page 102 and "stop Command" on page 375.

You can determine the line at which you wish to stop by using the file command to set the current file and the list command to list the function in which you wish to stop. Then use the stop at command to set the breakpoint on the source line:

```
(dbx) file t.c
(dbx) list main
10    main(int argc, char *argv[])
11    {
12        char *msg = "hello world\n";
13        printit(msg);
14    }
(dbx) stop at 13
(4) stop at "t.c":13
```

To continue execution of your program after it has stopped at a breakpoint, use the `cont` command (see "Continuing Execution of a Program" on page 97 and "`cont` Command" on page 316).

To get a list of all current breakpoints use the `status` command:

```
(dbx) status
(2) stop in main
(3) stop at "t.c":13
```

Now if you run your program, it stops at the first breakpoint:

```
(dbx) run
...
stopped in main at line 12 in file "t.c"
12        char *msg = "hello world\n";
```

Stepping Through Your Program

After you have stopped at a breakpoint, you may want to step through your program one source line at a time while you compare its actual state with the expected state. You can use the `step` and `next` commands to do so. Both commands execute one source line of your program, stopping when that line has completed execution. The commands handle source lines that contain function calls differently: the `step` command steps into the function, while the `next` command steps over the function. The `step up` command continues execution until the current function returns control to the function that called it.

Note – Some functions, notably library functions such as `printf`, may not have been compiled with the `-g` option, so `dbx` cannot step into them. In such cases, `step` and `next` perform similarly.

The following example shows the use of the `step` and `next` commands as well as the breakpoint set in "Setting Breakpoints" on page 43.

```
(dbx) stop at 13
(3) stop at "t.c":13
(dbx) run
Running: a.out
stopped in main at line 13 in file "t.c"
   13           printit(msg);
(dbx) next
Hello world
stopped in main at line 14 in file "t.c"
   14   }

(dbx) run
Running: a.out
stopped in main at line 13 in file "t.c"
   13           printit(msg);
(dbx) step
stopped in printit at line 6 in file "t.c"
    6           printf("%s\n", msg);
(dbx) step up
Hello world
printit returns
stopped in main at line 13 in file "t.c"
   13           printit(msg);
(dbx)
```

For more information on stepping through your program, see "Stepping Through a Program" on page 96. For more information on the `step` and `next` commands, see "`step` Command" on page 372 and "`next` Command" on page 353.

Looking at the Call Stack

The call stack represents all currently active routines—those that have been called but have not yet returned to their respective caller. In the stack, the functions and their arguments are listed in the order in which they were called. A stack trace shows where in the program flow execution stopped and how execution reached this point. It provides the most concise description of your program's state.

To display a stack trace, use the `where` command:

```
(dbx) stop in printf
(dbx) run
(dbx) where
  [1] printf(0x10938, 0x20a84, 0x0, 0x0, 0x0, 0x0), at 0xef763418
=>[2] printit(msg = 0x20a84 "hello world\n"), line 6 in "t.c"
  [3] main(argc = 1, argv = 0xefffe93c), line 13 in "t.c"
(dbx)
```

For functions that were compiled with the `-g` option, the arguments names and their types are known so accurate values are displayed. For functions without debugging information hexadecimal numbers are displayed for the arguments. These numbers are not necessarily meaningful. For example, in the stack trace above, frame 1 shows the contents of the SPARC input registers `$i0` through `$i5`; only the contents of registers `$i0` through `$i1` are meaningful since only two arguments were passed to `printf` in the example shown in "Stepping Through Your Program" on page 44.

You can stop in a function that was not compiled with the `-g` option. When you stop in such a function `dbx` searches down the stack for the first frame whose function is compiled with the `-g` option—in this case `printit()`—and sets the current scope (see "Program Scope" on page 76) to it. This is denoted by the arrow symbol (`=>`).

For more information on the call stack, see Chapter 7.

Examining Variables

While a stack trace may contain enough information to fully represent the state of your program, you may need to see the values of more variables. The `print` command evaluates an expression and prints the value according to the type of the expression. The following example shows several simple C expressions:

```
(dbx) print msg
msg = 0x20a84 "Hello world"
(dbx) print msg[0]
msg[0] = 'h'
(dbx) print *msg
*msg = 'h'
(dbx) print &msg
&msg = 0xefffe8b4
```

You can track when the values of variables and expressions change using data change breakpoints (see "Setting Data Change Breakpoints" on page 106). For example, to stop execution when the value of the variable count changes, type:

```
(dbx) stop change count
```

Finding Memory Access Problems and Memory Leaks

Runtime checking consists of two parts: memory access checking, and memory use and leak checking. Access checking checks for improper use of memory by the debugged application. Memory use and leak checking involves keeping track of all the outstanding heap space and then on demand or at termination of the program, scanning the available data spaces and identifying the space that has no references.

Note – Runtime checking is available only on Solaris platforms.

Memory access checking, and memory use and leak checking, are enabled with the `check` command. To turn on memory access checking only, type:

```
(dbx) check -access
```

To turn on memory use and memory leak checking, type:

```
(dbx) check -memuse
```

After turning on the types of runtime checking you want, run your program. The program runs normally, but slowly because each memory access is checked for validity just before it occurs. If `dbx` detects invalid access, it displays the type and location of the error. You can then use `dbx` commands such as `where` to get the current stack trace or `print` to examine variables.

Note – You cannot use runtime checking on an application that is a mixture of Java code and C JNI code or C++ JNI code.

For detailed information on using runtime checking, see Chapter 9.

Quitting dbx

A dbx session runs from the time you start dbx until you quit dbx; you can debug any number of programs in succession during a dbx session.

To quit a dbx session, type **quit** at the dbx prompt.

```
(dbx) quit
```

When you start dbx and attach it to a running process using the *process_id* option, the process survives and continues when you quit the debugging session. dbx performs an implicit detach before quitting the session.

For more information about quitting dbx, see "Quitting Debugging" on page 59.

Accessing dbx Online Help

dbx includes a help file that you can access with the help command:

```
(dbx) help
```

CHAPTER 2

Starting dbx

This chapter explains how to start, execute, save, restore, and quit a dbx debugging session. It contains the following sections:

Starting a Debugging Session

How you start dbx depends on what you are debugging, where you are, what you need dbx to do, how familiar you are with dbx, and whether or not you have set up any dbx environment variables.

The simplest way to start a dbx session is to type the dbx command at a shell prompt.

```
$ dbx
```

To start dbx from a shell and load a program to be debugged, type:

```
$ dbx program_name
```

To start dbx and load a program that is a mixture of Java code and C JNI code or C++ JNI code:

```
$ dbx program_name{.class | .jar}
```

For more information on the dbx command and start-up options, see "dbx Command" on page 317 and the dbx(1) man page.

Debugging a Core File

If the program that dumped core was dynamically linked with any shared libraries, it is best to debug the core file in the same operating environment in which it was created. dbx has limited support for the debugging of "mismatched" core files (for example, core files produced on a system running a different version or patch level of the Solaris operating environment.

Note – dbx cannot tell you the state of a Java application from a core file as it can with native code.

Debugging a Core File in the Same Operating Environment

To debug a core file, type:

```
$ dbx program_name core
```

You can also debug a core file using the debug command when dbx is already running:

```
(dbx) debug -c core program_name
```

You can substitute - for the program name and dbx will attempt to extract the program name from the core file. dbx may not find the executable if its full path name is not available in the core file. If this happens, specify the complete path name of the binary when you tell dbx to load the core file.

If the core file is not in the current directory, you can specify its path name (for example, `/tmp/core`).

Use the `where` command (see "`where` Command" on page 399) to determine where the program was executing when it dumped core.

When you debug a core file, you can also evaluate variables and expressions to see the values they had at the time the program crashed, but you cannot evaluate expressions that make function calls. You cannot single step or set breakpoints.

If Your Core File Is Truncated

If you have problems loading a core file, check whether you have a truncated core file. If you have the maximum allowable size of core files set too low when the core file is created, then `dbx` cannot read the resulting truncated core file. In the C shell, you can set the maximum allowable core file size using the `limit` command (see the `limit`(1) man page). In the Bourne shell and Korn shell, use the `ulimit` command (see the `limit`(1) man page). You can change the limit on core file size in your shell start-up file, re-source the start-up file, and then rerun the program that produced the core file to produce a complete core file.

If the core file is incomplete, and the stack segment is missing, then stack retrace information is not available. If the runtime linker information is missing, then the list of loadobjects is not available. In this case, you get an error message about `librtld_db.so` not being initialized. If the list of LWPs is missing, then no thread information, lwp information, or stack retrace information is available.If you run the `where` command, you get an error saying the program was not "active."

Debugging a Mismatched Core File

Sometimes a core file is created on one system (the core-host) and you want to load the core file on another machine (the `dbx`-host) to debug it. However, two problems with libraries may arise when you do so:

- The shared libraries used by the program on the core-host may not be the same libraries as those on the dbx-host. To get proper stack traces involving the libraries, you'll want to make these original libraries available on the dbx-host.
- `dbx` uses system libraries in `/usr/lib` to help understand the implementation details of the run time linker and threads library on the system. It may also be necessary to provide these system libraries from the core-host so that `dbx` can understand the runtime linker data structures and the threads data structures.

The user libraries and system libraries can change in patches as well as major Solaris operating environment upgrades, so this problem can even occur on the same host, if, for example, a patch was installed after the core file was collected, but before running `dbx` on the core file.

`dbx` may display one or more of the following error messages when you load a "mismatched" core file:

```
dbx: core file read error: address 0xff3dd1bc not available
dbx: warning: could not initialize librtld_db.so.1 -- trying
libDP_rtld_db.so
dbx: cannot get thread info for 1 -- generic libthread_db.so error
dbx: attempt to fetch registers failed - stack corrupted
dbx: read of registers from (0xff363430) failed -- debugger service
failed
```

Eliminating Shared Library Problems

To eliminate the library problems and debug a "mismatched" core file with `dbx`, you can now do the following:

1. **Set the `dbx` environment variable `core_lo_pathmap` to `on`.**
2. **Use the `pathmap` command to tell `dbx` where the correct libraries for the core file are located.**
3. **Use the `debug` command to load the program and the core file.**

   For example, assuming that the root partition of the core-host has been exported over NFS and can be accessed via `/net/core-host/` on the `dbx`-host machine, you would use the following commands to load the program `prog` and the core file `prog.core` for debugging:

```
(dbx) dbxenv core_lo_pathmap on
(dbx) pathmap /usr /net/core-host/usr
(dbx) pathmap /appstuff /net/core-host/appstuff
(dbx) debug prog prog.core
```

   If you are not exporting the root partition of the core-host, you must copy the libraries by hand. You need not re-create the symbolic links. (For example, you need not make a link from `libc.so` to `libc.so.1`; just make sure `libc.so.1` is available.)

Things to Remember

Keep the following things in mind when debugging a mismatched core file:

- The `pathmap` command does not recognize a pathmap for '/' so you cannot use the following command:

  ```
  pathmap / /net/core-host
  ```

- The single-argument mode for the `pathmap` command does not work with loadobject pathnames, so use the two argument from-path to-path mode.
- Debugging the core file is likely to work better if the dbx-host has either the same or a more recent version of the Solaris operating environment than the core-host, though this is not always necessary.
- The system libraries that you might need are:
  - For the runtime linker:

    ```
    /usr/lib/ld.so.1
    /usr/lib/librtld_db.so.1
    /usr/lib/sparcv9/ld.so.1
    /usr/lib/sparcv9/librtld_db.so.1
    ```

  - For the threads library, depending on which implementation of `libthread` you are using:

    ```
    /usr/lib/libthread_db.so.1
    /usr/lib/sparcv9/libthread_db.so.1
    /usr/lib/lwp/libthread_db.so.1
    /usr/lib/lwp/sparcv9/libthread_db.so.1
    ```

    The `/usr/lib/lwp` files apply only if you are running dbx in the Solaris 8 operating environment and only if you are using the alternate `libthread` library.

    You will need the SPARC-V9 versions of the *xxx*`_db.so` libraries if `dbx` is running on a 64-bit capable version of the Solaris operating environment (if the command `isalist` displays sparcv9) since these system libraries are loaded and used as part of `dbx`, not as part of the target program.

    The `ld.so.1` libraries are part of the core file image like `libc.so` or any other library, so you need the SPARC or SPARC-V9 `ld.so.1` library that matches the program that created the core file.

- If you are looking at a core file from a threaded program, and the `where` command does not display a stack, try using `lwp` commands. For example:.

```
(dbx) where
current thread: t@0
[1] 0x0(), at 0xffffffff
(dbx) lwps
o>l@1 signal SIGSEGV in _sigfillset()
(dbx) lwp l@1
(dbx) where
=>[1] _sigfillset(), line 2 in "lo.c"
  [2] _liblwp_init(0xff36291c, 0xff2f9740, ...
  [3] _init(0x0, 0xff3e2658, 0x1, ...
...
```

The lack of a thread stack can indicate a problem with `thread_db.so.1` Therefore, you might also want to try copying the proper `libthread_db.so.1` library from the core-host.

Using the Process ID

You can attach a running process to `dbx` using the process ID as an argument to the `dbx` command.

```
$ dbx program_name  process_id
```

To attach `dbx` to a running process that includes Java™ code and C JNI (Java™ Native Interface) code or C++ JNI code:

```
$ dbx program_name{.class | .jar} process_id
```

You can also attach to a process using its process ID without knowing the name of the program.

```
$ dbx - process_id
```

Because the program name remains unknown to `dbx`, you cannot pass arguments to the process in a `run` command.

For more information, see "Attaching dbx to a Running Process" on page 94.

The dbx Startup Sequence

When you start dbx,if you do not specify the -S option, dbx looks for the installed startup file, dbxrc, in the directory */installation_directory*/lib. (The default *installation_directory* is /opt/SUNWspro on Solaris platforms and /opt/sun/sunstudio9 on Linux platforms.) If your Sun Studio software is not installed in the default directory, dbx derives the path to the dbxrc file from the path to the dbx executable.

Then dbx searches for a .dbxrc file in the current directory, then in $HOME. You can specify a different startup file than .dbxrc explicitly by specifying the file path using the -s option. For more information, see "Using the dbx Initialization File" on page 65.

A startup file may contain any dbx command, and commonly contains alias, dbxenv, pathmap, and Korn shell function definitions. However, certain commands require that a program has been loaded or a process has been attached to. All startup files are loaded before the program or process is loaded. The startup file may also source other files using the source or .(period) command. You can also use the startup file to set other dbx options.

As dbx loads program information, it prints a series of messages, such as Reading *filename*.

Once the program is finished loading, dbx is in a ready state, visiting the "main" block of the program (for C or C++: main(); for Fortran 95: MAIN()). Typically, you set a breakpoint (for example, stop in main) and then issue a run command for a C program.

Setting Startup Properties

You can use the pathmap, dbxenv, and alias commands to set startup properties for your dbx sessions.

Mapping the Compile-time Directory to the Debug-time Directory

By default, dbx looks in the directory in which the program was compiled for the source files associated with the program being debugged. If the source or object files are not there or the machine you are using does not use the same path name, you must inform dbx of their location.

If you move the source or object files, you can add their new location to the search path. The pathmap command creates a mapping from your current view of the file system to the name in the executable image. The mapping is applied to source paths and object file paths.

Add common pathmaps to your .dbxrc file.

To establish a new mapping from the directory *from* to the directory *to*, type:

```
(dbx) pathmap [ -c ] from to
```

If -c is used, the mapping is applied to the current working directory as well.

The pathmap command is useful for dealing with automounted and explicit NFS-mounted file systems with different base paths on differing hosts. Use -c when you try to correct problems due to the automounter because current working directories are inaccurate on automounted file systems.

The mapping of /tmp_mnt to / exists by default.

For more information, see "pathmap Command" on page 356.

Setting dbx Environment Variables

You can use the dbxenv command to either list or set dbx customization variables. You can place dbxenv commands in your .dbxrc file. To list variables, type:

```
$ dbxenv
```

You can also set dbx environment variables. See Chapter 3 for more information about the .dbxrc file and about setting these variables.

For more information, see "Setting dbx Environment Variables" on page 66 and "dbxenv Command" on page 320.

Creating Your Own dbx Commands

You can create your own dbx commands using the kalias or dalias commands. For more information, see "dalias Command" on page 316.

Compiling a Program for Debugging

You must prepare your program for debugging with dbx by compiling it with the -g or -g0 option.

The -g option instructs the compiler to generate debugging information during compilation.

For example, to compile using C++, type:

```
% CC -g example_source.cc
```

In C++, the -g option turns on debugging and turns off inlining of functions. The -g0 (zero) option turns on debugging and does not affect inlining of functions. You cannot debug inline functions with the -g0 option. The -g0 option can significantly decrease link time and dbx start-up time (depending on the use of inlined functions by the program).

To compile optimized code for use with dbx, compile the source code with both the -O (uppercase letter O) and the -g options.

Debugging Optimized Code

The dbx tool provides partial debugging support for optimized code. The extent of the support depends largely upon how you compiled the program.

When analyzing optimized code, you can:

- Stop execution at the start of any function (stop in *function* command)
- Evaluate, display, or modify arguments
- Evaluate, display, or modify global or static variables
- Single-step from one line to another (next or step command)

However, with optimized code, dbx cannot evaluate, display, or modify local variables

When programs are compiled with optimization and debugging enabled at the same time (using the `-O -g` options), `dbx` operates in a restricted mode.

The details about which compilers emit which kind of symbolic information under what circumstances is considered an unstable interface and is likely to change from release to release.

Source line information is available, but the code for one source line might appear in several different places for an optimized program, so stepping through a program by source line results in the "current line" jumping around in the source file, depending on how the code was scheduled by the optimizer.

Tail call optimization can result in missing stack frames when the last effective operation in a function is a call to another function.

Generally, symbolic information for parameters, local variables, and global variables is available for optimized programs. Type information about structs, unions, C++ classes, and the types and names of local variables, global variables, and parameters should be available. Complete information about the location of these items in the program is not available for optimized programs. The C++ compiler does not provide symbolic type information about local variables; the C compiler does.

Code Compiled Without the `-g` Option

While most debugging support requires that a program be compiled with `-g`, `dbx` still provides the following level of support for code compiled without `-g`:

- Backtrace (`dbx where` command)
- Calling a function (but without parameter checking)
- Checking global variables

Note, however, that `dbx` cannot display source code unless the code was compiled with the `-g` option. This restriction also applies to code that has had `strip -x` applied to it.

Shared Libraries Require the `-g` Option for Full `dbx` Support

For full support, a shared library must also be compiled with the `-g` option. If you build a program with shared library modules that were not compiled with the `-g` option, you can still debug the program. However, full `dbx` support is not possible because the information was not generated for those library modules.

Completely Stripped Programs

The dbx tool can debug programs that have been completely stripped. These programs contain some information that can be used to debug your program, but only externally visible functions are available. Some runtime checking works on stripped programs or load objects: memory use checking works, and access checking works with code stripped with `strip -x` but not with code stripped with `strip`.

Quitting Debugging

A dbx session runs from the time you start dbx until you quit dbx; you can debug any number of programs in succession during a dbx session.

To quit a dbx session, type `quit` at the dbx prompt.

```
(dbx) quit
```

When you start dbx and attach it to a running process using the *process_id* option, the process survives and continues when you quit the debugging session. dbx performs an implicit `detach` before quitting the session.

Stopping a Process Execution

You can stop execution of a process at any time by pressing Ctrl+C without leaving dbx.

Detaching a Process From dbx

If you have attached dbx to a process, you can detach the process from dbx without killing it or the dbx session by using the `detach` command.

To detach a process from dbx without killing the process, type:

```
(dbx) detach
```

You can detach a process and leave it in a stopped state while you temporarily apply other `/proc`-based debugging tools that might be blocked when `dbx` has exclusive access. For more information, see "Detaching `dbx` From a Process" on page 95.

For more information on the `detach` command, see "`detach` Command" on page 323.

Killing a Program Without Terminating the Session

The `dbx kill` command terminates debugging of the current process as well as killing the process. However, `kill` preserves the `dbx` session itself leaving `dbx` ready to debug another program.

Killing a program is a good way of eliminating the remains of a program you were debugging without exiting `dbx`.

To kill a program executing in `dbx`, type:

```
(dbx) kill
```

For more information, see "`kill` Command" on page 341.

Saving and Restoring a Debugging Run

The `dbx` tool provides three commands for saving all or part of a debugging run and replaying it later:

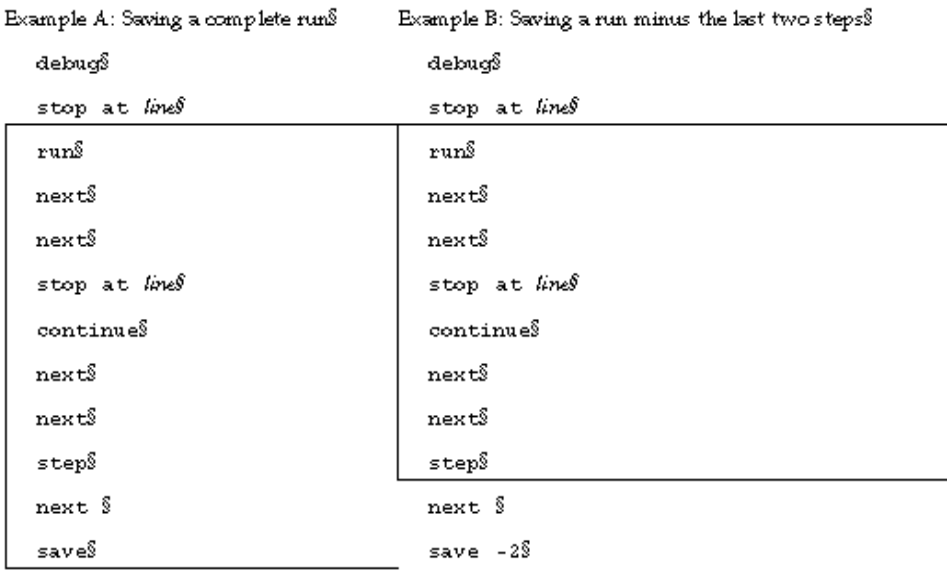
- `save [-number] [filename]`
- `restore [filename]`
- `replay [-number]`

Using the `save` Command

The `save` command saves to a file all debugging commands issued from the last `run`, `rerun`, or `debug` command up to the `save` command. This segment of a debugging session is called a *debugging run*.

The `save` command saves more than the list of debugging commands issued. It saves debugging information associated with the state of the program at the start of the run—breakpoints, display lists, and the like. When you restore a saved run, `dbx` uses the information in the save-file.

You can save part of a debugging run; that is, the whole run minus a specified number of commands from the last one entered. Example A shows a complete saved run. Example B shows the same run saved, minus the last two steps



If you are not sure where you want to end the run you are saving, use the `history` command to see a list of the debugging commands issued since the beginning of the session.

Note – By default, the `save` command writes information to a special save-file. If you want to save a debugging run to a file you can restore later, you can specify a file name with the `save` command. See "Saving a Series of Debugging Runs as Checkpoints" on page 62.

To save an entire debugging run up to the `save` command, type:

```
(dbx) save
```

To save part of a debugging run, use the `save` *number* command, where *number* is the number of commands back from the `save` command that you do not want saved.

```
(dbx) save -number
```

Saving a Series of Debugging Runs as Checkpoints

If you save a debugging run without specifying a file name, `dbx` writes the information to a special save-file. Each time you save, `dbx` overwrites this save-file. However, by giving the `save` command a *filename* argument, you can save a debugging run to a file that you can restore later, even if you have saved other debugging runs since the one saved to *filename*.

Saving a series of runs gives you a set of *checkpoints*, each one starting farther back in the session. You can restore any one of these saved runs, continue, then reset `dbx` to the program location and state saved in an earlier run.

To save a debugging run to a file other than the default save-file:

```
(dbx) save filename
```

Restoring a Saved Run

After saving a run, you can restore the run using the `restore` command. `dbx` uses the information in the save-file. When you restore a run, `dbx` first resets the internal state to what it was at the start of the run, then reissues each of the debugging commands in the saved run.

Note – The `source` command also reissues a set of commands stored in a file, but it does not reset the state of `dbx`; it only reissues the list of commands from the current program location.

Prerequisites for an Exact Restoration of a Saved Run

For exact restoration of a saved debugging run, all the inputs to the run must be exactly the same: arguments to a `run`-type command, manual inputs, and file inputs.

Note – If you save a segment and then issue a `run`, `rerun`, or `debug` command before you do a `restore`, `restore` uses the arguments to the second, post-save `run`, `rerun`, or `debug` command. If those arguments are different, you might not get an exact restoration.

To restore a saved debugging run:, type:

```
(dbx) restore
```

To restore a debugging run saved to a file other than the default save-file:, type:

```
(dbx) restore filename
```

Saving and Restoring Using `replay`

The `replay` command is a combination command, equivalent to issuing a `save -1` followed immediately by a `restore`. The `replay` command takes a negative *number* argument, which it passes to the `save` portion of the command. By default, the value of *-number* is `-1`, so `replay` works as an undo command, restoring the last run up until, but not including, the last command issued.

To replay the current debugging run, minus the last debugging command issued, type:

```
(dbx) replay
```

To replay the current debugging run and stop the run before a specific command, use the `dbx replay` command, where *number* is the number of commands back from the last debugging command.

```
(dbx) replay –number
```

CHAPTER 3

Customizing dbx

This chapter describes the dbx environment variables you can use to customize certain attributes of your debugging environment, and how to use the initialization file, .dbxrc, to preserve your changes and adjustments from session to session.

This chapter is organized into the following sections:

Using the dbx Initialization File

The dbx initialization file stores dbx commands that are executed each time you start dbx. Typically, the file contains commands that customize your debugging environment, but you can place any dbx commands in the file. If you customize dbx from the command line while you are debugging, those settings apply only to the current debugging session.

Note – A .dbxrc file should not contain commands that execute your code. However, you can put such commands in a file, and then use the dbx source command to execute the commands in that file.

During startup, the search order is:

1. Installation directory (unless you specify the -S option to the dbx command) */installation_directory*/lib/dbxrc (the default *installation_directory* is /opt/SUNWspro on Solaris platforms and /opt/sun/sunstudio9 on Linux platform). If your Sun Studio software is not installed in the default *installation_directory*, dbx derives the path to the dbxrc file from the path to the dbx executable.

2. Current directory `./.dbxrc`
3. Home directory `$HOME/.dbxrc`

Creating a `.dbxrc` File

To create a `.dbxrc` file that contains common customizations and aliases, type in the command pane:

```
help .dbxrc>$HOME/.dbxrc
```

You can then customize the resulting file by using your text editor to uncomment the entries you want to have executed.

Initialization File Sample

Here is a sample `.dbxrc` file:

```
dbxenv input_case_sensitive false
catch FPE
```

The first line changes the default setting for the case sensitivity control:

- `dbxenv` is the command used to set `dbx` environment variables. (For a complete list of dbx environment variables, see "Setting dbx Environment Variables" on page 66.)
- `input_case_sensitive` is the dbx environment variable that controls case sensitivity.
- `false` is the setting for `input_case_sensitive`.

The next line is a debugging command, `catch`, which adds a system signal, `FPE`, to the default list of signals to which `dbx` responds, stopping the program.

Setting `dbx` Environment Variables

You can use the `dbxenv` command to set the `dbx` environment variables that customize your `dbx` sessions.

To display the value of a specific variable, type:

```
(dbx) dbxenv variable
```

To show all variables and their values, type:

```
(dbx) dbxenv
```

To set the value of a variable, type:

```
(dbx) dbxenv variable value
```

TABLE 3-1 shows all of the dbx environment variables that you can set:

TABLE 3-1 dbx Environment Variables

dbx **Environment Variable**	**What the Variable Does**
`array_bounds_check on|off`	If set to `on`, dbx checks the array bounds. Default: `on`.
`CLASSPATHX`	Lets you specify to dbx a path for Java class files that are loaded by custom class loaders
`core_lo_pathmap on|off`	Controls whether dbx uses pathmap settings to locate the correct libraries for a "mismatched" core file. Default: `off`.
`disassembler_version autodetect|v8|v9|v9vis`	SPARC platform: Sets the version of dbx's built-in disassembler for SPARC V8, V9, or V9 with the Visual Instruction set. Default is `autodetect`, which sets the mode dynamically depending on the type of the machine `a.out` is running on. IA platforms: The valid choice is `autodetect`.
`fix_verbose on|off`	Governs the printing of compilation line during a `fix`. Default: `off`
`follow_fork_inherit on|off`	When following a child, inherits or does not inherit breakpoints. Default: `off`

TABLE 3-1 dbx Environment Variables *(Continued)*

dbx **Environment Variable**	**What the Variable Does**
`follow_fork_mode parent|child|both|ask`	Determines which process is followed after a fork; that is, when the current process executes a `fork`, `vfork`, or `fork1`. If set to `parent`, the process follows the parent. If set to `child`, it follows the child. If set to `both`, it follows the child, but the parent process remains active. If set to `ask`, you are asked which process to follow whenever a fork is detected. Default: `parent`.
`follow_fork_mode_inner unset| parent|child|both`	Of relevance after a fork has been detected if `follow_fork_mode` was set to `ask`, and you chose `stop`. By setting this variable, you need not use `cont -follow`.
`input_case_sensitive autodetect| true|false`	If set to `autodetect`, `dbx` automatically selects case sensitivity based on the language of the file: `false` for Fortran files, otherwise `true`. If `true`, case matters in variable and function names; otherwise, case is not significant. Default: `autodetect`.
`JAVASRCPATH`	Specifies the directories in which `dbx` should look for Java source files.
`jdbx_mode java| jni| native`	Stores the current `dbx` mode. It can have the following settings: `java`, `jni`, or `native`.
`jvm_invocation`	The `jvm_invocation` environment variable lets you customize the way the JVM™ software is started. (The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java™ platform.) For more information, see "Customizing Startup of the JVM Software" on page 236.
`language_mode autodetect|main|c| c++|fortran|fortran90`	Governs the language used for parsing and evaluating expressions. • `autodetect` sets the expression language to the language of the current file. Useful if debugging programs with mixed languages (default). • `main` sets the expression language to the language of the main routine in the program. Useful if debugging homogeneous programs. • `c`, `c++`, `c++`, `fortran`, or `fortran90` sets the expression language to the selected language.
`mt_scalable on|off`	When enabled, `dbx` will be more conservative in its resource usage and will be able to debug processes with upwards of 300 LWPs. The down side is significant slowdown. Default: `off`.

TABLE 3-1 dbx Environment Variables *(Continued)*

dbx **Environment Variable**	**What the Variable Does**
`output_auto_flush on|off`	Automatically calls `fflush()` after each `call`. Default: `on`
`output_base 8|10|16|automatic`	Default base for printing integer constants. Default: `automatic` (pointers in hexadecimal characters, all else in decimal).
`output_class_prefix on | off`	Used to cause a class member to be prefixed with classname(s) when its value or declaration is printed. If set to `on`, it causes the class member to be prefixed. Default: `on`.
`output_dynamic_type on|off`	When set to `on`, `-d` is the default for printing, displaying, and inspecting. Default: `off`.
`output_inherited_members on|off`	When set to `on`, `-r` is the default for printing, displaying, and inspecting. Default: `off`.
`output_list_size` *num*	Governs the default number of lines to print in the `list` command. Default: 10.
`output_log_file_name` *filename*	Name of the command logfile. Default: `/tmp/dbx.log.`*uniqueID*
`output_max_string_length` *number*	Sets *number* of characters printed for `char *s`. Default: 512.
`output_pretty_print on|off`	Sets `-p` as the default for printing, displaying, and inspecting. Default: `off`.
`output_short_file_name on|off`	Displays short path names for files. Default: `on`.
`overload_function on|off`	For C++, if set to `on`, does automatic function overload resolution. Default: `on`.
`overload_operator on|off`	For C++, if set to `on`, does automatic operator overload resolution. Default: `on`.
`pop_auto_destruct on|off`	If set to `on`, automatically calls appropriate destructors for locals when `pop`ping a frame. Default: `on`.
`proc_exclusive_attach on|off`	If set to `on`, keeps `dbx` from attaching to a process if another tool is already attached. Warning: be aware that if more than one tool attaches to a process and tries to control it chaos ensues. Default: `on`.
`rtc_auto_continue on|off`	Logs errors to `rtc_error_log_file_name` and continue. Default: `off`.
`rtc_auto_suppress on|off`	If set to `on`, an RTC error at a given location is reported only once. Default: `off`.

TABLE 3-1 dbx Environment Variables *(Continued)*

dbx **Environment Variable**	**What the Variable Does**
`rtc_biu_at_exit on|off|verbose`	Used when memory use checking is on explicitly or via `check -all`. If the value is `on`, a non-verbose memory use (blocks in use) report is produced at program exit. If the value is `verbose`, a verbose memory use report is produced at program exit. The value `off` causes no output. Default: `on`.
`rtc_error_limit` *number*	*Number* of RTC errors to be reported. Default: 1000.
`rtc_error_log_file_name` *filename*	Name of file to which RTC errors are logged if `rtc_auto_continue` is set. Default: `/tmp/dbx.errlog.`*uniqueID*
`rtc_error_stack on|off`	If set to `on`, stack traces show frames corresponding to RTC internal mechanisms. Default: `off`.
`rtc_inherit on|off`	If set to `on`, enables runtime checking on child processes that are executed from the debugged program and causes `LD_PRELOAD` to be inherited. Default: `off`.
`rtc_mel_at_exit on|off|verbose`	Used when memory leak checking is `on`. If the value is `on`, a non-verbose memory leak report is produced at program exit. If the value is `verbose`, a verbose memory leak report is produced at program exit. The value `off` causes no output. Default: `on`.
`run_autostart on|off`	If set to `on` with no active program, `step`, `next`, `stepi`, and `nexti` implicitly run the program and stop at the language-dependent `main` routine. If set to `on`, `cont` implies `run` when necessary. Default: `off`.
`run_io stdio|pty`	Governs whether the user program's input/output is redirected to `dbx`'s `stdio` or a specific pty. The pty is provided by `run_pty`. Default: `stdio`.
`run_pty` *ptyname*	Sets the name of the pty to use when `run_io` is set to `pty`. Ptys are used by graphical user interface wrappers.
`run_quick on|off`	If set to `on`, no symbolic information is loaded. The symbolic information can be loaded on demand using `prog -readsysms`. Until then, `dbx` behaves as if the program being debugged is stripped. Default: `off`.

TABLE 3-1 dbx Environment Variables *(Continued)*

dbx **Environment Variable**	**What the Variable Does**
`run_savetty on | off`	Multiplexes tty settings, process group, and keyboard settings (if `-kbd` was used on the command line) between `dbx` and the program being debugged. Useful when debugging editors and shells. Set to `on` if `dbx` gets `SIGTTIN` or `SIGTTOU` and pops back into the shell. Set to `off` to gain a slight speed advantage. The setting is irrelevant if `dbx` is attached to the program being debugged or is running under the dbx Debugger. Default: `on`.
`run_setpgrp on | off`	If set to `on`, when a program is run, `setpgrp(2)` is called right after the fork. Default: `off`.
`scope_global_enums on |off`	If set to `on`, enumerators are put in global scope and not in file scope. Set before debugging information is processed (`~/.dbxrc`). Default: `off`.
`scope_look_aside on | off`	If set to `on`, finds file static symbols, in scopes other than the current scope. Default: `on`.
`session_log_file_name` *filename*	Name of the file where `dbx` logs all commands and their output. Output is appended to the file. Default: "" (no session logging).
`stack_find_source on|off`	When set to `on`, `dbx` attempts to find and automatically make active the first stack frame with source when the program being debugged comes to a stop in a function that is not compiled with `-g`. Default: `on`.
`stack_max_size` *number*	Sets the default size for the `where` command. Default: 100.
`stack_verbose on|off`	Governs the printing of arguments and line information in `where`. Default: `on`.
`step_events on|off`	When set to `on`, allows breakpoints while using `step` and `next` commands to step through code. Default: `off`.
`step_granularity statement | line`	Controls granularity of source line stepping. When set to `statement` the following code: `a(); b();` takes the two `next` commands to execute. When set to `line` a single `next` command executes the code. The granularity of line is particularly useful when dealing with multiline macros. Default: `statement`.

TABLE 3-1 dbx Environment Variables *(Continued)*

dbx **Environment Variable**	**What the Variable Does**
`suppress_startup_message` n*umber*	Sets the release level below which the startup message is not printed. Default: 3.01.
`symbol_info_compression on|off`	When set to `on`, reads debugging information for each `include` file only once. Default: `on`.
`trace_speed` *number*	Sets the speed of tracing execution. Value is the number of seconds to pause between steps. Default: 0.50.

The dbx Environment Variables and the Korn Shell

Each `dbx` environment variable is also accessible as a `ksh` variable. The name of the `ksh` variable is derived from the `dbx` environment variable by prefixing it with `DBX_`. For example `dbxenv stack_verbose` and `echo $DBX_stack_verbose` yield the same output. You can assign the value of the variable directly or with the `dbxenv` command.

CHAPTER 4

Viewing and Navigating To Code

Each time the program you are debugging stops, `dbx` prints the source line associated with the *stop location*. At each program stop, `dbx` resets the value of the *current function* to the function in which the program is stopped. Before the program starts running and when it is stopped, you can move to, or navigate through, functions and files elsewhere in the program.

This chapter describes how `dbx` navigates to code and locates functions and symbols. It also covers how to use commands to navigate to code or look up declarations for identifiers, types, and classes.

This chapter is organized into the following sections

Navigating To Code

When a program is stopped, you can navigate to code elsewhere in the program. You can navigate to any function or file that is part of the program. Navigating sets the current scope (see "Program Scope" on page 76). It is useful for determining when and at what source line you want to set a `stop at` breakpoint.

Navigating To a File

You can navigate to any file `dbx` recognizes as part of the program (even if a module or file was not compiled with the `-g` option). To navigate to a file:

```
(dbx) file filename
```

Using the `file` command without arguments echoes the file name of the file you are currently navigating.

```
(dbx) file
```

`dbx` displays the file from its first line unless you specify a line number.

```
(dbx) file filename ; list line_number
```

For information on setting a stop at breakpoint at a line of source code, see "Setting a `stop` Breakpoint at a Line of Source Code" on page 102.

Navigating To Functions

You can use the `func` command to navigate to a function. To navigate to a function, type the command `func` followed by the function name. For example:

```
(dbx) func adjust_speed
```

The `func` command by itself echoes the current function.

For more information, see "`func` Command" on page 333

Selecting From a List of C++ Ambiguous Function Names

If you try to navigate to a C++ member function with an ambiguous name or an overloaded function name, a list is displayed, showing all functions with the overloaded name. Type the number of the function you want to navigate. If you know which specific class a function belongs to, you can type the class name and function name. For example:

```
(dbx) func block::block
```

Choosing Among Multiple Occurrences

If multiple symbols are accessible from the same scope level, `dbx` prints a message reporting the ambiguity.

```
(dbx) func main
(dbx) which C::foo
More than one identifier 'foo'.
Select one of the following:
 0) Cancel
 1) 'a.out't.cc'C::foo(int)
 2) 'a.out't.cc'C::foo()
>1
'a.out't.cc'C::foo(int)
```

In the context of the `which` command, choosing from the list of occurrences does not affect the state of `dbx` or the program. Whichever occurrence you choose, `dbx` echoes the name.

Printing a Source Listing

Use the `list` command to print the source listing for a file or function. Once you navigate through a file, `list` prints *number* lines from the top. Once you navigate through a function, `list` prints its lines.

For detailed information on the `list` command, see "`list` Command" on page 343.

Walking the Call Stack to Navigate To Code

Another way to navigate to code when a live process exists is to "walk the call stack," using the stack commands to view functions currently on the call stack, which represent all currently active routines. Walking the stack causes the current function and file to change each time you display a stack function. The stop location is considered to be at the "bottom" of the stack, so to move away from it, use the `up` command, that is, move toward the `main` or `begin` function. Use the `down` command to move toward the current frame.

For more information on walking the call stack, see "Walking the Stack and Returning Home" on page 118.

Types of Program Locations

`dbx` uses three global locations to track the parts of the program you are inspecting:

- The current address, which is used and updated by the `dis` command (see "`dis` Command" on page 324) and the `examine` command (see "`examine` Command" on page 328).
- The current source code line, which is used and updated by the `list` command (see "`list` Command" on page 343). This line number is reset by some commands that alter the visiting scope (see "Changing the Visiting Scope" on page 78).
- The current visiting scope, which is a compound variable described in the "Visiting Scope" on page 77. The visiting scope is used during expression evaluation. It is updated by the `line` command, the `func` command, the `file` command, the `list` *func* command and the `list` *file* command.

Program Scope

A scope is a subset of the program defined in terms of the visibility of a variable or function. A symbol is said to be "in scope" if its name is visible at a given point of execution. In C, functions may have global or file-static scope; variables may have global, file-static, function, or block scope.

Variables That Reflect the Current Scope

The following variables always reflect the current program counter of the current thread or LWP, and are not affected by the various commands that change the visiting scope:

`$scope`	Scope of the current program counter
`$lineno`	Current line number
`$func`	Current function
`$class`	Class to which `$func` belongs
`$file`	Current source file
`$loadobj`	Current loadobject

Visiting Scope

When you inspect various elements of your program with `dbx`, you modify the visiting scope. `dbx` uses the visiting scope during expression evaluation for purposes such as resolving ambiguous symbols. For example, if you type the following command, `dbx` uses the visiting scope to determine which `i` to print.

```
(dbx) print i
```

Each thread or LWP has its own visiting scope. When you switch between threads, each thread remembers its visiting scope.

Components of the Visiting Scope

Some of the components of the visiting scope are visible in the following predefined `ksh` variables:

$vscope	Language scope
`$vloadobj`	Current visiting loadobject
`$vfile`	Current visiting source file
`$vfunc`	Current visiting function
`$vlineno`	Current visiting line number
`$vclass`	C++ class

All of the components of the current visiting scope stay compatible with one another. For example, if you visit a file that contains no functions, the current visiting source file is updated to the new file name and the current visiting function is updated to NULL.

Changing the Visiting Scope

The following commands are the most common ways of changing the visiting scope:

- `func`
- `file`
- `up`
- `down`
- `frame`
- `list` *procedure*

The `debug` command and the `attach` command set the initial visiting scope.

When you hit a breakpoint, `dbx` sets the visiting scope to the current location. If the `stack_find_source` environment variable (see "Setting dbx Environment Variables" on page 66) is set to `ON`, `dbx` attempts to find and make active a stack frame that has source code.

When you use the `up` command (see "up Command" on page 395), the `down` command ("down Command" on page 326), the `frame` *number* command (see "frame Command" on page 333), or the `pop` command (see "pop Command" on page 357)to change the current stack frame, `dbx` sets the visiting scope according to the program counter from the new stack frame.

The line number location used by the `list` command (see "list Command" on page 343) changes the visiting scope only if you use the `list` *function* or `list` *file* command. When the visiting scope is set, the line number location for the `list` command is set to the first line number of the visiting scope. When you subsequently use the `list` command, the current line number location for the `list` command is updated, but as long as you are listing lines in the current file, the visiting scope does not change. For example, if you type the following, `dbx` lists the start of the source for `my_func`, and changes the visiting scope to `my_func`.

```
(dbx) list my_func
```

If you type the following, `dbx` lists line 127 in the current source file, and does not change the visiting scope.

```
(dbx) list 127
```

When you use the `file` command or the `func` command to change the current file or the current function, the visiting scope is updated accordingly.

Qualifying Symbols With Scope Resolution Operators

When using the `func` or `file` command, you might need to use *scope resolution operators* to qualify the names of the functions that you give as targets.

`dbx` provides three scope resolution operators with which to qualify symbols: the backquote operator (`` ` ``), the C++ double colon operator (`::`), and the block local operator (:*lineno*). You use them separately or, in some cases, together.

In addition to qualifying file and function names when navigating through code, symbol name qualifying is also necessary for printing and displaying out-of-scope variables and expressions, and for displaying type and class declarations (using the `whatis` command). The symbol qualifying rules are the same in all cases; this section covers the rules for all types of symbol name qualifying.

Backquote Operator

Use the backquote character (`` ` ``) to find a variable or function of global scope:

```
(dbx) print `item
```

A program can use the same function name in two different files (or compilation modules). In this case, you must also qualify the function name to `dbx` so that it registers which function you will navigate. To qualify a function name with respect to its file name, use the general purpose backquote (`` ` ``) scope resolution operator.

```
(dbx) func `file_name`function_name
```

C++ Double Colon Scope Resolution Operator

Use the double colon operator (`::`) to qualify a C++ member function, a top level function, or a variable with global scope with:

- An overloaded name (same name used with different argument types)
- An ambiguous name (same name used in different classes)

You might want to qualify an overloaded function name. If you do not qualify it, `dbx` displays an overload list so you can choose which function you will navigate. If you know the function class name, you can use it with the double colon scope resolution operator to qualify the name.

```
(dbx) func class::function_name (args)
```

For example, if `hand` is the class name and `draw` is the function name, type:

```
(dbx) func hand::draw
```

Block Local Operator

The block local operator (:*line_number*) allows you to refer specifically to a variable in a nested block. You might want to do so if you have a local variable shadowing a parameter or member name, or if you have several blocks, each with its own version of a local variable. The *line_number* is the number of the first line of code within the block for the variable of interest. When `dbx` qualifies a local variable with the block local operator, `dbx` uses the line number of the first block of code, but you can use any line number within the scope in `dbx` expressions.

In the following example, the block local operator (`:230`) is combined with the backquote operator.

```
(dbx) stop in `animate.o`change_glyph:230`item
```

The following example shows how dbx evaluates a variable name qualified with the block local operator when there are multiple occurrences in a function.

```
(dbx) list 1,$
    1   #include <stddef.h>
    2
    3   int main(int argc, char** argv) {
    4
    5   int i=1;
    6
    7       {
    8            int i=2;
    9            {
   10                   int j=4;
   11                   int i=3;
   12                   printf("hello");
   13            }
   14            printf("world\n");
   15       }
   16       printf("hi\n");
   17   }
   18
(dbx) whereis i
variable:`a.out`t.c`main`i
variable:`a.out`t.c`main:8`i
variable:`a.out`t.c`main:10`i
(dbx) stop at 12 ; run
...
(dbx) print i
i = 3
(dbx) which i
`a.out`t.c`main:10`i
(dbx) print `main:7`i
`a.out`t.c`main`i = 1
(dbx) print `main:8`i
`a.out`t.c`main:8`i = 2
(dbx) print `main:10`i
`a.out`t.c`main:10`i = 3
(dbx) print `main:14`i
`a.out`t.c`main:8`i = 2
(dbx) print `main:15`i
`a.out`t.c`main`i = 1
```

Linker Names

dbx provides a special syntax for looking up symbols by their linker names (mangled names in C++). Prefix the symbol name with a # (pound sign) character (use the ksh escape character \ (backslash) before any $ (dollar sign) characters), as in these examples:

```
(dbx) stop in #.mul
(dbx) whatis #\$FEcopyPc
(dbx) print `foo.c`#staticvar
```

Locating Symbols

In a program, the same name might refer to different types of program entities and occur in many scopes. The dbx whereis command lists the fully qualified name—hence, the location—of all symbols of that name. The dbx which command tells you which occurrence of a symbol dbx would use if you give that name in an expression (see "which Command" on page 401).

Printing a List of Occurrences of a Symbol

To print a list of all the occurrences of a specified symbol, use whereis *symbol*, where *symbol* can be any user-defined identifier. For example:

```
(dbx) whereis table
forward: `Blocks`block_draw.cc`table
function: `Blocks`block.cc`table::table(char*, int, int, const
point&)
class: `Blocks`block.cc`table
class: `Blocks`main.cc`table
variable:       `libc.so.1`hsearch.c`table
```

The output includes the name of the loadable object(s) where the program defines *symbol*, as well as the kind of entity each object is: class, function, or variable.

Because information from the dbx symbol table is read in as it is needed, the whereis command registers only occurrences of a symbol that are already loaded. As a debugging session gets longer, the list of occurrences can grow (see "Debugging Information in Object files and Executables" on page 89).

For more information, see "whereis Command" on page 401.

Determining Which Symbol dbx Uses

The which command tells you which symbol with a given name dbx uses if you specify that name (without fully qualifying it) in an expression. For example:

```
(dbx) func
wedge::wedge(char*, int, int, const point&, load_bearing_block*)
(dbx) which draw
`block_draw.cc`wedge::draw(unsigned long)
```

If a specified symbol name is not in a local scope, the which command searches for the first occurrence of the symbol along the *scope resolution search path*. If which finds the name, it reports the fully qualified name.

If at any place along the search path, the search finds multiple occurrences of *symbol* at the same scope level, dbx prints a message in the command pane reporting the ambiguity.

```
(dbx) which fid
More than one identifier 'fid'.
Select one of the following:
 0) Cancel
 1) `example`file1.c`fid
 2) `example`file2.c`fid
```

dbx shows the overload display, listing the ambiguous symbols names. In the context of the which command, choosing from the list of occurrences does not affect the state of dbx or the program. Whichever occurrence you choose, dbx echoes the name.

The which command gives you a preview of what happens if you make *symbol* (in this example, block) an argument of a command that must operate on *symbol* (for example, a print command). In the case of ambiguous names, the overload display list indicates that dbx does not yet register which occurrence of two or more names it uses. dbx lists the possibilities and waits for you to choose one. For more information on the which command, see "which Command" on page 401.

Scope Resolution Search Path

When you issue a debugging command that contains an expression, the symbols in the expression are looked up in the following order. dbx resolves the symbols as the compiler would at the current visiting scope.

1. Within the scope of the current function using the current visiting scope (see "Visiting Scope" on page 77). If the program is stopped in a nested block, `dbx` searches within that block, then in the scope of all enclosing blocks.
2. For C++ only: class members of the current function's class and its base class.
3. For C++ only: the current name space.
4. The parameters of the current function.
5. The immediately enclosing module, generally, the file containing the current function.
6. Symbols that were made private to this shared library or executable. These symbols can be created using linker scoping.
7. Global symbols for the main program, and then for shared libraries.
8. If none of the above searches are successful, `dbx` assumes you are referencing a private, or file static, variable or function in another file. `dbx` optionally searches for a file static symbol in every compilation unit depending on the value of the `dbxenv` setting `scope_look_aside`.

`dbx` uses whichever occurrence of the symbol it first finds along this search path. If `dbx` cannot find the symbol, it reports an error.

Relaxing the Scope Lookup Rules

To relax the scope lookup rules for static symbols and C++ member functions, set the `dbx` environment variable `scope_look_aside` to on:

```
dbxenv scope_look_aside on
```

or use the "double backquote" prefix:

```
stop in ``func4          func4 may be static and not in scope
```

If the `dbx` environment variable `scope_look_aside` is set to on, `dbx` looks for:

- Static variables defined in other files if not found in current scope. Files from libraries in `/usr/lib` are not searched.
- C++ member functions without class qualification

- Instantiations of C++ inline member functions in other files if a member function is not instantiated in current file.

The `which` command tells you which symbol `dbx` would choose. In the case of ambiguous names, the overload display list indicates that `dbx` has not yet determined which occurrence of two or more names it would use. `dbx` lists the possibilities and waits for you to choose one.

For more information, see "`func` Command" on page 333.

Viewing Variables, Members, Types, and Classes

The `whatis` command prints the declarations or definitions of identifiers, structs, types and C++ classes, or the type of an expression. The identifiers you can look up include variables, functions, fields, arrays, and enumeration constants.

For more information, see "`whatis` Command" on page 396.

Looking Up Definitions of Variables, Members, and Functions

To print out the declaration of an identifier, type:

```
(dbx) whatis identifier
```

Qualify the identifier name with file and function information as needed.

For C++ programs, `whatis` *identifier* lists function template instantiations. Template definitions are displayed with `whatis -t` *identifier*. See "Looking Up Definitions of Types and Classes" on page 87.

For Java programs, `whatis` *identifier*, lists the declaration of a class, a method in the current class, a local variable in the current frame, or a field in the current class

To print out the member function, type:

```
(dbx) whatis block::draw
void block::draw(unsigned long pw);
(dbx) whatis table::draw
void table::draw(unsigned long pw);
(dbx) whatis block::pos
class point *block::pos();
(dbx) whatis table::pos
class point *block::pos();
```

To print out the data member, type:

```
(dbx) whatis block::movable
int movable;
```

On a variable, `whatis` tells you the variable's type.

```
(dbx) whatis the_table
class table *the_table;
```

On a field, `whatis` gives the field's type.

```
(dbx) whatis the_table->draw
void table::draw(unsigned long pw);
```

When you are stopped in a member function, you can look up the `this` pointer.

```
(dbx) stop in brick::draw
(dbx) cont
(dbx) where 1
brick::draw(this = 0x48870, pw = 374752), line 124 in
     "block_draw.cc"
(dbx) whatis this
class brick *this;
```

Looking Up Definitions of Types and Classes

The `-t` option of the `whatis` command displays the definition of a type. For C++, the list displayed by `whatis -t` includes template definitions and class template instantiations.

To print the declaration of a type or C++ class, type:

```
(dbx) whatis -t type_or_class_name
```

To see inherited members, the `whatis` command takes an `-r` option (for recursive) that displays the declaration of a specified class together with the members it inherits from base classes.

```
(dbx) whatis -t -r class_name
```

The output from a `whatis -r` query may be long, depending on the class hierarchy and the size of the classes. The output begins with the list of members inherited from the most ancestral class. The inserted comment lines separate the list of members into their respective parent classes.

Here are two examples, using the class `table`, a child class of the parent class `load_bearing_block`, which is, in turn, a child class of `block`.

Without `-r`, `whatis` reports the members declared in class `table`:

```
(dbx) whatis -t class table
class table : public load_bearing_block {
public:
    table::table(char *name, int w, int h, const class point
&pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
```

Here are results when `whatis -r` is used on a child class to see members it inherits:

```
(dbx) whatis -t -r class table
class table : public load_bearing_block {
public:
  /* from base class table::load_bearing_block::block */
  block::block();
  block::block(char *name, int w, int h, const class point &pos,
class load_bearing_block *blk);
```

```
    virtual char *block::type();
    char *block::name();
    int block::is_movable();
//  deleted several members from example protected:
    char *nm;
    int movable;
    int width;
    int height;
    class point  position;
    class load_bearing_block *supported_by;
    Panel_item panel_item;
    /* from base class table::load_bearing_block */
public:
    load_bearing_block::load_bearing_block();
    load_bearing_block::load_bearing_block(char *name, int w, int
h,const class point &pos, class load_bearing_block *blk);
    virtual int load_bearing_block::is_load_bearing();
    virtual class list *load_bearing_block::supported_blocks();
    void load_bearing_block::add_supported_block(class block &b);
    void load_bearing_block::remove_supported_block(class block
&b);
    virtual void load_bearing_block::print_supported_blocks();
    virtual void load_bearing_block::clear_top();
    virtual void load_bearing_block::put_on(class block &object);
    class point load_bearing_block::get_space(class block
&object);
    class point load_bearing_block::find_space(class block
&object);
    class point load_bearing_block::make_space(class block
&object);
protected:
    class list *support_for;
    /* from class table */
public:
    table::table(char *name, int w, int h, const class point &pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
```

Debugging Information in Object files and Executables

Generally, you want to compile your source files with the `-g` option to make your program more debuggable. The `-g` option causes the compilers to record debugging information (in stabs or Dwarf format) into the object files along with the code and data for the program.

`dbx` parses and loads debugging information for each object file (module) on demand, when the information is needed. You can ask `dbx` to load debug information for any specific module, or for all modules, by using the `module` command. See also "Finding Source and Object Files" on page 91.

Object File Loading

When the object (`.o`) files are linked together, the linker can optionally store only summary information into the resulting loadobject. This summary information can be used by `dbx` at runtime to load the rest of the debug information from the object files themselves instead of from the executable file. The resulting executable has a smaller disk-footprint, but requires that the object files be available when `dbx` runs.

You can override this requirement by compiling object files with the `-xs` option to cause all the debugging information for those object files to be put into the executable at link time.

If you create archive libraries (`.a` files) with your object files, and use the archive libraries in your program, then `dbx` extracts the object files from the archive library as needed. The original object files are not needed at that point.

The only drawback to putting all the debugging information into the executable file is using additional disk space. The program does not run more slowly, because the debugging information is not loaded into the process image at run time.

The default behavior when using stabs (the default format for debugging information) is for the compiler to put only summary information into the executable.

The DWARF format doesn't yet support object file loading.

Note – The DWARF format is significantly more compact than recording the same information in stabs format. However, because all the information is copied into the executable, DWARF information can appear to be larger than stabs information.

Listing Debugging Information for Modules

The `module` command and its options help you to keep track of program modules during the course of a debugging session. Use the `module` command to read in debugging information for one or all modules. Normally, `dbx` automatically and "lazily" reads in debugging information for modules as needed.

To read in debugging information for a module *name*, type:

```
(dbx) module [-f] [-q] name
```

To read in debugging information for all modules, type:

```
(dbx) module [-f] [-q] -a
```

where:

`-a`	Specifies all modules.
`-f`	Forces reading of debugging information, even if the file is newer than the executable.
`-q`	Specifies quiet mode.
`-v`	Specifies verbose mode, which prints language, file names, and so on. This is the default.

To print the name of the current module, type:

```
(dbx) module
```

Listing Modules

The `modules` command helps you keep track of modules by listing module names.

To list the names of modules containing debugging information that have already been read into `dbx`, type:

```
(dbx) modules [-v] -read
```

To list names of all program modules (whether or not they contain debugging information), type:

```
(dbx) modules [-v]
```

To list all program modules that contain debugging information, type:

```
(dbx) modules [-v] -debug
```

where:

-v Specifies verbose mode, which prints language, file names, and so on.

Finding Source and Object Files

`dbx` must know the location of the source and object code files associated with a program. The default directory for the object files is the directory the files were in when the program was last linked. The default directory for the source files is the one they were in when last compiled. If you move the source or object files, or copy them to a new location, you must either relink the program, change to the new location before debugging, or use the `pathmap` command.

`dbx` sometimes uses object files to load additional debugging information. Source files are used when `dbx` displays source code.

If you have moved the source files or object files since you compiled and linked the program, you can add their new location to the search path. The `pathmap` command creates a mapping from your current view of the file system to the name in the executable image. The mapping is applied to source paths and object file paths.

To establish a new mapping from the directory *from* to the directory *to*:

```
(dbx) pathmap [-c] from to
```

If `-c` is used, the mapping is applied to the current working directory as well.

The `pathmap` command is also useful for dealing with automounted and explicit NFS mounted file systems with different base paths on differing hosts. Use `-c` when you try to correct problems due to the automounter because current working directories are inaccurate on automounted file systems.

The mapping of `/tmp_mnt` to `/` exists by default.

For more information, see "`pathmap` Command" on page 356.

CHAPTER 5

Controlling Program Execution

The commands used for running, stepping, and continuing (`run`, `rerun`, `next`, `step`, and `cont`) are called *process control* commands. Used together with the event management commands described in Appendix B, you can control the run-time behavior of a program as it executes under `dbx`.

This chapter is organized into the following sections:

Running a Program

When you first load a program into `dbx`, `dbx` navigates to the program's "main" block (`main` for C, C++, and Fortran 90; `MAIN` for Fortran 77; the main class for Java code). `dbx` waits for you to issue further commands; you can navigate through code or use event management commands.

You can set breakpoints in the program before running it.

Note – When debugging an application that is a mixture of Java™ code and C JNI (Java™ Native Interface) code or C++ JNI code, you may want to set breakpoints in code that has not yet been loaded. For information on setting breakpoints on such code, see "Setting Breakpoints on Code That Has Not Yet Been Loaded by the JVM Software" on page 235.

Use the `run` command to start program execution.

To run a program in dbx without arguments, type:

```
(dbx) run
```

You can optionally add command-line arguments and redirection of input and output.

```
(dbx) run [arguments][ < input_file] [ > output_file]
```

Note – You cannot redirect the input and output of a Java application.

Output from the run command overwrites an existing file even if you have set noclobber for the shell in which you are running dbx.

The run command without arguments restarts the program using the previous arguments and redirection. For more information, see "run Command" on page 366. The rerun command restarts the program and clears the original arguments and redirection. For more information, see "rerun Command" on page 364.

Attaching dbx to a Running Process

You might need to debug a program that is already running. You would attach to a running process if:

- You wanted to debug a running server, and you did not want to stop or kill it.
- You wanted to debug a running program that has a graphical user interface, and you didn't want to restart it.
- Your program was looping indefinitely, and you want to debug it without killing it.

You can attach dbx to a running program by using the program's *process_id* number as an argument to the dbx debug command.

Once you have debugged the program, you can then use the detach command to take the program out from the control of dbx without terminating the process.

If you quit dbx after attaching it to a running process, dbx implicitly detaches before terminating.

To attach dbx to a program that is running independently of dbx, you can use either the attach command or the debug command.

To attach dbx to a process that is already running, type:

```
(dbx) debug program_name process_id
```

or

```
(dbx) attach process_id
```

You can substitute a – (dash) for the *program_name*; dbx automatically finds the program associated with the process ID and loads it.

For more information, see "debug Command" on page 320 and "attach Command" on page 300.

If dbx is not running, start dbx by typing:

```
% dbx program_name process_id
```

After you have attached dbx to a program, the program stops executing. You can examine it as you would any program loaded into dbx. You can use any event management or process control command to debug it.

When you attach dbx to a new process while you are debugging an existing process, the following occurs:

- If you started the process you are currently debugging with a run command, then dbx terminates that process before attaching to the new process.
- If you started debugging the current process with an attach command or by specifying the process ID on the command line then dbx detaches from the current process before attaching to the new process.

You can use runtime checking on an attached process with certain exceptions. See "Using Runtime Checking on an Attached Process" on page 159.

Detaching dbx From a Process

When you have finished debugging the program, use the detach command to detach dbx from the program. The program then resumes running independently of dbx, unless you specify the -stop option when you detach it.

To detach a process from running under the control of dbx:

```
(dbx) detach
```

You can detach a process and leave it in a stopped state while you temporarily apply other /proc-based debugging tools that might be blocked when dbx has exclusive access. For example:

```
(dbx) oproc=$proc            # Remember the old process ID
(dbx) detach -stop
(dbx) /usr/proc/bin/pwdx $oproc
(dbx) attach $oproc
```

For more information, see "detach Command" on page 323.

Stepping Through a Program

dbx supports two basic single-step commands: next and step, plus two variants of step, called step up and step to. Both the next command and the step command let the program execute one source line before stopping again.

If the line executed contains a function call, the next command allows the call to be executed and stops at the following line ("steps over" the call). The step command stops at the first line in a called function ("steps into" the call).

The step up command returns the program to the caller function after you have stepped into a function.

The step to command attempts to step into a specified function in the current source line, or if no function is specified, into the last function called as determined by the assembly code for the current source line. The function call may not occur due to a conditional branch, or there may be no function called in the current source line. In these cases, step to steps over the current source line.

Single Stepping

To single step a specified number of lines of code, use the `dbx` commands `next` or `step` followed by the number of lines `[`*n*`]` of code you want executed.

```
(dbx) next n
```

or

```
(dbx) step n
```

The `step_granularity` environment variable determines the unit by which the `step` command and `next` command step through your code (see "Setting dbx Environment Variables" on page 66). The unit can be either statement or line.

For more information on the commands, see "next Command" on page 353 and "step Command" on page 372.

Continuing Execution of a Program

To continue a program, use the `cont` command.

```
(dbx) cont
```

The `cont` command has a variant, `cont at` *line_number*, which lets you specify a line other than the current program location line at which to resume program execution. This allows you to skip over one or more lines of code that you know are causing problems, without having to recompile.

To continue a program at a specified line, type:

```
(dbx) cont at 124
```

The line number is evaluated relative to the file in which the program is stopped; the line number given must be within the scope of the current function.

Using `cont at` *line_number* with `assign`, you can avoid executing a line of code that contains a call to a function that might be incorrectly computing the value of some variable.

To resume program execution at a specific line:

1. **Use `assign` to give the variable a correct value.**
2. **Use `cont at` *line_number* to skip the line that contains the function call that would have computed the value incorrectly.**

   Assume that a program is stopped at line 123. Line 123 calls a function, `how_fast()`, that computes incorrectly a variable, `speed`. You know what the value of speed should be, so you assign a value to `speed`. Then you continue program execution at line 124, skipping the call to `how_fast()`.

```
(dbx) assign speed = 180; cont at 124;
```

   For more information, see "`cont` Command" on page 316.

   If you use the `cont` command with a `when` breakpoint command, the program skips the call to `how_fast()` each time the program attempts to execute line 123.

```
(dbx) when at 123 { assign speed = 180; cont at 124;}
```

   For more information on the `when` command, see:

- "Setting a `stop` Breakpoint at a Line of Source Code" on page 102
- "Setting Breakpoints in Member Functions of Different Classes" on page 104
- "Setting Breakpoints in Member Functions of the Same Class" on page 105
- "Setting Multiple Breakpoints in Nonmember Functions" on page 105
- "`when` Command" on page 397

Calling a Function

When a program is stopped, you can call a function using the `dbx call` command, which accepts values for the parameters that must be passed to the called function.

To call a procedure, type the name of the function and supply its parameters. For example:

```
(dbx) call change_glyph(1,3)
```

While the parameters are optional, you must type the parentheses after the *function_name*. For example:

```
(dbx) call type_vehicle()
```

You can call a function explicitly, using the `call` command, or implicitly, by evaluating an expression containing function calls or using a conditional modifier such as `stop in glyph -if animate()`.

A C++ virtual function can be called like any other function using the `print` command or `call` command (see "`print` Command" on page 358 or "`call` Command" on page 302), or any other command that executes a function call.

If the source file in which the function is defined was compiled with the `-g` option, or if the prototype declaration is visible at the current scope, `dbx` checks the number and type of arguments and issues an error message if there is a mismatch. Otherwise, `dbx` does not check the number of parameters and proceeds with the call.

By default, after every `call` command, `dbx` automatically calls `fflush(stdout)` to ensure that any information stored in the I/O buffer is printed. To turn off automatic flushing, set the `dbx` environment variable `output_autoflush` to `off`.

For C++, `dbx` handles the implicit `this` pointer, default arguments, and function overloading. The C++ overloaded functions are resolved automatically if possible. If any ambiguity remains (for example, functions not compiled with `-g`), `dbx` displays a list of the overloaded names.

When you use the `call` command, `dbx` behaves as though you used the `next` command, returning from the called function. However, if the program encounters a breakpoint in the called function, `dbx` stops the program at the breakpoint and issues a message. If you now type a `where` command, the stack trace shows that the call originated from `dbx` command level.

If you continue execution, the call returns normally. If you attempt to kill, run, rerun, or debug, the command aborts as `dbx` tries to recover from the nesting. You can then re-issue the command. Alternatively, you can use the command `pop -c` to pop all frames up to the most recent call.

Using Ctrl+C to Stop a Process

You can stop a process running in `dbx` by pressing Ctrl+C (`^C`). When you stop a process using `^C`, `dbx` ignores the `^C`, but the child process accepts it as a `SIGINT` and stops. You can then inspect the process as if it had been stopped by a breakpoint.

To resume execution after stopping a program with `^C`, use the `cont` command. You do not need to use the `cont` optional modifier, `sig` *signal_name*, to resume execution. The `cont` command resumes the child process after cancelling the pending signal.

CHAPTER 6

Setting Breakpoints and Traces

When an event occurs, `dbx` allows you to stop a process, execute arbitrary commands, or print information. The simplest example of an event is a breakpoint. Examples of other events are faults, signals, system calls, calls to `dlopen()`, and data changes.

A trace displays information about an event in your program, such as a change in the value of a variable. Although a trace's behavior is different from that of a breakpoint, traces and breakpoints share similar event handlers (see "Event Handlers" on page 275).

This chapter describes how to set, clear, and list breakpoints and traces. For complete information on the event specifications you can use in setting breakpoints and traces, see "Setting Event Specifications" on page 277.

The chapter is organized into the following sections:

Setting Breakpoints

In `dbx`, you can use three commands to set breakpoints:

- `stop` breakpoints–If the program arrives at a breakpoint created with a `stop` command, the program halts. The program cannot resume until you issue another debugging command, such as `cont`, `step`, or `next`.

- `when` breakpoints–If the program arrives at a breakpoint created with a `when` command, the program halts and `dbx` executes one or more debugging commands, then the program continues (unless one of the executed commands is `stop`).
- `trace` breakpoints–If a program arrives at a breakpoint created with a `trace` command, the program halts and an event-specific `trace` information line is emitted, then the program continues.

The `stop`, `when`, and `trace` commands all take as an argument an event specification, which describes the event on which the breakpoint is based. Event specifications are discussed in detail in "Setting Event Specifications" on page 277.

To set machine-level breakpoints, use the `stopi`, `wheni`, and `tracei` commands (see Chapter 18).

Note – When debugging an application that is a mixture of Java™ code and C JNI (Java™ Native Interface) code or C++ JNI code, you may want to set breakpoints in code that has not yet been loaded. For information on setting breakpoints on such code, see "Setting Breakpoints on Code That Has Not Yet Been Loaded by the JVM Software" on page 235.

Setting a `stop` Breakpoint at a Line of Source Code

You can set a breakpoint at a line number, using the `stop at` command, where *n* is a source code line number and *filename* is an optional program file name qualifier.

```
(dbx) stop at filename: n
```

For example:

```
(dbx) stop at main.cc:3
```

If the line specified is not an executable line of source code, `dbx` sets the breakpoint at the next executable line. If there is no executable line, `dbx` issues an error.

You can determine the line at which you wish to stop by using the `file` command to set the current file and the `list` command to list the function in which you wish to stop. Then use the `stop at` command to set the breakpoint on the source line:

```
(dbx) file t.c
(dbx) list main
10    main(int argc, char *argv[])
11    {
12        char *msg = "hello world\n";
13        printit(msg);
14    }
(dbx) stop at 13
```

For more information on specifying at an location event, see "at [filename:]line_number" on page 278.

Setting a `stop` Breakpoint in a Function

You can set a breakpoint in a function, using the `stop in` command:

```
(dbx) stop in function
```

An In Function breakpoint suspends program execution at the beginning of the first source line in a procedure or function.

`dbx` should be able to determine which variable or function you are referring to except when:

- You reference an overloaded function by name only.
- You reference a function or variable with a leading `.

Consider the following set of declarations:

```
int foo(double);
int foo(int);
int bar();
class x {
   int bar();
};
```

When you stop at a non-member function, you can type:

```
stop in foo(int)
```

to set a breakpoint at the global `foo(int)`.

To set a breakpoint at the member function you can use the command:

```
stop in x::bar()
```

If you type:

```
stop in foo
```

`dbx` cannot determine whether you mean the global function `foo(int)` or the global function `foo(double)` and may be forced to display an overloaded menu for clarification.

If you type:

```
stop in `bar
```

`dbx` cannot determine whether you mean the global function `bar()` or the member function `bar()` and displays an overload menu.

For more information on specifying an in function event, see "`in` function" on page 278.

Setting Multiple Breaks in C++ Programs

You can check for problems related to calls to members of different classes, calls to any members of a given class, or calls to overloaded top-level functions. You can use a keyword—`inmember`, `inclass`, `infunction`, or `inobject`—with a `stop`, `when`, or `trace` command to set multiple breaks in C++ code.

Setting Breakpoints in Member Functions of Different Classes

To set a breakpoint in each of the object-specific variants of a particular member function (same member function name, different classes), use `stop inmember`.

For example, if the function `draw` is defined in several different classes, then to place a breakpoint in each function, type:

```
(dbx) stop inmember draw
```

For more information on specifying an `inmember` or `inmethod` event, see "`inmember function inmethod function`" on page 279.

Setting Breakpoints in Member Functions of the Same Class

To set a breakpoint in all member functions of a specific class, use the `stop inclass` command.

By default, breakpoints are inserted only in the class member functions defined in the class, not those that it might inherit from base classes. To insert breakpoints in the functions that inherit from the base classes also, specify the `-recurse` option

To set a breakpoint in all member functions defined in the class `shape`, type:

```
(dbx) stop inclass shape
```

To set a breakpoint in all member functions defined in the class shape, and also in functions that inherit from the class, type:

```
(dbx) stop inclass shape -recurse
```

For more information on specifying an `inclass` event, see "`inclass classname [-recurse | -norecurse]`" on page 279 and "`stop` Command" on page 375.

Due to the large number of breakpoints that may be inserted by `stop inclass` and other breakpoint selections, you should be sure to set the `dbx` environment variable `step_events` to `on` to speed up the `step` and `next` commands (see "Efficiency Considerations" on page 115).

Setting Multiple Breakpoints in Nonmember Functions

To set multiple breakpoints in nonmember functions with overloaded names (same name, different type or number of arguments), use the `stop infunction` command.

For example, if a C++ program has defined two versions of a function named `sort()` (one that passes an `int` type argument and the other a `float`) then, to place a breakpoint in both functions, type:

```
(dbx) stop infunction sort [command;]
```

For more information on specifying an `infunction` event, see "`infunction` function" on page 279.

Setting Breakpoints in Objects

Set an In Object breakpoint to check the operations applied to a specific object instance.

By default, an In Object breakpoint suspends program execution in all nonstatic member functions of the object's class, including inherited ones, when called from the object. To set a breakpoint to suspend program execution in only nonstatic member functions defined in the object's class and not inherited classes, specify the `-norecurse` option.

To set a breakpoint in all nonstatic member functions defined in the base class of object `foo`, and in all nonstatic member functions defined in inherited classes of object `foo`, type:

```
(dbx) stop inobject &foo
```

To set a breakpoint in all nonstatic member functions defined in the class of object foo, but not those defined in inherited classes of object `foo`, type:

```
(dbx) stop inobject &foo -norecurse
```

For more information on specifying an `inobject` event, see "`inobject` object-expression `[-recurse | -norecurse]`" on page 279 and "`stop` Command" on page 375

Setting Data Change Breakpoints

You can use data change breakpoints in `dbx` to note when the value of a variable or expression has changed.

Stopping Execution When an Address Is Accessed

To stop execution when a memory address has been accessed, type:

```
(dbx) stop access mode address-expression [, byte-size-expression]
```

mode specifies how the memory was accessed. It can be composed of one or all of the letters:

`r`	The memory at the specified address has been read.
`w`	The memory has been written to.
`x`	The memory has been executed.

mode can also contain either of the following:

`a`	Stops the process after the access (default).
`b`	Stops the process before the access.

In both cases the program counter will point at the offending instruction. The "before" and "after" refer to the side effect.

address-expression is any expression that can be evaluated to produce an address. If you give a symbolic expression, the size of the region to be watched is automatically deduced; you can override it by specifying *byte-size-expression*. You can also use nonsymbolic, typeless address expressions; in which case, the size is mandatory.

In the following example, execution will stop execution after the memory address `0x4762` has been read:

```
(dbx) stop access r 0x4762
```

In this example, execution will stop before the variable speed has be written to:

```
(dbx) stop access wb &speed
```

Keep these points in mind when using the `stop access` command:

- The event occurs when a variable is written to even if it is the same value.
- By default, the event occurs after execution of the instruction that wrote to the variable. You can indicate that you want the event to occur before the instruction is executed by specifying the mode as b.

For more information on specifying an access event, see "access mode address-expression [, byte-size-expression]" on page 279 and "stop Command" on page 375.

Stopping Execution When Variables Change

To stop program execution if the value of a specified variable has changed, type:

```
(dbx) stop change variable
```

Keep these points in mind when using the `stop change` command:

- `dbx` stops the program at the line *after* the line that caused a change in the value of the specified variable.
- If *variable* is local to a function, the variable is considered to have changed when the function is first entered and storage for *variable* is allocated. The same is true with respect to parameters.
- The command does not work with multithreaded applications.

For more information on specifying a change event, see "change variable" on page 280 and "stop Command" on page 375.

`dbx` implements `stop change` by causing automatic single stepping together with a check on the value at each step. Stepping skips over library calls if the library was not compiled with the `-g` option. So, if control flows in the following manner, `dbx` does not trace the nested `user_routine2` because tracing skips the library call and the nested call to `user_routine2`.

```
user_routine calls
   library_routine, which calls
     user_routine2, which changes variable
```

The change in the value of *variable* appears to have occurred after the return from the library call, not in the middle of `user_routine2`.

`dbx` cannot set a breakpoint for a change in a block local variable—a variable nested in {}. If you try to set a breakpoint or trace in a block local "nested" variable, `dbx` issues an error informing you that it cannot perform this operation.

Note – It is faster to watch data changes using the `access` event than the `change` event. Instead of automatically single-stepping the program, the `access` event uses a page protection scheme that is much faster.

Stopping Execution on a Condition

To stop program execution if a conditional statement evaluates to true, type:

```
(dbx) stop cond condition
```

The program stops executing when the *condition* occurs.

Keep these points in mind when using the `stop cond` command:

- `dbx` stops the program at the line *after* the line that caused the condition to evaluate to true.
- The command does not work with multithreaded applications.

For more information on specifying a condition event, see "`cond` condition-expression" on page 281 and "`stop` Command" on page 375.

Setting Filters on Breakpoints

In `dbx`, most of the event management commands also support an optional *event filter* modifier. The simplest filter instructs `dbx` to test for a condition after the program arrives at a breakpoint or trace handler, or after a watch condition occurs.

If this filter condition evaluates to true (non 0), the event command applies and program execution stops at the breakpoint. If the condition evaluates to false (0), `dbx` continues program execution as if the event had never happened.

To set a breakpoint that includes a filter at a line or in a function, add an optional `-if` *condition* modifier statement to the end of a `stop` or `trace` command.

The condition can be any valid expression, including function calls, returning Boolean or integer in the language current at the time the command is entered.

With a location-based breakpoint like `in` or `at`, the scope is that of the breakpoint location. Otherwise, the scope of the condition is the scope at the time of entry, not at the time of the event. You might have to use the backquote operator (see "Backquote Operator" on page 79) to specify the scope precisely.

For example, these two filters are not the same:

```
stop in foo -if a>5
stop cond a>5
```

The former breaks at `foo` and tests the condition. The latter automatically single steps and tests for the condition.

Using the Return Value of a Function Call as a Filter

You can use the return value of a function call as a breakpoint filter. In this example, if the value in the string `str` is `abcde`, then execution stops in function `foo()`:

```
(dbx) stop in foo -if !strcmp("abcde",str)
```

Using Variable Scope as a Filter

Variable scope can be used in setting a breakpoint filter. In this example, the current scope is in `function foo()` and `local` is a local variable defined in `main()`:

```
(dbx) stop access w &main`local -if pr(main`local) -in main
```

Using a Filter With a Conditional Event

New users sometimes confuse setting a conditional event command (a watch-type command) with using filters. Conceptually, "watching" creates a *precondition* that must be checked before each line of code executes (within the scope of the watch). But even a breakpoint command with a conditional trigger can also have a filter attached to it.

Consider this example:

```
(dbx) stop access w &speed -if speed==fast_enough
```

This command instructs `dbx` to monitor the variable, *speed*; if the variable *speed* is written to (the "watch" part), then the `-if` filter goes into effect. `dbx` checks whether the new value of *speed* is equal to `fast_enough`. If it is not, the program continues, "ignoring" the `stop`.

In `dbx` syntax, the filter is represented in the form of an `[-if` *condition*`]` statement at the end of the command.

```
stop in function [-if condition]
```

If you set a breakpoint with a filter that contains function calls in a multithreaded program, dbx stops execution of all threads when it hits the breakpoint, and then evaluates the condition. If the condition is met and the function is called, dbx resumes all threads for the duration of the call.

For example, you might set the following breakpoint in a multithreaded application where many threads call `lookup()`:

```
(dbx) stop in lookup -if strcmp(name, "troublesome") == 0
```

dbx stops when thread `t@1` calls `lookup()`, evaluates the condition, and calls `strcmp()` resuming all threads. If dbx hits the breakpoint in another thread during the function call, it issues a warning such as one of the following:

```
event infinite loop causes missed events in the following handlers:
...
```

```
Event reentrancy
first event BPT(VID 6m TID 6, PC echo+0x8)
second event BPT*VID 10, TID 10, PC echo+0x8)
the following handlers will miss events:
...
```

In such a case, if you can ascertain that the function called in the conditional expression will not grab a mutex, you can use the `-resumeone` event specification modifier to force dbx to resume only the first thread in which it hit the breakpoint. For example, you might set the following breakpoint:

```
(dbx) stop in lookup -resumeone -if strcmp(name, "troublesome") =
= 0
```

The `-resumeone` modifier does not prevent problems in all cases. For example, it would not help if:

- The second breakpoint on `lookup()` occurs in the same thread as the first because the condition recursively calls `lookup()`.
- The thread on which the condition runs relinquishes control to another thread.

For detailed information on event modifiers, see "Event Specification Modifiers" on page 289.

Tracing Execution

Tracing collects information about what is happening in your program and displays it. If a program arrives at a breakpoint created with a `trace` command, the program halts and an event-specific `trace` information line is emitted, then the program continues.

A trace displays each line of source code as it is about to be executed. In all but the simplest programs, this trace produces volumes of output.

A more useful trace applies a filter to display information about events in your program. For example, you can trace each call to a function, every member function of a given name, every function in a class, or each exit from a function. You can also trace changes to a variable.

Setting a Trace

Set a trace by typing a `trace` command at the command line. The basic syntax of the `trace` command is:

```
trace event-specification [ modifier ]
```

For the complete syntax of the trace command, see "`trace` Command" on page 387.

The information a trace provides depends on the type of *event* associated with it (see "Setting Event Specifications" on page 277).

Controlling the Speed of a Trace

Often trace output goes by too quickly. The `dbx` environment variable `trace_speed` lets you control the delay after each trace is printed. The default delay is `0.5` seconds.

To set the interval in seconds between execution of each line of code during a trace, type:

```
dbxenv trace_speed number
```

Directing Trace Output to a File

You can direct the output of a trace to a file using the `-file` *filename* option. For example, the following command direct trace output to the file `trace1`:

```
(dbx) trace -file trace1
```

To revert trace output to standard output use - for *filename*. Trace output is always appended to *filename*. It is flushed whenever `dbx` prompts and when the application has exited. The *filename* is always re-opened on a new run or resumption after an attach.

Setting a `when` Breakpoint at a Line

A `when` breakpoint command accepts other `dbx` commands such as `list`, letting you write your own version of `trace`.

```
(dbx) when at 123 {list $lineno;}
```

The `when` command operates with an implied `cont` command. In the example above, after listing the source code at the current line, the program continues executing. If you included a `stop` command after the `list` command, the program would not continue executing.

For the complete syntax of the `when` command, see "`when` Command" on page 397. For detailed information on event modifiers, see "Event Specification Modifiers" on page 289.

Setting a Breakpoint in a Shared Library

`dbx` provides full debugging support for code that uses the programmatic interface to the run-time linker: code that calls `dlopen()`, `dlclose()` and their associated functions. The run-time linker binds and unbinds shared libraries during program execution. Debugging support for `dlopen()` and `dlclose()` lets you step into a function or set a breakpoint in functions in a dynamically shared library just as you can in a library linked when the program is started.

However, there are exceptions. dbx is unable to place breakpoints in loadobjects that have not been loaded (by, for example, using `dlopen()`:

- You cannot set a breakpoint in a library loaded by `dlopen()` before that library is loaded by `dlopen()`.
- You cannot set a breakpoint in a filter library loaded by `dlopen()` until the first function in it is called.

You can put the names of such loadobjects on the preload list with the `loadobject` command (see "`loadobject` Command" on page 345).

`dbx` does not forget about a loadobject that was loaded using `dlopen()`. For example, a breakpoint set in a freshly loaded loadobject remains until the next run command, or even if the loadobject is unloaded with `dlclose()` and then subsequently loaded with `dlopen()` again.

Listing and Clearing Breakpoints

Often, you set more than one breakpoint or trace handler during a debugging session. `dbx` supports commands for listing and clearing them.

Listing Breakpoints and Traces

To display a list of all active breakpoints, use the `status` command to display ID numbers in parentheses, which can then be used by other commands.

`dbx` reports multiple breakpoints set with the `inmember`, `inclass`, and `infunction` keywords as a single set of breakpoints with one status ID number.

Deleting Specific Breakpoints Using Handler ID Numbers

When you list breakpoints using the `status` command, `dbx` displays the ID number assigned to each breakpoint when it was created. Using the `delete` command, you can remove breakpoints by ID number, or use the keyword `all` to remove all breakpoints currently set anywhere in the program.

To delete breakpoints by ID number (in this case 3 and 5), type:

```
(dbx) delete 3 5
```

To delete all breakpoints set in the program currently loaded in dbx, type:

```
(dbx) delete all
```

For more information, see "delete Command" on page 323.

Enabling and Disabling Breakpoints

Each event management command (stop, trace, when) that you use to set a breakpoint creates an event handler (see "Event Handlers" on page 275). Each of these commands returns a number known as the handler ID (*hid*). You can use the handler ID as an argument to the handler command (see "handler Command" on page 336) to enable or disable the breakpoint.

Efficiency Considerations

Various events have different degrees of overhead in respect to the execution time of the program being debugged. Some events, like the simplest breakpoints, have practically no overhead. Events based on a single breakpoint have minimal overhead.

Multiple breakpoints such as inclass, that might result in hundreds of breakpoints, have an overhead only during creation time. This is because dbx uses permanent breakpoints; the breakpoints are retained in the process at all times and are not taken out on every stoppage and put in on every cont.

Note – In the case of step and next, by default all breakpoints are taken out before the process is resumed and reinserted once the step completes. If you are using many breakpoints or multiple breakpoints on prolific classes, the speed of step and next slows down considerably. Use the dbx step_events environment variable to control whether breakpoints are taken out and reinserted after each step or next.

The slowest events are those that utilize automatic single stepping. This might be explicit and obvious as in the `trace step` command, which single steps through every source line. Other events, like the `stop change` *expression* or `trace cond` *variable* not only single step automatically but also have to evaluate an expression or a variable at each step.

These events are very slow, but you can often overcome the slowness by bounding the event with a function using the `-in` modifier. For example:

```
trace next -in mumble
stop change clobbered_variable -in lookup
```

Do not use `trace -in main` because the `trace` is effective in the functions called by `main` as well. Do use it in the cases where you suspect that the `lookup()` function is clobbering your variable.

CHAPTER 7

Using the Call Stack

This chapter discusses how `dbx` uses the *call stack*, and how to use the `where`, `hide`, `unhide`, and `pop` commands when working with the call stack.

The call stack represents all currently active routines—routines that have been called but have not yet returned to their respective caller. A stack frame is a section to the call stack allocated for use by a single function.

Because the call stack grows from higher memory (larger addresses) to lower memory, *up* means going toward the caller's frame (and eventually `main()`) and *down* means going toward the frame of the called function (and eventually the current function). The frame for the routine executing when the program stopped at a breakpoint, after a single-step, or when a fault occurs and produces a core file, is in lower memory. A caller routine, such as `main()`, is located in higher memory.

This chapter is organized into the following sections:

Finding Your Place on the Stack

Use the `where` command to find your current location on the stack.

```
where [-f] [-h] [l] [-q] [-v] number_id
```

When debugging an application that is a mixture of Java™ code and C JNI (Java™ Native Interface) code or C++ JNI code, the syntax of the `where` command is:

```
where [-f] [-q] [-v] [ thread_id ] number_id
```

The `where` command is also useful for learning about the state of a program that has crashed and produced a core file. When this occurs, you can load the core file into `dbx` (see "Debugging a Core File" on page 50)

For more information on the `where` command, see "`where` Command" on page 399.

Walking the Stack and Returning Home

Moving up or down the stack is referred to as "walking the stack." When you visit a function by moving up or down the stack, `dbx` displays the current function and the source line. The location from which you start, *home*, is the point where the program stopped executing. From home, you can move up or down the stack using the `up`, `down`, or `frame` commands.

The `dbx` commands `up` and `down` both accept a *number* argument that instructs `dbx` to move a number of frames up or down the stack from the current frame. If *number* is not specified, the default is 1. The `-h` option includes all hidden frames in the count.

Moving Up and Down the Stack

You can examine the local variables in functions other than the current one.

Moving Up the Stack

To move up the call stack (toward `main`) *number* levels:

```
up [-h] [ number ]
```

If you do not specify *number*, the default is one level. For more information, see "`up` Command" on page 395.

Moving Down the Stack

To move down the call stack (toward the current stopping point) *number* levels:

```
down [-h] [ number ]
```

If you do not specify *number*, the default is one level. For more information, see "down Command" on page 326.

Moving to a Specific Frame

The `frame` command is similar to the `up` and `down` commands. It lets you go directly to the frame as given by numbers displayed by the `where` command.

```
frame
frame -h
frame [-h] number
frame [-h] +[number]
frame [-h] -[number]
```

The `frame` command without an argument displays the current frame number. With *number*, the command lets you go directly to the frame indicated by the number. By including a + (plus sign) or - (minus sign), the command lets you move an increment of one level up (+) or down (-). If you include a plus or minus sign with a *number*, you can move up or down the specified number of levels. The `-h` option includes any hidden frames in the count.

You can also move to a specific frame using the `pop` command (see "Popping the Call Stack" on page 119).

Popping the Call Stack

You can remove the stopped in function from the call stack, making the calling function the new stopped in function.

Unlike moving up or down the call stack, popping the stack changes the execution of your program. When the stopped in function is removed from the stack, it returns your program to its previous state, except for changes to global or static variables, external files, shared members, and similar global states.

The `pop` command removes one or more frames from the call stack. For example, to pop five frames from the stack, type:

```
pop 5
```

You can also pop to a specific frame. To pop to frame 5, type:

```
pop -f 5
```

For more information, see "`pop` Command" on page 357.

Hiding Stack Frames

Use the `hide` command to list the stack frame filters currently in effect.

To hide or delete all stack frames matching a regular expression, type:

```
hide [ regular_expression ]
```

The *regular_expression* matches either the function name, or the name of the loadobject, and uses `sh` or `ksh` syntax for file matching.

Use `unhide` to delete all stack frame filters.

```
unhide 0
```

Because the `hide` command lists the filters with numbers, you can also use the `unhide` command with the filter number.

```
unhide [ number | regular_expression ]
```

Displaying and Reading a Stack Trace

A stack trace shows where in the program flow execution stopped and how execution reached this point. It provides the most concise description of your program's state.

To display a stack trace, use the `where` command.

For functions that were compiled with the `-g` option, the names and types of the arguments are known so accurate values are displayed. For functions without debugging information hexadecimal numbers are displayed for the arguments. These numbers are not necessarily meaningful. When a function call is made through function pointer 0, the function value is shown as a low hexadecimal number instead of a symbolic name.

You can stop in a function that was not compiled with the `-g` option. When you stop in such a function `dbx` searches down the stack for the first frame whose function is compiled with the `-g` option and sets the current scope (see "Program Scope" on page 76) to it. This is denoted by the arrow symbol (=>).

In the following example, `main()` was compiled with the `-g` option, so the symbolic names as well as the values of the arguments are displayed The library functions called by `main()` were not compiled with `-g`, so the symbolic names of the functions are displayed but the hexadecimal contents of the SPARC input registers `$i0` through `$i5` are shown for the arguments:

```
(dbx) where
  [1] _libc_poll(0xffbef3b0, 0x1, 0xffffffff, 0x0, 0x10,
0xffbef604), at 0xfef9437c
  [2] _select(0xffbef3b8, 0xffbef580, 0xffbef500, 0xffbef584,
0xffbef504, 0x4), at 0xfef4e3dc
  [3] _XtWaitForSomething(0x5a418, 0x0, 0x0, 0xf4240, 0x0, 0x1),
at 0xff0bdb6c
  [4] XtAppNextEvent(0x5a418, 0x2, 0x2, 0x0, 0xffbef708, 0x1), at
0xff0bd5ec
  [5] XtAppMainLoop(0x5a418, 0x0, 0x1, 0x5532d, 0x3, 0x1), at
0xff0bd424
=>[6] main(argc = 1, argv = 0xffbef83c), line 48 in "main.cc"
```

In this example, the program has crashed with a segmentation fault. Again only `main()` has been compiled with the -g option, so the arguments to the library functions are displayed as hexadecimal without symbolic names. The cause of the crash is most likely the null arguments to `strlen()` in SPARC input registers `$i0` and `$i1`

```
(dbx) run
Running: Cdlib
(process id 6723)

CD Library Statistics:

 Titles:         1

 Total time:     0:00:00
 Average time:   0:00:00

signal SEGV (no mapping at the fault address) in strlen at
0xff2b6c5c
0xff2b6c5c: strlen+0x0080:ld      [%o1], %o2
Current function is main
(dbx) where
  [1] strlen(0x0, 0x0, 0x11795, 0x7efefeff, 0x81010100,
0xff339323), at 0xff2b6c5c
  [2] _doprnt(0x11799, 0x0, 0x0, 0x0, 0x0, 0xff00), at 0xff2fec18
  [3] printf(0x11784, 0xff336264, 0xff336274, 0xff339b94,
0xff331f98, 0xff00), at 0xff300780
=>[4] main(argc = 1, argv = 0xffbef894), line 133 in "Cdlib.c"
(dbx)
```

For more examples of stack traces, see "Looking at the Call Stack" on page 45 and "Tracing Calls" on page 218.

CHAPTER 8

Evaluating and Displaying Data

In dbx, you can perform two types of data checking:

- Evaluate data (print) – Spot-checks the value of an expression
- Display data (display) – Monitors the value of an expression each time the program stops

This chapter is organized into the following sections:

Evaluating Variables and Expressions

This section discusses how to use dbx to evaluate variables and expressions.

Verifying Which Variable dbx Uses

If you are not sure which variable dbx is evaluating, use the which command to see the fully qualified name dbx is using.

To see other functions and files in which a variable name is defined, use the whereis command.

For information on the commands, see "which Command" on page 401 and "whereis Command" on page 401.

Variables Outside the Scope of the Current Function

When you want to evaluate or monitor a variable outside the scope of the current function:

- Qualify the name of the function. See "Qualifying Symbols With Scope Resolution Operators" on page 79.

  or

- Visit the function by changing the current function. See "Navigating To Code" on page 73.

Printing the Value of a Variable, Expression, or Identifier

An expression should follow current language syntax, with the exception of the meta syntax that `dbx` introduces to deal with scope and arrays.

To evaluate a variable or expression in native code, type:

```
print expression
```

You can use the `print` command to evaluate an expression, local variable, or parameter in Java code.

For more information, see "`print` Command" on page 358.

Note – `dbx` supports the C++ `dynamic_cast` and `typeid` operators. When evaluating expressions with these two operators, `dbx` makes calls to certain rtti functions made available by the compiler. If the source doesn't explicitly use the operators, those functions might not have been generated by the compiler, and `dbx` fails to evaluate the expression.

Printing C++

In C++ an object pointer has two types, its *static type* (what is defined in the source code) and its *dynamic type* (what an object was before any casts were made to it). `dbx` can sometimes provide you with the information about the dynamic type of an object.

In general, when an object has a virtual function table (a vtable) in it, `dbx` can use the information in the vtable to correctly determine an object's type.

You can use the `print` or `display` command with the `-r` (recursive) option. `dbx` displays all the data members directly defined by a class and those inherited from a base class.

These commands also take a `-d` or `+d` option that toggles the default behavior of the `dbx` environment variable `output_derived_type`.

Using the `-d` flag or setting the `dbx` environment variable `output_dynamic_type` to `on` when there is no process running generates a "program is not active" error message because it is not possible to access dynamic information when there is no process. An "illegal cast on class pointers" error message is generated if you try to find a dynamic type through a virtual inheritance. (Casting from a virtual base class to a derived class is not legal in C++.)

Evaluating Unnamed Arguments in C++ Programs

C++ lets you define functions with unnamed arguments. For example:

```
void tester(int)
{
};
main(int, char **)
{
   tester(1);
};
```

Though you cannot use unnamed arguments elsewhere in a program, the compiler encodes unnamed arguments in a form that lets you evaluate them. The form is as follows, where the compiler assigns an integer to `%n`:

```
_ARG%n
```

To obtain the name assigned by the compiler, type the `whatis` command with the function name as its target.

```
(dbx) whatis tester
void tester(int _ARG1);
(dbx) whatis main
int main(int _ARG1, char **_ARG2);
```

For more information, see "`whatis` Command" on page 396.

To evaluate (or display) an unnamed function argument, type:

```
(dbx) print _ARG1
_ARG1 = 4
```

Dereferencing Pointers

When you dereference a pointer, you ask for the contents of the container to which the pointer points.

To dereference a pointer, `dbx` displays the evaluation in the command pane; in this case, the value pointed to by `t`:

```
(dbx) print *t
*t = {
a = 4
}
```

Monitoring Expressions

Monitoring the value of an expression each time the program stops is an effective technique for learning how and when a particular expression or variable changes. The `display` command instructs `dbx` to monitor one or more specified expressions or variables. Monitoring continues until you turn it off with the `undisplay` command.

To display the value of a variable or expression each time the program stops, type:

`display` *expression*`, ...`

You can monitor more than one variable at a time. The `display` command used with no options prints a list of all expressions being displayed.

For more information, see "`display` Command" on page 325.

Turning Off Display (Undisplaying)

`dbx` continues to display the value of a variable you are monitoring until you turn off display with the `undisplay` command. You can turn off the display of a specified expression or turn off the display of all expressions currently being monitored.

To turn off the display of a particular variable or expression, type:

```
undisplay expression
```

To turn off the display of all currently monitored variables, type:

```
undisplay 0
```

For more information, see "`undisplay` Command" on page 392.

Assigning a Value to a Variable

To assign a value to a variable, type:

```
assign variable = expression
```

Evaluating Arrays

You evaluate arrays the same way you evaluate other types of variables.

Here is a sample Fortran array:

```
integer*4 arr(1:6, 4:7)
```

To evaluate the array, use the `print` command. For example:

```
(dbx) print arr(2,4)
```

The `dbx print` command lets you evaluate part of a large array. Array evaluation includes:

- Array Slicing – Prints any rectangular, *n*-dimensional box of a multidimensional array.
- Array Striding – Prints certain elements only, in a fixed pattern, within the specified slice (which may be an entire array).

You can slice an array, with or without striding. (The default stride value is 1, which means print each element.)

Array Slicing

Array slicing is supported in the `print` and `display` commands for C, C++, and Fortran.

Array Slicing Syntax for C and C++

For each dimension of an array, the full syntax of the `print` command to slice the array is:

```
print array-expression [first-expression .. last-expression : stride-expression]
```

where:

array-expression	Expression that should evaluate to an array or pointer type.
first-expression	First element to be printed. Defaults to 0.
last-expression	Last element to be printed. Defaults to upper bound.
stride-expression	Length of the stride (the number of elements skipped is *stride-expression*`-1`). Defaults to 1.

The first, last, and stride expressions are optional expressions that should evaluate to integers.

For example:

```
(dbx) print arr[2..4]
arr[2..4] =
[2] = 2
[3] = 3
[4] = 4
(dbx) print arr[..2]
arr[0..2] =
[0] = 0
[1] = 1
[2] = 2

(dbx) print arr[2..6:2]
arr[2..6:2] =
[2] = 2
[4] = 4
[6] = 6
```

Array Slicing Syntax for Fortran

For *each* dimension of an array, the full syntax of the `print` command to slice the array is:

```
print array-expression (first-expression : last-expression : stride-expression)
```

where:

array-expression	Expression that should evaluate to an array type.
first-expression	First element in a range, also first element to be printed. Defaults to lower bound.
last-expression	Last element in a range, but might not be the last element to be printed if stride is not equal to 1. Defaults to upper bound.
stride-expression	Length of the stride. Defaults to 1.

The first, last, and stride expressions are optional expressions that should evaluate to integers. For an *n*-dimensional slice, separate the definition of each slice with a comma.

For example:

```
(dbx) print arr(2:6)
arr(2:6) =
(2) 2
(3) 3
(4) 4
(5) 5
(6) 6

(dbx) print arr(2:6:2)
arr(2:6:2) =
(2) 2
(4) 4
(6) 6
```

To specify rows and columns, type:

```
demo% f77 -g -silent ShoSli.f
demo% dbx a.out
Reading symbolic information for a.out
(dbx) list 1,12
       1       INTEGER*4 a(3,4), col, row
       2       DO row = 1,3
       3           DO col = 1,4
       4             a(row,col) = (row*10) + col
       5           END DO
       6       END DO
       7       DO row = 1, 3
       8            WRITE(*,'(4I3)') (a(row,col),col=1,4)
       9       END DO
       10      END
(dbx) stop at 7
(1) stop at "ShoSli.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "ShoSli.f"
       7        DO row = 1, 3
```

To print row 3, type:

```
(dbx) print a(3:3,1:4)
'ShoSli'MAIN'a(3:3, 1:4) =
        (3,1)   31
        (3,2)   32
        (3,3)   33
        (3,4)   34
(dbx)
```

To print column 4, type:

```
(dbx) print a(1:3,4:4)
'ShoSli'MAIN'a(1:3, 1:4) =
        (1,4)   14
        (2,4)   24
        (3,4)   34
(dbx)
```

Slices

Here is an example of a two-dimensional, rectangular slice, with the default stride of 1 omitted.

```
print arr(201:203, 101:105)
```

This command prints a block of elements in a large array. Note that the command omits *stride-expression*, using the default stride value of 1.

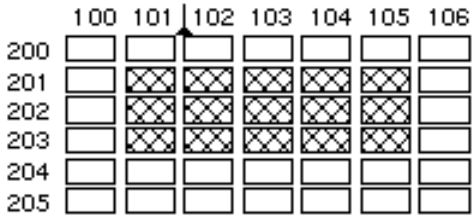


FIGURE 8-1 Example of a Two-dimensional, Rectangular Slice With a Stride of 1

As illustrated in FIGURE 8-1, the first two expressions (`201:203`) specify a slice in the first dimension of this two-dimensional array (the three-row column). The slice starts with row `201` and ends with `203`. The second set of expressions, separated by a comma from the first, defines the slice for the second dimension. The slice begins with column `101` and ends with column `105`.

Strides

When you instruct `print` to *stride* across a slice of an array, `dbx` evaluates certain elements in the slice only, skipping over a fixed number of elements between each one it evaluates.

The third expression in the array slicing syntax, *stride-expression*, specifies the length of the stride. The value of *stride-expression* specifies the elements to print. The default stride value is 1, meaning: evaluate all of the elements in the specified slices.

Here is the same array used in the previous example of a slice. This time the print command includes a stride of 2 for the slice in the second dimension.

```
print arr(201:203, 101:105:2)
```

As shown in FIGURE 8-2, a stride of 2 prints every second element, skipping every other element.

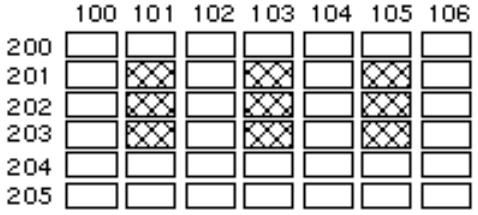


FIGURE 8-2 Example of a Two-dimensional, Rectangular Slice With a Stride of 2

For any expression you omit, print takes a default value equal to the declared size of the array. Here are examples showing how to use the shorthand syntax.

For a one-dimensional array, use the following commands:

`print arr`	Prints the entire array with default boundaries.
`print arr(:)`	Prints the entire array with default boundaries and default stride of 1.
`print arr(::`*stride-expression*`)`	Prints the entire array with a stride of *stride-expression*.

For a two-dimensional array, the following command prints the entire array.

```
print arr
```

To print every third element in the second dimension of a two-dimensional array, type:

```
print arr (:,::3)
```

CHAPTER 9

Using Runtime Checking

Runtime checking (RTC) lets you automatically detect runtime errors, such as memory access errors and memory leak, in a native code application during the development phase. It also lets you monitor memory usage. You cannot use runtime checking on Java code.

The following topics are covered in this chapter:

Note – Runtime checking is available on Solaris platforms only. It is not available on Linux platforms.

Note – Memory access checking is available on SPARC platforms only. It is not available on Solaris OS x86 platforms

Capabilities of Runtime Checking

Because runtime checking is an integral debugging feature, you can perform all debugging operations while using runtime checking except collecting performance data using the Collector.

Runtime checking:

- Detects memory access errors
- Detects memory leaks
- Collects data on memory use
- Works with all languages
- Works with multithreaded code
- Requires no recompiling, relinking, or makefile changes

Compiling with the `-g` flag provides source line number correlation in the runtime checking error messages. Runtime checking can also check programs compiled with the optimization `-O` flag. There are some special considerations with programs not compiled with the `-g` option.

You can use runtime checking by using the `check` command.

When to Use Runtime Checking

One way to avoid seeing a large number of errors at once is to use runtime checking earlier in the development cycle—as you are developing the individual modules that make up your program. Write a unit test to drive each module and use runtime checking incrementally to check one module at a time. That way, you deal with a smaller number of errors at a time. When you integrate all of the modules into the full program, you are likely to encounter few new errors. When you reduce the number of errors to zero, you need to run runtime checking again only when you make changes to a module.

Runtime Checking Requirements

To use runtime checking, you must fulfill the following requirements:

- Programs compiled using a Sun compiler.
- Dynamic linking with `libc`.

- Use of the standard `libc malloc`, `free`, and `realloc` functions or allocators based on those functions. Runtime checking provides an application programming interface (API) to handle other allocators. See "Runtime Checking Application Programming Interface" on page 162.
- Programs that are not fully stripped; programs stripped with `strip -x` are acceptable.

Limitations

Runtime checking does not handle program text areas and data areas larger than 8 megabytes on hardware that is not based on UltraSPARC® processors. For more information, see "Runtime Checking's 8 Megabyte Limit" on page 164.

A possible solution is to insert special files in the executable image to handle program text areas and data areas larger than 8 megabytes.

Using Runtime Checking

To use runtime checking, enable the type of checking you want to use before you run the program.

Turning On Memory Use and Memory Leak Checking

To turn on memory use and memory leak checking, type:

```
(dbx) check -memuse
```

When memory use checking or memory leak checking is turned on, the `showblock` command shows the details about the heap block at a given address. The details include the location of the block's allocation and its size. For more information, see "`showblock` Command" on page 369.

Turning On Memory Access Checking

To turn on memory access checking only, type:

```
(dbx) check -access
```

Note – Memory access checking is not available on Solaris OS x86 platforms.

Turning On All Runtime Checking

To turn on memory leak, memory use, and memory access checking, type:

```
(dbx) check -all
```

For more information, see "check Command" on page 304.

Turning Off Runtime Checking

To turn off runtime checking entirely, type:

```
(dbx) uncheck -all
```

For detailed information, see "uncheck Command" on page 391.

Running Your Program

After turning on the types of runtime checking you want, run the program being tested, with or without breakpoints.

The program runs normally, but slowly because each memory access is checked for validity just before it occurs. If `dbx` detects invalid access, it displays the type and location of the error. Control returns to you (unless the `dbx` environment variable `rtc_auto_continue` is set to `on` (see "Setting `dbx` Environment Variables" on page 66.)

You can then issue `dbx` commands, such as `where` to get the current stack trace or `print` to examine variables. If the error is not a fatal error, you can continue execution of the program with the `cont` command. The program continues to the next error or breakpoint, whichever is detected first. For detailed information, see "`cont` Command" on page 316.

If `rtc_auto_continue` is set to `on`, runtime checking continues to find errors, and keeps running automatically. It redirects errors to the file named by the `dbx` environment variable `rtc_error_log_file_name`. (See "Setting `dbx` Environment Variables" on page 66.) The default log file name is `/tmp/dbx.errlog.`*uniqueid*.

You can limit the reporting of runtime checking errors using the `suppress` command. For detailed information, see "`suppress` Command" on page 380.

Below is a simple example showing how to turn on memory access and memory use checking for a program called `hello.c`.

```
% cat -n hello.c
     1  #include <stdio.h>
     2  #include <stdlib.h>
     3  #include <string.h>
     4
     5  char *hello1, *hello2;
     6
     7  void
     8  memory_use()
     9  {
    10       hello1 = (char *)malloc(32);
    11       strcpy(hello1, "hello world");
    12       hello2 = (char *)malloc(strlen(hello1)+1);
    13       strcpy(hello2, hello1);
    14  }
    15
    16  void
    17  memory_leak()
    18  {
```

```
       19      char *local;
       20      local = (char *)malloc(32);
       21      strcpy(local, "hello world");
       22 }
       23
       24 void
       25 access_error()
       26 {
       27      int i,j;
       28
       29      i = j;
       30 }
       31
       32 int
       33 main()
       34 {
       35      memory_use();
       36      access_error();
       37      memory_leak();
       38      printf("%s\n", hello2);
       39      return 0;
       40 }
% cc -g -o hello hello.c
```

```
% dbx -C hello
Reading ld.so.1
Reading librtc.so
Reading libc.so.1
Reading libdl.so.1

(dbx) check -access
access checking - ON
(dbx) check -memuse
memuse checking - ON
(dbx) run Running: hello
(process id 18306)
Enabling Error Checking... done
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff068
        which is 96 bytes above the current stack pointer
Variable is 'j'
Current function is access_error
       29       i = j;
(dbx) cont
hello world
Checking for memory leaks...
Actual leaks report    (actual leaks:        1 total size:     32 bytes)

 Total  Num of  Leaked      Allocation call stack
 Size   Blocks  Block
                       Address
======  ====== ==========  ======================================
       32       1    0x21aa8  memory_leak < main

Possible leaks report  (possible leaks:      0  total size:     0 bytes)

Checking for memory use...
Blocks in use report   (blocks in use:       2  total size:     44 bytes

 Total  % of Num of  Avg     Allocation call stack
 Size    All Blocks  Size
======= ==== ====== ======  ======================================
        32  72%      1     32  memory_use < main
        12  27%      1     12  memory_use < main

execution completed, exit code is 0
```

The function `access_error()` reads variable `j` before it is initialized. Runtime checking reports this access error as a Read from uninitialized (`rui`).

The function `memory_leak()` does not free the variable `local` before it returns. When `memory_leak()` returns, this variable goes out of scope and the block allocated at line 20 becomes a leak.

The program uses global variables `hello1` and `hello2`, which are in scope all the time. They both point to dynamically allocated memory, which is reported as Blocks in use (`biu`).

Using Access Checking

Access checking checks whether your program accesses memory correctly by monitoring each read, write, and memory free operation.

Note – Memory access checking is not available on Solaris OS x86 platforms.

Programs might incorrectly read or write memory in a variety of ways; these are called memory access errors. For example, the program may reference a block of memory that has been deallocated through a `free()`call for a heap block. Or a function might return a pointer to a local variable, and when that pointer is accessed an error would result. Access errors might result in wild pointers in the program and can cause incorrect program behavior, including wrong outputs and segmentation violations. Some kinds of memory access errors can be very hard to track down.

Runtime checking maintains a table that tracks the state of each block of memory being used by the program. Runtime checking checks each memory operation against the state of the block of memory it involves and then determines whether the operation is valid. The possible memory states are:

- Unallocated, initial state – Memory has not been allocated. It is illegal to read, write, or free this memory because it is not owned by the program.
- Allocated, but uninitialized – Memory has been allocated to the program but not initialized. It is legal to write to or free this memory, but is illegal to read it because it is uninitialized. For example, upon entering a function, stack memory for local variables is allocated, but uninitialized.
- Read-only – It is legal to read, but not write or free, read-only memory.
- Allocated and initialized – It is legal to read, write, or free allocated and initialized memory.

Using runtime checking to find memory access errors is not unlike using a compiler to find syntax errors in your program. In both cases, a list of errors is produced, with each error message giving the cause of the error and the location in the program where the error occurred. In both cases, you should fix the errors in your program starting at the top of the error list and working your way down. One error can cause other errors in a chain reaction. The first error in the chain is, therefore, the “first cause,” and fixing that error might also fix some subsequent errors.

For example, a read from an uninitialized section of memory can create an incorrect pointer, which when dereferenced can cause another invalid read or write, which can in turn lead to yet another error.

Understanding the Memory Access Error Report

Runtime checking prints the following information for memory access errors:

Error	Information
`type`	Type of error.
`access`	Type of access attempted (read or write).
`size`	Size of attempted access.
`addr`	Address of attempted access.
`detail`	More detailed information about `addr`. For example, if `addr` is in the vicinity of the stack, then its position relative to the current stack pointer is given. If `addr` is in the heap, then the address, size, and relative position of the nearest heap block is given.
`stack`	Call stack at time of error (with batch mode).
`allocation`	If `addr` is in the heap, then the allocation trace of the nearest heap block are given.
`location`	Where the error occurred. If line number information is available, this information includes line number and function. If line numbers are not available, runtime checking provides function and address.

The following example shows a typical access error.

```
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xefffee50
       which is 96 bytes above the current stack pointer
Variable is 'j'
Current function is rui
   12           i = j;
```

Memory Access Errors

Runtime checking detects the following memory access errors:

- `rui` (see "Read From Uninitialized Memory (`rui`) Error" on page 168)
- `rua` (see "Read From Unallocated Memory (`rua`) Error" on page 168)

- `wua` (see "Write to Unallocated Memory (wua) Error" on page 169)
- `wro` (see "Write to Read-Only Memory (wro) Error" on page 169)
- `mar` (see "Misaligned Read (mar) Error" on page 167)
- `maw` (see "Misaligned Write (maw) Error" on page 168)
- `duf` (see "Duplicate Free (duf) Error" on page 167)
- `baf` (see "Bad Free (baf) Error" on page 166)
- `maf` (see "Misaligned Free (maf) Error" on page 167)
- `oom` (see "Out of Memory (oom) Error" on page 168)

Note – Runtime checking does not perform array bounds checking and, therefore, does not report array bound violations as access errors.

Using Memory Leak Checking

A memory leak is a dynamically allocated block of memory that has no pointers pointing to it anywhere in the data space of the program. Such blocks are orphaned memory. Because there are no pointers pointing to the blocks, programs cannot reference them, much less free them. Runtime checking finds and reports such blocks.

Memory leaks result in increased virtual memory consumption and generally result in memory fragmentation. This might slow down the performance of your program and the whole system.

Typically, memory leaks occur because allocated memory is not freed and you lose a pointer to the allocated block. Here are some examples of memory leaks:

```
void
foo()
{
    char *s;
    s = (char *) malloc(32);

    strcpy(s, "hello world");

    return; /* no free of s. Once foo returns, there is no     */
           /* pointer pointing to the malloc'ed block,         */
            /* so that block is leaked.                        */
}
```

A leak can result from incorrect use of an API.

```
void
printcwd()
{

    printf("cwd = %s\n", getcwd(NULL, MAXPATHLEN));

    return; /* libc function getcwd() returns a pointer to     */
            /* malloc'ed area when the first argument is NULL, */
            /* program should remember to free this. In this   */
            /* case the block is not freed and results in leak.*/
}
```

You can avoid memory leaks by always freeing memory when it is no longer needed and paying close attention to library functions that return allocated memory. If you use such functions, remember to free up the memory appropriately.

Sometimes the term *memory leak* is used to refer to any block that has not been freed. This is a much less useful definition of a memory leak, because it is a common programming practice not to free memory if the program will terminate shortly. Runtime checking does not report a block as a leak, if the program still retains one or more pointers to it.

Detecting Memory Leak Errors

Runtime checking detects the following memory leak errors:

- `mel` (see "Memory Leak (`mel`) Error" on page 170)
- `air` (see "Address in Register (`air`) Error" on page 170)
- `aib` (see "Address in Block (`aib`) Error" on page 169)

Note – Runtime checking only finds leaks of `malloc` memory. If your program does not use `malloc`, runtime checking cannot find memory leaks.

Possible Leaks

There are two cases where runtime checking can report a "possible" leak. The first case is when no pointers are found pointing to the beginning of the block, but a pointer is found pointing to the *interior* of the block. This case is reported as an "Address in Block (`aib`)" error. If it was a stray pointer that pointed into the block, this would be a real memory leak. However, some programs deliberately move the

only pointer to an array back and forth as needed to access its entries. In this case, it would not be a memory leak. Because runtime checking cannot distinguish between these two cases, it reports both of them as possible leaks, letting you determine which are real memory leaks.

The second type of possible leak occurs when no pointers to a block are found in the data space, but a pointer is found in a register. This case is reported as an "Address in Register (`air`)" error. If the register points to the block accidentally, or if it is an old copy of a memory pointer that has since been lost, then this is a real leak. However, the compiler can optimize references and place the only pointer to a block in a register without ever writing the pointer to memory. Such a case would not be a real leak. Hence, if the program has been optimized and the report was the result of the `showleaks` command, it is likely not to be a real leak. In all other cases, it is likely to be a real leak. For more information, see "`showleaks` Command" on page 369.

Note – Runtime leak checking requires the use of the standard `libc malloc/free/realloc` functions or allocators based on those functions. For other allocators, see "Runtime Checking Application Programming Interface" on page 162.

Checking for Leaks

If memory leak checking is turned on, a scan for memory leaks is automatically performed just before the program being tested exits. Any detected leaks are reported. The program should not be killed with the `kill` command. Here is a typical memory leak error message:

```
Memory leak (mel):
Found leaked block of size 6 at address 0x21718
At time of allocation, the call stack was:
    [1] foo() at line 63 in test.c
    [2] main() at line 47 in test.c
```

A UNIX program has a `main` procedure (called `MAIN` in f77) that is the top-level user function for the program. Normally, a program terminates either by calling `exit(3)` or by returning from `main`. In the latter case, all variables local to `main` go out of scope after the return, and any heap blocks they pointed to are reported as leaks (unless global variables point to those same blocks).

It is a common programming practice not to free heap blocks allocated to local variables in `main`, because the program is about to terminate and return from `main` without calling `exit()`. To prevent runtime checking from reporting such blocks as memory leaks, stop the program just before `main` returns by setting a breakpoint on

the last executable source line in `main`. When the program halts there, use the `showleaks` command to report all the true leaks, omitting the leaks that would result merely from variables in `main` going out of scope.

For more information, see "`showleaks` Command" on page 369.

Understanding the Memory Leak Report

With leak checking turned on, you receive an automatic leak report when the program exits. All possible leaks are reported—provided the program has not been killed using the `kill` command. The level of detail in the report is controlled by the `dbx` environment variable `rtc_mel_at_exit` (see "Setting `dbx` Environment Variables" on page 66). By default, a nonverbose leak report is generated.

Reports are sorted according to the combined size of the leaks. Actual memory leaks are reported first, followed by possible leaks. The verbose report contains detailed stack trace information, including line numbers and source files whenever they are available.

Both reports include the following information for memory leak errors:

Information	Description
location	Location where leaked block was allocated.
addr	Address of leaked block.
size	Size of leaked block.
stack	Call stack at time of allocation, as constrained by `check -frames`.

Here is the corresponding nonverbose memory leak report.

```
Actual leaks report   (actual leaks:   3 total size:   2427 bytes)

 Total  Num of  Leaked      Allocation call stack
 Size   Blocks  Block
                Address
======  ====== ==========  ======================================
  1852       2      -      true_leak < true_leak
   575       1    0x22150  true_leak < main

Possible leaks report  (possible leaks:  1  total size:       8
bytes)

 Total  Num of  Leaked      Allocation call stack
 Size   Blocks  Block
                Address
======  ====== ==========  ======================================
     8       1    0x219b0  in_block < main
```

Following is a typical verbose leak report.

```
Actual leaks report    (actual leaks:         3  total size:
2427 bytes)

Memory Leak (mel):
Found 2 leaked blocks with total size 1852 bytes
At time of each allocation, the call stack was:
       [1] true_leak() at line 220 in "leaks.c"
       [2] true_leak() at line 224 in "leaks.c"

Memory Leak (mel):
Found leaked block of size 575 bytes at address 0x22150
At time of allocation, the call stack was:
       [1] true_leak() at line 220 in "leaks.c"
       [2] main() at line 87 in "leaks.c"

Possible leaks report  (possible leaks:      1  total size:
8 bytes)

Possible memory leak -- address in block (aib):
Found leaked block of size 8 bytes at address 0x219b0
At time of allocation, the call stack was:
       [1] in_block() at line 177 in "leaks.c"
       [2] main() at line 100 in "leaks.c"
```

Generating a Leak Report

You can ask for a leak report at any time using the `showleaks` command, which reports new memory leaks since the last `showleaks` command. For more information, see "`showleaks` Command" on page 369.

Combining Leaks

Because the number of individual leaks can be very large, runtime checking automatically combines leaks allocated at the same place into a single combined leak report. The decision to combine leaks, or report them individually, is controlled by the `number-of-frames-to-match` parameter specified by the `-match` *m* option on a `check -leaks` or the `-m` option of the `showleaks` command. If the call stack at the time of allocation for two or more leaks matches to *m* frames to the exact program counter level, these leaks are reported in a single combined leak report.

Consider the following three call sequences:

Block 1	Block 2	Block 3
`[1] malloc`	`[1] malloc`	`[1] malloc`
`[2] d() at 0x20000`	`[2] d() at 0x20000`	`[2] d() at 0x20000`
`[3] c() at 0x30000`	`[3] c() at 0x30000`	`[3] c() at 0x31000`
`[4] b() at 0x40000`	`[4] b() at 0x41000`	`[4] b() at 0x40000`
`[5] a() at 0x50000`	`[5] a() at 0x50000`	`[5] a() at 0x50000`

If all of these blocks lead to memory leaks, the value of *m* determines whether the leaks are reported as separate leaks or as one repeated leak. If *m* is 2, Blocks 1 and 2 are reported as one repeated leak because the 2 stack frames above `malloc()` are common to both call sequences. Block 3 will be reported as a separate leak because the trace for `c()` does not match the other blocks. For *m* greater than 2, runtime checking reports all leaks as separate leaks. (The `malloc` is not shown on the leak report.)

In general, the smaller the value of *m*, the fewer individual leak reports and the more combined leak reports are generated. The greater the value of *m*, the fewer combined leak reports and the more individual leak reports are generated.

Fixing Memory Leaks

Once you have obtained a memory leak report, follow these guidelines for fixing the memory leaks.

- Most importantly, determine where the leak is. The leak report tells you the allocation trace of the leaked block, the place where the leaked block was allocated.
- You can then look at the execution flow of your program and see how the block was used. If it is obvious where the pointer was lost, the job is easy; otherwise you can use `showleaks` to narrow your leak window. By default the `showleaks` command gives you the new leaks created only since the last `showleaks` command. You can run `showleaks` repeatedly while stepping through your program to narrow the window where the block was leaked.

For more information, see "`showleaks` Command" on page 369.

Using Memory Use Checking

Memory use checking lets you see all the heap memory in use. You can use this information to get a sense of where memory is allocated in your program or which program sections are using the most dynamic memory. This information can also be useful in reducing the dynamic memory consumption of your program and might help in performance tuning

Memory use checking is useful during performance tuning or to control virtual memory use. When the program exits, a memory use report can be generated. Memory usage information can also be obtained at any time during program execution with the `showmemuse` command, which causes memory usage to be displayed. For information, see "`showmemuse` Command" on page 370.

Turning on memory use checking also turns on leak checking. In addition to a leak report at the program exit, you also get a blocks in use (biu) report. By default, a nonverbose blocks in use report is generated at program exit. The level of detail in the memory use report is controlled by the `dbx` environment variable `rtc_biu_at_exit` (see "Setting `dbx` Environment Variables" on page 66).

The following is a typical nonverbose memory use report.

```
Blocks in use report   (blocks in use: 5   total size:   40 bytes)

 Total  % of Num of  Avg     Allocation call stack
 Size    All Blocks  Size
======= ==== ====== ======  ====================================
     16  40%      2      8  nonleak < nonleak
      8  20%      1      8  nonleak < main
      8  20%      1      8  cyclic_leaks < main
      8  20%      1      8  cyclic_leaks < main
```

The following is the corresponding verbose memory use report:

```
Blocks in use report   (blocks in use: 5   total size:   40 bytes)

Block in use (biu):
Found 2 blocks totaling 16 bytes (40.00% of total; avg block size
8)
At time of each allocation, the call stack was:
     [1] nonleak() at line 182 in "memuse.c"
     [2] nonleak() at line 185 in "memuse.c"

Block in use (biu):
Found block of size 8 bytes at address 0x21898 (20.00% of total)
At time of allocation, the call stack was:
     [1] nonleak() at line 182 in "memuse.c"
     [2] main() at line 74 in "memuse.c"

Block in use (biu):
Found block of size 8 bytes at address 0x21958 (20.00% of total)
At time of allocation, the call stack was:
     [1] cyclic_leaks() at line 154 in "memuse.c"
     [2] main() at line 118 in "memuse.c"

Block in use (biu):
Found block of size 8 bytes at address 0x21978 (20.00% of total)
At time of allocation, the call stack was:
     [1] cyclic_leaks() at line 155 in "memuse.c"
     [2] main() at line 118 in "memuse.c"
```

You can ask for a memory use report any time with the `showmemuse` command.

Suppressing Errors

Runtime checking provides a powerful error suppression facility that allows great flexibility in limiting the number and types of errors reported. If an error occurs that you have suppressed, then no report is given, and the program continues as if no error had occurred.

You can suppress errors using the `suppress` command (see "`suppress` Command" on page 380).

You can undo error suppression using the `unsuppress` command (see "`unsuppress` Command" on page 394).

Suppression is persistent across `run` commands within the same debug session, but not across `debug` commands.

Types of Suppression

The following types of suppression are available:

Suppression by Scope and Type

You must specify which type of error to suppress. You can specify which parts of the program to suppress. The options are:

Option	Description
Global	The default; applies to the whole program.
Load Object	Applies to an entire load object, such as a shared library, or the main program.
File	Applies to all functions in a particular file.
Function	Applies to a particular function.
Line	Applies to a particular source line.
Address	Applies to a particular instruction at an address.

Suppression of Last Error

By default, runtime checking suppresses the most recent error to prevent repeated reports of the same error. This is controlled by the `dbx` environment variable `rtc_auto_suppress`. When `rtc_auto_suppress` is set to `on` (the default), a particular access error at a particular location is reported only the first time it is encountered and suppressed thereafter. This is useful, for example, for preventing multiple copies of the same error report when an error occurs in a loop that is executed many times.

Limiting the Number of Errors Reported

You can use the `dbx` environment variable `rtc_error_limit` to limit the number of errors that will be reported. The error limit is used separately for access errors and leak errors. For example, if the error limit is set to 5, then a maximum of five access errors and five memory leaks are shown in both the leak report at the end of the run and for each `showleaks` command you issue. The default is 1000.

Suppressing Error Examples

In the following examples, `main.cc` is a file name, `foo` and `bar` are functions, and `a.out` is the name of an executable.

Do not report memory leaks whose allocation occurs in function `foo`.

```
suppress mel in foo
```

Suppress reporting blocks in use allocated from `libc.so.1`.

```
suppress biu in libc.so.1
```

Suppress read from uninitialized in all functions in `a.out`.

```
suppress rui in a.out
```

Do not report read from unallocated in file `main.cc`.

```
suppress rua in main.cc
```

Suppress duplicate free at line 10 of `main.cc`.

```
suppress duf at main.cc:10
```

Suppress reporting of all errors in function `bar`.

```
suppress all in bar
```

For more information, see "`suppress` Command" on page 380.

Default Suppressions

To detect all errors, runtime checking does not require the program be compiled using the `-g` option (symbolic). However, symbolic information is sometimes needed to guarantee the correctness of certain errors, mostly `rui` errors. For this reason certain errors, `rui` for `a.out` and `rui`, `aib`, and `air` for shared libraries, are suppressed by default if no symbolic information is available. This behavior can be changed using the `-d` option of the `suppress` and `unsuppress` commands.

The following command causes runtime checking to no longer suppress read from uninitialized memory (`rui`) in code that does not have symbolic information (compiled without `-g`):

```
unsuppress -d rui
```

For more information, see "`unsuppress` Command" on page 394.

Using Suppression to Manage Errors

For the initial run on a large program, the large number of errors might be overwhelming. It might be better to take a phased approach. You can do so using the `suppress` command to reduce the reported errors to a manageable number, fixing just those errors, and repeating the cycle; suppressing fewer and fewer errors with each iteration.

For example, you could focus on a few error types at one time. The most common error types typically encountered are `rui`, `rua`, and `wua`, usually in that order. `rui` errors are less serious (although they can cause more serious errors to happen later).

Often a program might still work correctly with these errors. `rua` and `wua` errors are more serious because they are accesses to or from invalid memory addresses and always indicate a coding error.

You can start by suppressing `rui` and `rua` errors. After fixing all the `wua` errors that occur, run the program again, this time suppressing only `rui` errors. After fixing all the `rua` errors that occur, run the program again, this time with no errors suppressed. Fix all the `rui` errors. Lastly, run the program a final time to ensure no errors are left.

If you want to suppress the last reported error, use `suppress -last`.

Using Runtime Checking on a Child Process

To use runtime checking on a child process, you must have the `dbx` environment variable `rtc_inherit` set to `on`. By default, it is set to `off`. (See "Setting dbx Environment Variables" on page 66.)

`dbx` supports runtime checking of a child process if runtime checking is enabled for the parent and the `dbx` environment variable `follow_fork_mode` is set to `child` (see "Setting dbx Environment Variables" on page 66).

When a fork happens, `dbx` automatically performs runtime checking on the child. If the program calls `exec()`, the runtime checking settings of the program calling `exec()` are passed on to the program.

At any given time, only one process can be under runtime checking control. The following is an example.

```
% cat -n program1.c
     1 #include <sys/types.h>
     2 #include <unistd.h>
     3 #include <stdio.h>
     4
     5 int
     6 main()
     7 {
     8       pid_t child_pid;
     9       int parent_i, parent_j;
    10
    11       parent_i = parent_j;
```

```
       12
       13      child_pid = fork();
       14
       15      if (child_pid == -1) {
       16          printf("parent: Fork failed\n");
       17          return 1;
       18      } else if (child_pid == 0) {
       19          int child_i, child_j;
       20
       21          printf("child: In child\n");
       22          child_i = child_j;
       23          if (execl("./program2", NULL) == -1) {
       24             printf("child: exec of program2 failed\n");
       25               exit(1);
       26          }
       27      } else {
       28        printf("parent: child's pid = %d\n", child_pid);
       29      }
       30      return 0;
       31 }
```

```
% cat -n program2.c
     1
     2 #include <stdio.h>
     3
     4 main()
     5 {
     6      int program2_i, program2_j;
     7
     8      printf ("program2: pid = %d\n", getpid());
     9      program2_i = program2_j;
    10
    11      malloc(8);
    12
    13      return 0;
    14 }
%
```

```
% cc -g -o program1 program1.c
% cc -g -o program2 program2.c
% dbx -C program1
Reading symbolic information for program1
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for librtc.so
Reading symbolic information for libc.so.1
Reading symbolic information for libdl.so.1
Reading symbolic information for libc_psr.so.1
(dbx) check -all
access checking - ON
memuse checking - ON
(dbx) dbxenv follow_fork_mode child
(dbx) run
Running: program1
(process id 3885)
Enabling Error Checking... done
```

RTC reports first error in the parent, `program1`

```
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff110
    which is 104 bytes above the current stack pointer
Variable is 'parent_j'
Current function is main
  11       parent_i = parent_j;
(dbx) cont
dbx: warning: Fork occurred; error checking disabled in parent
detaching from process 3885
Attached to process 3886
```

Because `follow_fork_mode` *is set to* `child`*, when the fork occurs error checking is switched from the parent to the child process*

```
stopped in _fork at 0xef6b6040
0xef6b6040: _fork+0x0008:bgeu    _fork+0x30
Current function is main
   13       child_pid = fork();
parent: child's pid = 3886
(dbx) cont
child: In child
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff108
    which is 96 bytes above the current stack pointer
```

RTC reports an error in the child

```
Variable is 'child_j'
Current function is main
   22   child_i = child_j;
```

```
 (dbx) cont
 dbx: process 3886 about to exec("./program2")
 dbx: program "./program2" just exec'ed
 dbx: to go back to the original program use "debug $oprog"
 Reading symbolic information for program2
 Skipping ld.so.1, already read
 Skipping librtc.so, already read
 Skipping libc.so.1, already read
 Skipping libdl.so.1, already read
 Skipping libc_psr.so.1, already read
```

When the exec of `program2` *occurs, the RTC settings are inherited by* `program2` *so access and memory use checking are enabled for that process*

```
 Enabling Error Checking... done
 stopped in main at line 8 in file "program2.c"
     8       printf ("program2: pid = %d\n", getpid());
(dbx) cont
 program2: pid = 3886
 Read from uninitialized (rui):
 Attempting to read 4 bytes at address 0xeffff13c
     which is 100 bytes above the current stack pointer
```

RTC reports an access error in the executed program, `program2`

```
 Variable is 'program2_j'
 Current function is main
     9       program2_i = program2_j;
 (dbx) cont
 Checking for memory leaks...
```

RTC prints a memory use and memory leak report for the process that exited while under RTC control, `program2`

```
Actual leaks report   (actual leaks:      1  total size:   8
 bytes)

 Total  Num of  Leaked      Allocation call stack
 Size   Blocks  Block
                         Address
======  ====== ========== ====================================
       8       1    0x20c50  main
 Possible leaks report  (possible leaks:   0  total size:   0
 bytes)

 execution completed, exit code is 0
```

Using Runtime Checking on an Attached Process

Runtime checking works on an attached process with the exception that RUI cannot be detected if the affected memory has already been allocated. However, the process must have `rtcaudit.so` preloaded when it starts. If the process to which you are attaching is a 64-bit SPARC V9 process, use the sparcv9 `rtcaudit.so`. If the product is installed in `/opt`, `rtcaudit.so` is at:

`/opt/SUNWspro/lib/v9/rtcaudit.so` for SPARC V9

`/opt/SUNWspro/lib` for all other platforms

To preload `librtc.so`:

```
% setenv LD_AUDIT path-to-rtcaudit/rtcaudit.so
```

Set `LD_AUDIT` to preload `rtcaudit.so` only when needed; do not keep it loaded all the time. For example:

```
% setenv LD_AUDIT...
% start-your-application
% unsetenv LD_AUDIT
```

Once you attach to the process, you can enable runtime checking.

If the program you want to attach to is forked or executed from some other program, you need to set `LD_AUDIT` for the main program (which will fork). The setting of `LD_AUDIT` is inherited across forks and execution.

Some versions of the Solaris Operating System support `LD_AUDIT_32` and `LD_AUDIT_64`, which affect only 32-bit programs and 64-bit programs, respectively. See the *Linker and Libraries Guide* for the version of the Solaris Operating System you are running to determine if these variables are supported.

Using Fix and Continue With Runtime Checking

You can use runtime checking along with fix and continue to isolate and fix programming errors rapidly. Fix and continue provides a powerful combination that can save you a lot of debugging time. Here is an example:.

```
% cat -n bug.c
     1 #include stdio.h
     2 char *s = NULL;
     3
     4 void
     5 problem()
     6 {
     7      *s = 'c';
     8 }
     9
    10 main()
    11 {
    12      problem();
    13      return 0;
    14 }
% cat -n bug-fixed.c
     1 #include stdio.h
     2 char *s = NULL;
     3
     4 void
     5 problem()
     6 {
     7
     8      s = (char *)malloc(1);
     9      *s = 'c';
    10 }
       11
    12 main()
    13 {
    14      problem();
    15      return 0;
    16 }
yourmachine46: cc -g bug.c
yourmachine47: dbx -C a.out
```

```
Reading symbolic information for a.out
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for librtc.so
Reading symbolic information for libc.so.1
Reading symbolic information for libintl.so.1
Reading symbolic information for libdl.so.1
Reading symbolic information for libw.so.1
(dbx) check -access
access checking - ON
(dbx) run
Running: a.out
(process id 15052)
Enabling Error Checking... done
Write to unallocated (wua):
Attempting to write 1 byte through NULL pointer
Current function is problem
    7       *s = 'c';
(dbx) pop
stopped in main at line 12 in file "bug.c"
   12       problem();
(dbx) #at this time we would edit the file; in this example just
copy the correct version
(dbx) cp bug-fixed.c bug.c
(dbx) fix
fixing "bug.c" ......
pc moved to "bug.c":14
stopped in main at line 14 in file "bug.c"
   14       problem();
(dbx) cont

execution completed, exit code is 0
(dbx) quit
The following modules in `a.out' have been changed (fixed):
bug.c
Remember to remake program.
```

For more information on using fix and continue, see Chapter 10.

Runtime Checking Application Programming Interface

Both leak detection and access checking require that the standard heap management routines in the shared library `libc.so` be used so that runtime checking can keep track of all the allocations and deallocations in the program. Many applications write their own memory management routines either on top of the `malloc()` or `free()` `function` or stand-alone. When you use your own allocators (referred to as *private allocators*), runtime checking cannot automatically track them; thus you do not learn of leak and memory access errors resulting from their improper use.

However, runtime checking provides an API for the use of private allocators. This API allows the private allocators the same treatment as the standard heap allocators. The API itself is provided in the header file `rtc_api.h` and is distributed as a part of Sun Studio software. The man page `rtc_api`(3x) details the runtime checking API entry points.

Some minor differences might exist with runtime checking access error reporting when private allocators do not use the program heap. The error report will not include the allocation item.

Using Runtime Checking in Batch Mode

The `bcheck` utility is a convenient batch interface to the runtime checking feature of `dbx`. It runs a program under `dbx` and by default, places the runtime checking error output in the default file `program.errs`.

The `bcheck` utility can perform memory leak checking, memory access checking, memory use checking, or all three. Its default action is to perform only leak checking. See the `bcheck`(1) man page for more details on its use.

`bcheck` Syntax

The syntax for `bcheck` is:

```
bcheck [-V] [-access | -all | -leaks | -memuse] [-o logfile] [-q]
[-s script] program [args]
```

Use the -o *logfile* option to specify a different name for the logfile. Use the -s *script* option before executing the program to read in the dbx commands contained in the file *script*. The *script* file typically contains commands like suppress and dbxenv to tailor the error output of the bcheck utility.

The -q option makes the bcheck utility completely quiet, returning with the same status as the program. This option is useful when you want to use the bcheck utility in scripts or makefiles.

bcheck Examples

To perform only leak checking on hello, type:

```
bcheck hello
```

To perform only access checking on mach with the argument 5, type:

```
bcheck -access mach 5
```

To perform memory use checking on cc quietly and exit with normal exit status, type:

```
bcheck -memuse -q cc -c prog.c
```

The program does not stop when runtime errors are detected in batch mode. All error output is redirected to your error log file logfile. The program stops when breakpoints are encountered or if the program is interrupted.

In batch mode, the complete stack backtrace is generated and redirected to the error log file. The number of stack frames can be controlled using the dbx environment variable stack_max_size.

If the file logfile already exists, bcheck erases the contents of that file before it redirects the batch output to it.

Enabling Batch Mode Directly From dbx

You can also enable a batch-like mode directly from dbx by setting the dbx environment variables rtc_auto_continue and rtc_error_log_file_name (see "Setting dbx Environment Variables" on page 66).

If `rtc_auto_continue` is set to `on`, runtime checking continues to find errors and keeps running automatically. It redirects errors to the file named by the `dbx` environment variable `rtc_error_log_file_name`. (See "Setting `dbx` Environment Variables" on page 66.) The default log file name is `/tmp/dbx.errlog.`*uniqueid*. To redirect all errors to the terminal, set the `rtc_error_log_file_name` environment variable to `/dev/tty`.

By default, `rtc_auto_continue` is set to off.

Troubleshooting Tips

After error checking has been enabled for a program and the program is run, one of the following errors may be detected:

```
librtc.so and dbx version mismatch; Error checking disabled
```

This error can occur if you are using runtime checking on an attached process and have set `LD_AUDIT` to a version of `rtcaudit.so` other than the one shipped with your Sun Studio `dbx` image. To fix this, change the setting of `LD_AUDIT`.

```
patch area too far (8mb limitation); Access checking disabled
```

Runtime checking was unable to find patch space close enough to a loadobject for access checking to be enabled. See "Runtime Checking's 8 Megabyte Limit" next.

Runtime Checking's 8 Megabyte Limit

The 8 megabyte limit described below no longer applies on hardware based on UltraSPARC processors, on which `dbx` has the ability to invoke a trap handler instead of using a branch. The transfer of control to a trap handler is up to 10 times slower but does not suffer from the 8 megabyte limit. Traps are used automatically, as necessary, as long as the hardware is based on UltraSPARC processors. You can check your hardware by using the system command `isalist` and checking that the result contains the string `sparcv8plus`. The `rtc -showmap` command (see "`rtc -showmap` Command" on page 365) displays a map of instrument types sorted by address.

When access checking is enabled, `dbx` replaces each load and store instruction with a branch instruction that branches to a patch area. This branch instruction has an 8 megabyte range. This means that if the debugged program has used up all the address space within 8 megabytes of the particular load or store instruction being replaced, no place exists to put the patch area.

If runtime checking cannot intercept all loads and stores to memory, it cannot provide accurate information and so disables access checking completely. Leak checking is unaffected.

`dbx` internally applies some strategies when it runs into this limitation and continues if it can rectify this problem. In some cases `dbx` cannot proceed; when this happens, it turns off access checking after printing an error message.

If you encounter this 8 megabyte limit, try the following workarounds.

1. Try using 32-bit SPARC-V8 instead of 64-bit SPARC-V9

   If you encounter the 8 megabyte problem with an application that is compiled with the `-xarch=v9` option, try doing your memory testing on a 32-bit version of the application. Because the 64-bit addresses require longer patch instruction sequences, using 32-bit addresses can alleviate the 8 megabyte problem. If this is not a good workaround, the following methods can be used on both 32-bit and 64-bit programs.

2. Try adding patch area object files.

   You can use the `rtc_patch_area` shell script to create special `.o` files that can be linked into the middle of a large executable or shared library to provide more patch space. See the `rtc_patch_area`(1) man page.

   When `dbx` reaches the 8 megabyte limit, it tells you which load object was too large (the main program, or a shared library) and it prints out the total patch space needed for that load object.

   For the best results, the special patch object files should be evenly spaced throughout the executable or shared library, and the default size (8 megabytes) or smaller should be used. Also, do not add more than 10-20% more patch space than `dbx` says it requires. For example, if dbx says that it needs 31 megabytes for `a.out`, then add four object files created with the `rtc_patch_area` script, each one 8 megabytes in size, and space them approximately evenly throughout the executable.

   When `dbx` finds explicit patch areas in an executable, it prints the address ranges spanned by the patch areas, which can help you to place them correctly on the link line.

3. Try dividing the large load object into smaller load objects.

Split up the object files in your executable or your large library into smaller groups of object files. Then link them into smaller parts. If the large file is the executable, then split it up into a smaller executable and a series of shared libraries. If the large file is a shared library, then rearrange it into a set of smaller libraries.

This technique allows `dbx` to find space for patch code in between the different shared objects.

4. Try adding a "pad" `.so` file.

   This should only be necessary if you are attaching to a process after it has started up.

   The runtime linker might place libraries so close together that patch space cannot be created in the gaps between the libraries. When `dbx` starts up the executable with runtime checking turned on, it asks the runtime linker to place an extra gap between the shared libraries, but when attaching to a process that was not started by `dbx` with runtime checking enabled, the libraries might be too close together.

   If the runtime libraries are too close together, (and if it is not possible to start the program using `dbx`) then you can try creating a shared library using the `rtc_patch_area` script and linking it into your program between the other shared libraries. See the `rtc_patch_area`(1) man page for more details.

Runtime Checking Errors

Errors reported by runtime checking generally fall in two categories. Access errors and leaks.

Access Errors

When access checking is turned on, runtime checking detects and reports the following types of errors.

Bad Free (`baf`) Error

Problem: Attempt to free memory that has never been allocated.

Possible causes: Passing a non-heap data pointer to `free()` or `realloc()`.

Example:

```
char a[4];
```

```
char *b = &a[0];

free(b);                          /* Bad free (baf) */
```

Duplicate Free (`duf`) Error

Problem: Attempt to free a heap block that has already been freed.

Possible causes: Calling `free()` more than once with the same pointer. In C++, using the delete operator more than once on the same pointer.

Example:

```
char *a = (char *)malloc(1);
free(a);
free(a);                          /* Duplicate free (duf) */
```

Misaligned Free (`maf`) Error

Problem: Attempt to free a misaligned heap block.

Possible causes: Passing an improperly aligned pointer to `free()` or `realloc()`; changing the pointer returned by `malloc`.

Example:

```
char *ptr = (char *)malloc(4);
ptr++;
free(ptr);                        /* Misaligned free */
```

Misaligned Read (`mar`) Error

Problem: Attempt to read data from an address without proper alignment.

Possible causes: Reading 2, 4, or 8 bytes from an address that is not half-word-aligned, word-aligned, or double-word-aligned, respectively.

Example:

```
char *s = "hello world";
int *i = (int *)&s[1];
int j;

j = *i;                           /* Misaligned read (mar) */
```

Misaligned Write (maw) Error

Problem: Attempt to write data to an address without proper alignment.

Possible causes: Writing 2, 4, or 8 bytes to an address that is not half-word-aligned, word-aligned, or double-word-aligned, respectively.

Example:

```
char *s = "hello world";
int *i = (int *)&s[1];


*i = 0;                          /* Misaligned write (maw) */
```

Out of Memory (oom) Error

Problem: Attempt to allocate memory beyond physical memory available.

Cause: Program cannot obtain more memory from the system. Useful in locating problems that occur when the return value from `malloc()` is not checked for `NULL`, which is a common programming mistake.

Example:

```
char *ptr = (char *)malloc(0x7fffffff);
/* Out of Memory (oom), ptr == NULL */
```

Read From Unallocated Memory (rua) Error

Problem: Attempt to read from nonexistent, unallocated, or unmapped memory.

Possible causes: A stray pointer, overflowing the bounds of a heap block or accessing a heap block that has already been freed.

Example:

```
char c, *a = (char *)malloc(1);
c = a[1];        /* Read from unallocated memory (rua) */
```

Read From Uninitialized Memory (rui) Error

Problem: Attempt to read from uninitialized memory.

Possible causes: Reading local or heap data that has not been initialized.

Example:

```
foo()
{   int i, j;
    j = i;        /* Read from uninitialized memory (rui) */
}
```

Write to Read-Only Memory (wro) Error

Problem: Attempt to write to read-only memory.

Possible causes: Writing to a text address, writing to a read-only data section (.rodata), or writing to a page that mmap has made read-only.

Example:

```
foo()
{   int *foop = (int *) foo;
    *foop = 0;               /* Write to read-only memory (wro) */
}
```

Write to Unallocated Memory (wua) Error

Problem: Attempt to write to nonexistent, unallocated, or unmapped memory.

Possible causes: A stray pointer, overflowing the bounds of a heap block, or accessing a heap block that has already been freed.

Example:

```
char *a = (char *)malloc(1);
a[1] = '\0';             /* Write to unallocated memory (wua) */
```

Memory Leak Errors

With leak checking turned on, runtime checking reports the following types of errors.

Address in Block (aib) Error

Problem: A possible memory leak. There is no reference to the start of an allocated block, but there is at least one reference to an address within the block.

Possible causes: The only pointer to the start of the block is incremented.

Example;

```
char *ptr;
main()
{
   ptr = (char *)malloc(4);
   ptr++;        /* Address in Block */
}
```

Address in Register (`air`) Error

Problem: A possible memory leak. An allocated block has not been freed, and no reference to the block exists anywhere in program memory, but a reference exists in a register.

Possible causes: This can occur legitimately if the compiler keeps a program variable only in a register instead of in memory. The compiler often does this for local variables and function parameters when optimization is turned on. If this error occurs when optimization has not been turned on, it is likely to be an actual memory leak. This can occur if the only pointer to an allocated block goes out of scope before the block is freed.

Example:

```
if (i == 0) {
       char *ptr = (char *)malloc(4);
       /* ptr is going out of scope */
}
  /* Memory Leak or Address in Register */
```

Memory Leak (`mel`) Error

Problem: An allocated block has not been freed, and no reference to the block exists anywhere in the program.

Possible causes: Program failed to free a block no longer used.

Example:

```
char *ptr;
    ptr = (char *)malloc(1);
    ptr = 0;
/* Memory leak (mel) */
```

CHAPTER 10

Fixing and Continuing

Using the `fix` command lets you recompile edited native source code quickly without stopping the debugging process. You cannot use the `fix` command to recompile Java code.

Note – The `fix` command is not available on Linux platforms.

This chapter is organized into the following sections:

Using Fix and Continue

The fix and continue feature lets you modify and recompile a native source file and continue executing without rebuilding the entire program. By updating the `.o` files and splicing them into your program, you don't need to relink.

The advantages of using fix and continue are:

- You do not have to relink the program.
- You do not have to reload the program for debugging.
- You can resume running the program from the fix location.

Note – Do not use the `fix` command if a build is in process.

How Fix and Continue Operates

Before using the `fix` command you must edit the source in the editor window. (See "Modifying Source Using Fix and Continue" on page 172 for the ways you can modify your code). After saving changes, type `fix`. For information on the `fix` command, see "`fix` Command" on page 332.

Once you have invoked the `fix` command, `dbx` calls the compiler with the appropriate compiler options. The modified files are compiled and shared object (`.so`) files are created. Semantic tests are done by comparing the old and new files.

The new object file is linked to your running process using the runtime linker. If the function on top of the stack is being fixed, the new stopped in function is the beginning of the same line in the new function. All the breakpoints in the old file are moved to the new file.

You can use fix and continue on files that have been compiled with or without debugging information, but there are some limitations in the functionality of the `fix` command and the `cont` command for files originally compiled without debugging information. See the `-g` option description in "`fix` Command" on page 332 for more information.

You can fix shared objects (`.so`) files, but they must be opened in a special mode. You can use either `RTLD_NOW|RTLD_GLOBAL` or `RTLD_LAZY|RTLD_GLOBAL` in the call to the `dlopen` function.

Modifying Source Using Fix and Continue

You can modify source code in the following ways when using fix and continue:

- Add, delete, or change lines of code in functions
- Add or delete functions
- Add or delete global and static variables

Problems can occur when functions are mapped from the old file to the new file. To minimize such problems when editing a source file:

- Do not change the name of a function.
- Do not add, delete, or change the type of arguments to a function.
- Do not add, delete, or change the type of local variables in functions currently active on the stack.
- Do not make changes to the declaration of a template or to template instances. Only the body of a C++ template function definition can be modified.

If you make any of the above changes, rebuild your entire program rather than using fix and continue.

Fixing Your Program

You can use the `fix` command to relink source files after you make changes, without recompiling the entire program. You can then continue execution of the program.

To fix your file:

1. **Save the changes to your source.**
2. **Type `fix` at the `dbx` prompt.**

Although you can do an unlimited number of fixes, if you have done several fixes in a row, consider rebuilding your program. The `fix` command changes the program image in memory, but not on the disk. As you do more fixes, the memory image gets out of sync with what is on the disk.

The `fix` command does not make the changes within your executable file, but only changes the `.o` files and the memory image. Once you have finished debugging a program, you must rebuild your program to merge the changes into the executable. When you quit debugging, a message reminds you to rebuild your program.

If you invoke the `fix` command with an option other than `-a` and without a file name argument, only the current modified source file is fixed.

When `fix` is invoked, the current working directory of the file that was current at the time of compilation is searched before executing the compilation line. There might be problems locating the correct directory due to a change in the file system structure from compilation time to debugging time. To avoid this problem, use the command `pathmap`, which creates a mapping from one path name to another. Mapping is applied to source paths and object file paths.

Continuing After Fixing

You can continue executing using the `cont` command (see "`cont` Command" on page 316).

Before resuming program execution, be aware of the following conditions that determine the effect of your changes.

Changing an Executed Function

If you made changes in a function that has already executed, the changes have no effect until:

- You run the program again
- That function is called the next time

If your modifications involve more than simple changes to variables, use the `fix` command, then the `run` command. Using the `run` command is faster because it does not relink the program.

Changing a Function Not Yet Called

If you have made changes in a function not yet called, the changes will be in effect when that function is called.

Changing a Function Currently Being Executed

If you have made changes to the function currently being executed, the impact of the `fix` command depends on where the change is relative to the stopped in function:

- If the change is in code that has already been executed, the code is not re-executed. Execute the code by popping the current function off the stack (see "`pop` Command" on page 357) and continuing from where the changed function is called. You need to know your code well enough to determine whether the function has side effects that can't be undone (for example, opening a file).
- If the change is in code that is yet to be executed, the new code is run.

Changing a Function Presently on the Stack

If you have made changes to a function presently on the stack, but not to the stopped in function, the changed code is not used for the present call of that function. When the stopped in function returns, the old versions of the function on the stack are executed.

There are several ways to solve this problem:

- Use the `pop` command to pop the stack until all changed functions are removed from the stack. You need to know your code to be sure that no problems are created.
- Use the `cont at` *line_number* command to continue from another line.
- Manually repair data structures (use the `assign` command) before continuing.
- Rerun the program using the `run` command.

If there are breakpoints in modified functions on the stack, the breakpoints are moved to the new versions of the functions. If the old versions are executed, the program does not stop in those functions.

Changing Variables After Fixing

Changes made to global variables are not undone by the `pop` command or the `fix` command. To reassign correct values to global variables manually, use the `assign` command. (See "`assign` Command" on page 299.)

The following example shows how a simple bug can be fixed. The application gets a segmentation violation in line 6 when trying to dereference a NULL pointer.

```
dbx[1] list 1,$
       1   #include <stdio.h>
       2
       3   char *from = "ships";
       4   void copy(char *to)
       5   {
       6       while ((*to++ = *from++) != '\0');
       7       *to = '\0';
       8   }
       9
       10  main()
       11  {
       12      char buf[100];
       13
       14      copy(0);
       15      printf("%s\n", buf);
       16      return 0;
       17  }
(dbx) run
Running: testfix
(process id 4842)
signal SEGV (no mapping at the fault address) in copy at line 6
in file "testfix.cc"
       6       while ((*to++ = *from++) != '\0');
```

Change line 14 to `copy` to `buf` instead of `0` and save the file, then do a fix:

```
       14      copy(buf);  <=== modified line
(dbx) fix
fixing "testfix.cc" .....
pc moved to "testfix.cc":6
stopped in copy at line 6 in file "testfix.cc"
       6       while ((*to++ = *from++) != '\0')
```

If the program is continued from here, it still gets a segmentation fault because the zero-pointer is still pushed on the stack. Use the `pop` command to pop one frame of the stack:

```
(dbx) pop
stopped in main at line 14 in file "testfix.cc"
      14 copy(buf);
```

If the program is continued from here, it runs, but does not print the correct value because the global variable `from` has already been incremented by one. The program would print `hips` and not `ships`. Use the `assign` command to restore the global variable and then use the `cont command`. Now the program prints the correct string:

```
(dbx) assign from = from-1
(dbx) cont
ships
```

Modifying a Header File

Sometimes it may be necessary to modify a header (`.h`) file as well as a source file. To be sure that the modified header file is accessed by all source files in the program that include it, you must give as an argument to the `fix` command a list of all the source files that include that header file. If you do not include the list of source files, only the primary source file is recompiled and only it includes the modified version of the header file. Other source files in the program continue to include the original version of that header file.

Fixing C++ Template Definitions

C++ template definitions can be fixed directly. Fix the files with the template instances instead. You can use the `-f` option to overwrite the date-checking if the template definition file has not changed. `dbx` looks for template definition `.o` files in the default repository directory `SunWS_cache`. The `-ptr` compiler option is not supported by the `fix` command in `dbx`.

CHAPTER 11

Debugging Multithreaded Applications

dbx can debug multithreaded applications that use either Solaris threads or POSIX threads. With dbx, you can examine stack traces of each thread, resume all threads, step or next a specific thread, and navigate between threads.

dbx recognizes a multithreaded program by detecting whether it utilizes libthread.so. The program will use libthread.so either by explicitly being compiled with -lthread or -mt, or implicitly by being compiled with -lpthread.

This chapter describes how to find information about and debug threads using the dbx thread commands.

This chapter is organized into the following sections:

Understanding Multithreaded Debugging

When it detects a multithreaded program, dbx tries to load libthread_db.so, a special system library for thread debugging located in /usr/lib.

dbx is synchronous; when any thread or lightweight process (LWP) stops, all other threads and LWPs sympathetically stop. This behavior is sometimes referred to as the "stop the world" model.

Note – For information on multithreaded programming and LWPs, see the Solaris *Multithreaded Programming Guide*.

Thread Information

The following thread information is available in `dbx`:

```
(dbx) threads
    t@1 a l@1  ?()  running   in main()
    t@2      ?() asleep on 0xef751450  in_swtch()
    t@3 b l@2  ?()  running in sigwait()
    t@4     consumer()  asleep on 0x22bb0 in _lwp_sema_wait()
  *>t@5 b l@4 consumer()  breakpoint     in Queue_dequeue()
    t@6 b l@5 producer()     running       in _thread_start()
(dbx)
```

For native code, each line of information is composed of the following:

- The * (asterisk) indicates that an event requiring user attention has occurred in this thread. Usually this is a breakpoint.

  An 'o' instead of an asterisk indicates that a `dbx` internal event has occurred.
- The > (arrow) denotes the current thread.
- `t@`*number*, the thread id, refers to a particular thread. The *number* is the `thread_t` value passed back by `thr_create`.
- `b l@`*number* or `a l@`*number* means the thread is bound to or active on the designated LWP, meaning the thread is actually runnable by the operating system.
- The “Start function” of the thread as passed to `thr_create`. A `?()` means that the start function is not known.
- The thread state (See TABLE 11-1 for descriptions of the thread states.)
- The function that the thread is currently executing.

For Java code, each line of information is composed of the following:

- `t@`*number*, a `dbx`-style thread ID
- The thread state (See TABLE 11-1 for descriptions of the thread states.)
- The thread name in single quotation marks
- A number indicating the thread priority

TABLE 11-1 Thread and LWP States

Thread and LWP States	Description
suspended	The thread has been explicitly suspended.
runnable	The thread is runnable and is waiting for an LWP as a computational resource.
zombie	When a detached thread exits (`thr_exit()`), it is in a zombie state until it has rejoined through the use of `thr_join()`. `THR_DETACHED` is a flag specified at thread creation time (`thr_create()`). A non-detached thread that exits is in a zombie state until it has been reaped.
asleep on *syncobj*	Thread is blocked on the given synchronization object. Depending on what level of support `libthread` and `libthread_db` provide, *syncobj* might be as simple as a hexadecimal address or something with more information content.
active	The thread is active on an LWP, but `dbx` cannot access the LWP.
unknown	`dbx` cannot determine the state.
lwpstate	A bound or active thread state has the state of the LWP associated with it.
running	LWP was running but was stopped in synchrony with some other LWP.
syscall *num*	LWP stopped on an entry into the given system call #.
syscall return *num*	LWP stopped on an exit from the given system call #.
job control	LWP stopped due to job control.
LWP suspended	LWP is blocked in the kernel.
single stepped	LWP has just completed a single step.
breakpoint	LWP has just hit a breakpoint.
fault *num*	LWP has incurred the given fault #.
signal *name*	LWP has incurred the given signal.
process sync	The process to which this LWP belongs has just started executing.
LWP death	LWP is in the process of exiting.

Viewing the Context of Another Thread

To switch the viewing context to another thread, use the `thread` command. The syntax is:

```
thread [-blocks] [-blockedby] [-info] [-hide] [-unhide] [-
suspend] [-resume] thread_id
```

To display the current thread, type:

```
thread
```

To switch to thread *thread_id*, type:

```
thread thread_id
```

For more information on the `thread` command, see "`thread` Command" on page 383.

Viewing the Threads List

To view the threads list, use the `threads` command The syntax is:

```
threads [-all} [-mode [all|filter] [auto|manual]]
```

To print the list of all known threads, type:

```
threads
```

To print threads normally not printed (zombies), type:

```
threads -all
```

For an explanation of the threads list, see "Thread Information" on page 178.

For more information on the `threads` command, see "`threads` Command" on page 385.

Resuming Execution

Use the `cont` command to resume program execution. Currently, threads use synchronous breakpoints, so all threads resume execution.

Understanding LWP Information

Normally, you need not be aware of LWPs. There are times, however, when thread level queries cannot be completed. In these cases, use the `lwps` command to show information about LWPs.

```
(dbx) lwps
    l@1 running in main()
    l@2 running in sigwait()
    l@3 running in _lwp_sema_wait()
  *>l@4 breakpoint in Queue_dequeue()
    l@5 running in _thread_start()
(dbx)
```

Note – The `lwps` command is not available on Linux platforms.

Each line of the LWP list contains the following:

- The * (asterisk) indicates that an event requiring user attention has occurred in this LWP.
- The arrow denotes the current LWP.
- l@*number* refers to a particular LWP.
- The next item represents the LWP state.
- in *function_name()* identifies the function that the LWP is currently executing.

CHAPTER 12

Debugging OpenMP Programs

The OpenMP™ application programming interface (API) is a portable, parallel programming model for shared memory multiprocessor architectures, developed in collaboration with a number of computer vendors. Support for debugging Fortran and C OpenMP programs with `dbx` is based on the general multi-threaded debugging features of `dbx`. All of the `dbx` commands that operate on threads and LWPs can be used for OpenMP debugging. `dbx` does not support asynchronous thread control in OpenMP debugging.

This chapter is organized in to the following sections:

See the *OpenMP API Users Guide* for information on the directives, run-time library routines, and environment variables comprising the OpenMP Version 2.0 Application Program Interfaces, as implemented by the Sun Studio Fortran 95 and C compilers.

Note – OpenMP debugging is available only on SPARC platforms only. It is not available on Solaris OS x86 platforms or Linux platforms.

How Compilers Transform OpenMP Code

To better describe OpenMP debugging, it is helpful to understand how OpenMP code is transformed by the compilers. Consider the following Fortran example:

```
1       program example
2           integer i, n
3           parameter (n = 1000000)
4           real sum, a(n)
5
6           do i = 1, n
7           a(i) = i*i
8           end do
9
10          sum = 0
11
12      !$OMP PARALLEL DO DEFAULT(PRIVATE), SHARED(a, sum)
13
14          do i = 1, n
15          sum = sum + a(i)
16          end do
17
18      !$OMP END PARALLEL DO
19
20          print*, sum
21          end program example
```

The code in line 12 through line 18 is a parallel region. The f95 compiler converts this section of code to an outlined subroutine that will be called from the OpenMP runtime library. This outlined subroutine has an internally generated name, in this case `_$d1A12.MAIN_`. The f95 compiler then replaces the code for the parallel region with a call to the OpenMP runtime library and passes the outlined subroutine as one of its arguments. The OpenMP runtime library handles all the thread-related issues and dispatches slave threads that execute the outlined subroutine in parallel. The C compiler works in the same way.

When debugging an OpenMP program, the outlined subroutine is treated by `dbx` as any other function, with the exception that you cannot explicitly set a breakpoint in that function by using its internally generated name.

dbx Functionality Available for OpenMP Code

In addition to the usual functionality for debugging multithreaded programs, dbx allows you to do the following in an OpenMP program:

- **Single step into a parallel region**. Because a parallel region is outlined and called from the OpenMP runtime library, a single step of execution actually involves several layers of runtime library calls that are executed by slave threads created for this purpose. When you single step into the parallel region, the first thread that reaches the breakpoint causes the program to stop. This thread might be a slave thread rather than the master thread that initiated the stepping.

  For example, refer to the Fortran code in"How Compilers Transform OpenMP Code" on page 184, and assume that master thread `t@1` is at line 10. You single step into line 12, and slave threads `t@2`, `t@3`, and `t@4` are created to execute the runtime library calls. Thread `t@3` reaches the breakpoint first and causes the program execution to stop. So the single step that was initiated by thread `t@1`ends on thread `t@3`. This behavior is different from normal stepping in which you are usually on the same thread after the single step as before.

- **Print shared, private, and threadprivate variables.** `dbx` can print all shared, private, and threadprivate variables. If you try to print a threadprivate variable outside of a parallel region, the master thread's copy is printed. The `whatis` command does not tell you whether a variable is shared, private, or threadprivate.

Using Stack Traces With OpenMP Code

When execution is stopped in parallel region, a `where` command shows a stack trace that contains the outlined subroutine as well as several runtime library calls. Using the Fortran example from "How Compilers Transform OpenMP Code" on page 184, and stopping execution at line 15, the `where` command produces the following stack trace.

```
[t@4 l@4]: where
current thread: t@4
=>[1] _$d1A12.MAIN_(), line 15 in "example.f90"
[2] __mt_run_my_job_(0x45720, 0xff82ee48, 0x0, 0xff82ee58, 0x0,
0x0), at 0x16860
[3] __mt_SlaveFunction_(0x45720, 0x0, 0xff82ee48, 0x0, 0x455e0,
0x1), at 0x1aaf0
```

The top frame on the stack is the frame of the outlined function. Even though the code is outlined, the source line number still maps back to 15. The other two frames are for runtime library routines.

When execution is stopped in a parallel region, a `where` command from a slave thread does not have a stack traceback to its parent thread, as shown in the above example. A `where` command from the master thread, however, has a full traceback:

```
[t@4 l@4]: thread t@1
t@1 (l@1) stopped in _$d1A12.MAIN_ at line 15 in file "example.f90"
15            sum = sum + a(i)
[t@1 l@1]: where
current thread: t@1
=>[1] _$d1A12.MAIN_(), line 15 in "example.f90"
[2] __mt_run_my_job_(0x41568, 0xff82ee48, 0x0, 0xff82ee58, 0x0,
0x0), at 0x16860
[3] __mt_MasterFunction_(0x1, 0x0, 0x6, 0x0, 0x0, 0x40d78), at
0x16150
[4] MAIN(), line 12 in "example.f90"
```

If the number of threads is not large, you might be able to determine how execution reached the breakpoint in a slave thread by using the `threads` command (see "`threads` Command" on page 385) to list all the threads, and then switch to each thread to determine which one is the master thread.

Using the dump Command on OpenMP Code

When execution is stopped in a parallel region, a `dump` command may print more than one copy of private variables. In the following example, the `dump` command prints two copies of the variable `i`:

```
[t@1 l@1]: dump
i = 1
sum = 0.0
a = ARRAY
i = 1000001
```

Two copies of variable `i` are printed because the outlined routine is implemented as a nested function of the hosting routine, and private variables are implemented as local variables of the outlined routine. Since a `dump` command prints all the variables in scope, both the `i` in hosting routine and the `i` in the outlined routine are displayed.

Execution Sequence of OpenMP Code

When you are single stepping inside of a parallel region in an OpenMP program, the execution sequence may not be the same as the source code sequence. This difference in sequence occurs because the code in the parallel region is usually transformed and rearranged by the compiler. Single stepping in OpenMP code is similar to single stepping in optimized code where the optimizer has usually moved code around.

CHAPTER 13

Debugging Child Processes

This chapter describes how to debug a child process. `dbx` has several facilities to help you debug processes that create children using the `fork` (2) and `exec` (2) functions.

This chapter is organized into the following sections:

Attaching to Child Processes

You can attach to a running child process in one of the following ways.

- When starting `dbx`:

```
$ dbx program_name process_id
```

- From the `dbx` command line:

```
(dbx) debug program_name process_id
```

You can substitute *program_name* with the name - (minus sign), so that `dbx` finds the executable associated with the given process ID (*process_id*). After using a `-`, a subsequent `run` command or `rerun` command does not work because `dbx` does not know the full path name of the executable.

You can also attach to a running child process using the Sun WorkShop Debugging window. (See "Attaching to a Running Process" in the Using the Debugging window section of the Sun WorkShop online help.)

Following the `exec` Function

If a child process executes a new program using the `exec(2)` function or one of its variations, the process id does not change, but the process image does. `dbx` automatically takes note of a call to the `exec()` function and does an implicit reload of the newly executed program.

The original name of the executable is saved in `$oprog`. To return to it, use `debug $oprog`.

Following the `fork` Function

If a child process calls the `vfork()`, `fork(1)`, or `fork(2)` function, the process id changes, but the process image stays the same. Depending on how the `dbx` environment variable `follow_fork_mode` is set, `dbx` does one of the following.

Parent	In the traditional behavior, `dbx` ignores the fork and follows the parent.
Child	`dbx` automatically switches to the forked child using the new process ID. All connection to and awareness of the original parent is lost.
Both	This mode is available only when using `dbx` through the Sun Studio IDE.
Ask	You are prompted to choose `parent`, `child`, `both`, or `stop to investigate` whenever `dbx` detects a fork. If you choose `stop`, you can examine the state of the program, then type `cont` to continue; you will be prompted to select which way to proceed.

Interacting With Events

All breakpoints and other events are deleted for any `exec()` or `fork()` process. You can override the deletion for forked processes by setting the `dbx` environment variable `follow_fork_inherit` to `on`, or make the events permanent using the `-perm eventspec` modifier. For more information on using event specification modifiers, see Appendix B.

CHAPTER 14

Working With Signals

This chapter describes how to use `dbx` to work with signals. `dbx` supports the `catch` command, which instructs `dbx` to stop a program when `dbx` detects any of the signals appearing on the catch list.

The `dbx` commands `cont`, `step`, and `next` support the `-sig` *signal_name* option, which lets you resume execution of a program with the program behaving as if it had received the signal specified in the `cont -sig` command.

This chapter is organized into the following sections.

Understanding Signal Events

When a signal is to be delivered to a process that is being debugged, the signal is redirected to `dbx` by the kernel. When this happens, you usually receive a prompt. You then have two choices:

- "Cancel" the signal when the program is resumed—the default behavior of the `cont` command—facilitating easy interruption and resumption with `SIGINT` (Control-C) as shown in FIGURE 14-1.

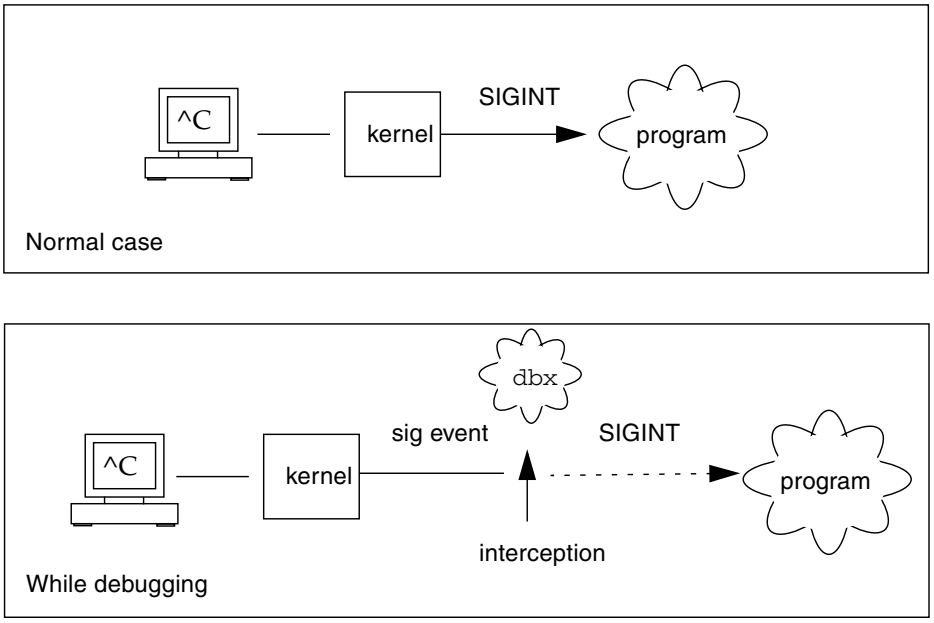


FIGURE 14-1 Intercepting and Cancelling the `SIGINT` Signal

- "Forward" the signal to the process using:

```
cont -sig signal
```

signal can be either a signal name or a signal number.

In addition, if a certain signal is received frequently, you can arrange for `dbx` to forward automatically the signal because you do not want it displayed:

```
ignore signal # "ignore"
```

However, the signal is still forwarded to the process. A default set of signals is automatically forwarded in this manner (see "`ignore` Command" on page 337).

Catching Signals

By default, the catch list contains many of the more than 33 detectable signals. (The numbers depend upon the operating system and version.) You can change the default catch list by adding signals to or removing them from the default catch list.

Note – The list of signal names that `dbx` accepts includes all of those supported by the versions of the Solaris operating environment that `dbx` supports. So `dbx` might accept a signal that is not supported by the version of the Solaris operating environment you are running. For example, `dbx` might accept a signal that is supported by the Solaris 9 Operating System even through you are running the Solaris 7 Operating Environment. For a list of the signals supported by the Solaris operating environment you are running, see the `signal`(3head) man page.

To see the list of signals currently being trapped, type **`catch`** with no *signal* argument.

```
(dbx) catch
```

To see a list of the signals currently being *ignored* by `dbx` when the program detects them, type **`ignore`** with no *signal* argument.

```
(dbx) ignore
```

Changing the Default Signal Lists

You control which signals cause the program to stop by moving the signal names from one list to the other. To move signal names, supply a signal name that currently appears on one list as an argument to the other list.

For example, to move the `QUIT` and `ABRT` signals from the catch list to the ignore list:

```
(dbx) ignore QUIT ABRT
```

Trapping the FPE Signal (Solaris Platforms Only)

Often programmers working with code that requires floating point calculations want to debug exceptions generated in a program. When a floating point exception like overflow or divide by zero occurs, the system returns a reasonable answer as the result for the operation that caused the exception. Returning a reasonable answer lets the program continue executing quietly. Solaris implements the IEEE Standard for Binary Floating Point Arithmetic definitions of reasonable answers for exceptions.

Because a reasonable answer for floating point exceptions is returned, exceptions do not automatically trigger the signal `SIGFPE`. Some integer exceptions, such as dividing an integer by zero and integer overflow do, by default, trigger the signal `SIGFPE`.

To find the cause of an exception, you need to set up a trap handler in the program so that the exception triggers the signal `SIGFPE`. (See `ieee_handler`(3m) man page for an example of a trap handler.)

You can enable a trap using:

- `ieee_handler`
- `fpsetmask` (see the `fpsetmask`(3c) man page)
- `-ftrap` compiler flag (for Fortran 95, see the `f95`(1) man page)

When you set up a trap handler using the `ieee_handler` command, the trap enable mask in the hardware floating point status register is set. This trap enable mask causes the exception to raise the `SIGFPE` signal at run time.

Once you have compiled the program with the trap handler, load the program into `dbx`. Before you can catch the `SIGFPE` signal, you must add `FPE` to the `dbx` signal catch list.

```
(dbx) catch FPE
```

By default, `FPE` is on the ignore list.

Determining Where the Exception Occurred

After adding `FPE` to the catch list, run the program in `dbx`. When the exception you are trapping occurs, the `SIGFPE` signal is raised and `dbx` stops the program. Then you can trace the call stack using the `dbx where` command to help find the specific line number of the program where the exception occurs (see "`where` Command" on page 399).

Determining the Cause of the Exception

To determine the cause of the exception, use the `regs -f` command to display the floating point state register (FSR). Look at the accrued exception (`aexc`) and current exception (`cexc`) fields of the register, which contain bits for the following floating-point exception conditions:

- Invalid operand
- Overflow
- Underflow
- Division by zero
- Inexact result

For more information on the floating-point state register, see Version 8 (for V8) or Version 9 (for V9) of *The SPARC Architecture Manual.* For more discussion and examples, see the *Numerical Computation Guide.*

Sending a Signal in a Program

The `dbx cont` command supports the `-sig` *signal* option, which lets you resume execution of a program with the program behaving as if it had received the system signal *signal*.

For example, if a program has an interrupt handler for `SIGINT` (`^C`), you can type `^C` to stop the application and return control to `dbx`. If you issue a `cont` command by itself to continue program execution, the interrupt handler never executes. To execute the interrupt handler, send the signal, `SIGINT`, to the program:

```
(dbx) cont -sig int
```

The `step`, `next`, and `detach` commands accept `-sig` as well.

Automatically Handling Signals

The event management commands can also deal with signals as events. These two commands have the same effect.

```
(dbx) stop sig signal
(dbx) catch signal
```

Having the signal event is more useful if you need to associate some pre-programmed action.

```
(dbx) when sig SIGCLD {echo Got $sig $signame;}
```

In this case, make sure to first move `SIGCLD` to the ignore list.

```
(dbx) ignore SIGCLD
```

CHAPTER 15

Debugging C++ With dbx

This chapter describes how dbx handles C++ exceptions and debugging C++ templates, including a summary of commands used when completing these tasks and examples with code samples.

This chapter is organized into the following sections:

- Using dbx With C++
- Exception Handling in dbx
- Debugging With C++ Templates

For information on compiling C++ programs, see "Debugging Optimized Code" on page 57.

Using dbx With C++

Although this chapter concentrates on two specific aspects of debugging C++, dbx allows you full functionality when debugging your C++ programs. You can:

- Find out about class and type definitions (see "Looking Up Definitions of Types and Classes" on page 87)
- Print or display inherited data members (see "Printing C++" on page 124)
- Find out dynamic information about an object pointer (see "Printing C++" on page 124)
- Debug virtual functions (see "Calling a Function" on page 98)
- Using runtime type information (see "Printing the Value of a Variable, Expression, or Identifier" on page 124)
- Set breakpoints on all member functions of a class (see "Setting Breakpoints in Member Functions of the Same Class" on page 105)

- Set breakpoints on all overloaded member functions (see "Setting Breakpoints in Member Functions of Different Classes" on page 104)
- Set breakpoints on all overloaded nonmember functions (see "Setting Multiple Breakpoints in Nonmember Functions" on page 105)
- Set breakpoints on all member functions of a particular object (see "Setting Breakpoints in Objects" on page 106)
- Deal with overloaded functions or data members (see "Setting a `stop` Breakpoint in a Function" on page 103)

Exception Handling in `dbx`

A program stops running if an exception occurs. Exceptions signal programming anomalies, such as division by zero or array overflow. You can set up blocks to catch exceptions raised by expressions elsewhere in the code.

While debugging a program, `dbx` enables you to:

- Catch unhandled exceptions before stack unwinding
- Catch unexpected exceptions
- Catch specific exceptions whether handled or not before stack unwinding
- Determine where a specific exception would be caught if it occurred at a particular point in the program

If you give a `step` command after stopping at a point where an exception is thrown, control is returned at the start of the first destructor executed during stack unwinding. If you `step` out of a destructor executed during stack unwinding, control is returned at the start of the next destructor. When all destructors have been executed, a `step` command brings you to the catch block handling the throwing of the exception

Commands for Handling Exceptions

`exception [-d | +d]` Command

Use the `exception` command to display an exception's type at any time during debugging. If you use the `exception` command without an option, the type shown is determined by the setting of the `dbx` environment variable `output_dynamic_type`:

- If it is set to `on`, the derived type is shown.
- If it is set to `off` (the default), the static type is shown.

Specifying the `-d` or `+d` option overrides the setting of the environment variable:

- If you specify `-d`, the derived type is shown.
- If you specify `+d`, the static type is shown.

For more information, see "`exception` Command" on page 330.

`intercept [-a] [-x] [`*typename*`]` Command

You can intercept, or catch, exceptions of a specific type before the stack has been unwound. Use the `intercept` command with no arguments to list the types that are being intercepted. Use `-a` to intercept all exceptions. Use *typename* to add a type to the intercept list. Use `-x` to exclude a particular type from being intercepted.

For example, to intercept all types except `int`, you could type:

```
(dbx) intercept -a
(dbx) intercept -x int
```

For more information, see "`intercept` Command" on page 338.

`unintercept [-a] [-x] [`*typename*`]` Command

Use the `unintercept` command to remove exception types from the intercept list. Use the command with no arguments to list the types that are being intercepted (same as the `intercept` command). Use `-a` to remove all intercepted types from the list. Use *typename* to remove a type from the intercept list. Use `-x` to stop excluding a particular type from being intercepted.

For more information, see "`unintercept` Command" on page 393.

`whocatches` *typename* Command

The `whocatches` command reports where an exception of *typename* would be caught if thrown at the current point of execution. Use this command to find out what would happen if an exception were thrown from the top frame of the stack.

The line number, function name, and frame number of the catch clause that would catch *typename* are displayed. The command returns "*type* `is unhandled`" if the catch point is in the same function that is doing the throw.

For more information, see "`whocatches` Command" on page 402.

Examples of Exception Handling

This example demonstrates how exception handling is done in dbx using a sample program containing exceptions. An exception of type int is thrown in the function bar and is caught in the following catch block.

```
1  #include <stdio.h>
2
3  class c {
4      int x;
5    public:
6      c(int i) { x = i; }
7      ~c() {
8                  printf("destructor for c(%d)\n", x);
9          }
10 };
11
12 void bar() {
13     c c1(3);
14     throw(99);
15 }
16
17 int main() {
18     try {
19         c c2(5);
20         bar();
21         return 0;
22     }
23     catch (int i) {
24         printf("caught exception %d\n", i);
25     }
26 }
```

The following transcript from the example program shows the exception handling features in dbx.

```
(dbx) intercept
-unhandled -unexpected
(dbx) intercept int
<dbx> intercept
-unhandled -unexpected int
(dbx) stop in bar
(2) stop in bar()
(dbx)run
```

```
Running: a.out
(process id 304)
Stopped in bar at line 13 in file "foo.cc"
   13       c c1(3);
(dbx) whocatches int
int is caught at line 24, in function main (frame number 2)
(dbx) whocatches c
dbx: no runtime type info for class c (never thrown or caught)
(dbx) cont
Exception of type int is caught at line 24, in function main (frame
number 4)
stopped in _exdbg_notify_of_throw at 0xef731494
0xef731494: _exdbg_notify_of_throw        :       jmp     %o7 + 0x8
Current function is bar
   14        throw(99);
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8         printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(3)
stopped in c::~c at line 9 in file "foo.cc"
    9       }
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8        printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(5)
stopped in c::~c at line 9 in file "foo.cc"
    9       )
(dbx) step
stopped in main at line 24 in file "foo.cc"
   24           printf("caught exception %d\n", i);
(dbx) step
caught exception 99
stopped in main at line 26 in file "foo.cc"
   26   }
```

Debugging With C++ Templates

dbx supports C++ templates. You can load programs containing class and function templates into dbx and invoke any of the dbx commands on a template that you would use on a class or function, such as:

- Setting breakpoints at class or function template instantiations (see "stop inclass classname Command" on page 208, "stop infunction name Command" on page 209, and "stop in function Command" on page 209)
- Printing a list of all class and function template instantiations (see "whereis name Command" on page 206)
- Displaying the definitions of templates and instances (see "whatis name Command" on page 207)
- Calling member template functions and function template instantiations (see "call function_name (parameters) Command" on page 209)
- Printing values of function template instantiations ("print Expressions" on page 210)
- Displaying the source code for function template instantiations (see "list Expressions" on page 210)

Template Example

The following code example shows the class template Array and its instantiations and the function template square and its instantiations.

```
 1        template<class C> void square(C num, C *result)
 2        {
 3            *result = num * num;
 4        }
 5
 6        template<class T> class Array
 7        {
 8        public:
 9            int getlength(void)
10            {
11                return length;
12            }
13
14            T & operator[](int i)
15            {
```

```
16                return array[i];
17            }
18
19            Array(int l)
20            {
21                length = l;
22                array = new T[length];
23            }
24
25            ~Array(void)
26            {
27                delete [] array;
28            }
29
30        private:
31            int length;
32            T *array;
33        };
34
35        int main(void)
36        {
37            int i, j = 3;
38            square(j, &i);
39
40            double d, e = 4.1;
41            square(e, &d);
42
43            Array<int> iarray(5);
44            for (i = 0; i < iarray.getlength(); ++i)
45            {
46                iarray[i] = i;
47            }
48
49            Array<double> darray(5);
50            for (i = 0; i < darray.getlength(); ++i)
51            {
52                darray[i] = i * 2.1;
53            }
54
55            return 0;
56        }
```

In the example:

- `Array` is a class template
- `square` is a function template
- `Array<int>` is a class template instantiation (template class)
- `Array<int>::getlength` is a member function of a template class
- `square(int, int*)` and `square(double, double*)` are function template instantiations (template functions)

Commands for C++ Templates

Use these commands on templates and template instantiations. Once you know the class or type definitions, you can print values, display source listings, or set breakpoints.

`whereis` *name* Command

Use the `whereis` command to print a list of all occurrences of function or class instantiations for a function or class template.

For a class template:

```
(dbx) whereis Array
member function: 'Array<int>::Array(int)
member function: 'Array<double>::Array(int)
class template instance: 'Array<int>
class template instance: 'Array<double>
class template: 'a.out'template_doc_2.cc'Array
```

For a function template:

```
(dbx) whereis square
function template instance: 'square<int>(__type_0,__type_0*)
function template instance: 'square<double>(__type_0,__type_0*)
```

The `__type_0` parameter refers to the 0th template parameter. A `__type_1` would refer to the next template parameter.

For more information, see "`whereis` Command" on page 401.

`whatis` *name* Command

Use the `whatis` command to print the definitions of function and class templates and instantiated functions and classes.

For a class template:

```
(dbx) whatis -t Array
template<class T> class Array
To get the full template declaration, try 'whatis -t Array<int>';
```

For the class template's constructors:

```
(dbx) whatis Array
More than one identifier 'Array'.
Select one of the following:
 0) Cancel
 1) Array<int>::Array(int)
 2) Array<double>::Array(int>
> 1
Array<int>::Array(int 1);
```

For a function template:

```
(dbx) whatis square
More than one identifier 'square'.
Select one of the following:
 0) Cancel
 1) square<int(__type_0,__type_0*)
 2) square<double>(__type_0,__type_0*)
> 2
void square<double>(double num, double *result);
```

For a class template instantiation:

```
(dbx) whatis -t Array<double>
class Array<double>; {
public:
    int Array<double>::getlength()
    double &Array<double>::operator [](int i);
    Array<double>::Array<double>(int l);
    Array<double>::~Array<double>();
private:
    int length;
    double *array;
};
```

For a function template instantiation:

```
(dbx) whatis square(int, int*)
void square(int num, int *result);
```

For more information, see "whatis Command" on page 396.

stop inclass *classname* Command

To stop in all member functions of a template class:

```
(dbx)stop inclass Array
(2) stop inclass Array
```

Use the stop inclass command to set breakpoints at all member functions of a particular template class:

```
(dbx) stop inclass Array<int>
(2) stop inclass Array<int>
```

For more information, see "stop Command" on page 375 and "inclass classname [-recurse | -norecurse]" on page 279.

`stop infunction` *name* Command

Use the `stop infunction` command to set breakpoints at all instances of the specified function template:

```
(dbx) stop infunction square
(9) stop infunction square
```

For more information, see "`stop` Command" on page 375 and "`infunction` function" on page 279.

`stop in` *function* Command

Use the `stop in` command to set a breakpoint at a member function of a template class or at a template function.

For a member of a class template instantiation:

```
(dbx) stop in Array<int>::Array(int l)
(2) stop in Array<int>::Array(int)
```

For a function instantiation:

```
(dbx) stop in square(double, double*)
(6) stop in square(double, double*)
```

For more information, "`stop` Command" on page 375 and "`in` function" on page 278.

`call` *function_name* (*parameters*) Command

Use the `call` command to explicitly call a function instantiation or a member function of a class template when you are stopped in scope. If `dbx` is unable to determine the correct instance, it displays a numbered list of instances from which you can choose.

```
(dbx) call square(j,&i)
```

For more information, see "`call` Command" on page 302.

`print` Expressions

Use the `print` command to evaluate a function instantiation or a member function of a class template:.

```
(dbx) print iarray.getlength()
iarray.getlength() = 5
```

Use `print` to evaluate the `this` pointer.

```
(dbx) whatis this
class Array<int> *this;
(dbx) print *this
*this = {
    length = 5
    array   = 0x21608
}
```

For more information, see "`print` Command" on page 358.

`list` Expressions

Use the `list` command to print the source listing for the specified function instantiation.

```
(dbx) list square(int, int*)
```

For more information, see "`list` Command" on page 343.

CHAPTER 16

Debugging Fortran Using dbx

This chapter introduces dbx features you might use with Fortran. Sample requests to dbx are also included to provide you with assistance when debugging Fortran code using dbx.

This chapter includes the following topics:

Debugging Fortran

The following tips and general concepts are provided to help you while debugging Fortran programs. For information on debugging Fortran OpenMP code with dbx, see Chapter 12.

Current Procedure and File

During a debug session, dbx defines a procedure and a source file as current. Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, `stop at 5` sets different breakpoints, depending on which file is current.

Uppercase Letters

If your program has uppercase letters in any identifiers, dbx recognizes them. You need not provide case-sensitive or case-insensitive commands, as in some earlier versions.

Fortran 95 and dbx must be in the same case-sensitive or case-insensitive mode:

- Compile and debug in case-insensitive mode without the -U option. The default value of the dbx input_case_sensitive environment variable is then false.

  If the source has a variable named LAST, then in dbx, both the print LAST or print last commands work. Fortran 95 and dbx consider LAST and last to be the same, as requested.

- Compile and debug in case-sensitive mode using -U. The default value of the dbx input_case_sensitive environment variable is then true.

  If the source has a variable named LAST and one named last, then in dbx, print LAST works, but print last does not work. Fortran 95 and dbx distinguish between LAST and last, as requested.

Note – File or directory names are always case-sensitive in dbx, even if you have set the dbx input_case_sensitive environment variable to false.

Sample dbx Session

The following examples use a sample program called my_program.

Main program for debugging, a1.f:

```
      PARAMETER ( n=2 )
      REAL twobytwo(2,2) / 4 *-1 /
      CALL mkidentity( twobytwo, n )
      PRINT *, determinant( twobytwo )
      END
```

Subroutine for debugging, `a2.f`:

```
       SUBROUTINE mkidentity ( array, m )
       REAL array(m,m)
       DO 90 i = 1, m
           DO 20 j = 1, m
                IF ( i .EQ. j ) THEN
               array(i,j) = 1.
                ELSE
               array(i,j) = 0.
                END IF
20         CONTINUE
90     CONTINUE
       RETURN
       END
```

Function for debugging, `a3.f`:

```
       REAL FUNCTION determinant ( a )
       REAL a(2,2)
       determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
       RETURN
       END
```

1. **Compile and link with the `-g` option.**

   You can do this in one or two steps.

   Compile and link in one step, with `-g`:

```
demo% f95 -o my_program -g a1.f a2.f a3.f
```

   Or, compile and link in separate steps:

```
demo% f95 -c -g a1.f a2.f a3.f
demo% f95 -o my_program a1.o a2.o a3.o
```

2. **Start `dbx` on the executable named `my_program`.**

```
demo% dbx my_program
Reading symbolic information…
```

3. **Set a simple breakpoint by typing** `stop in` *subnam*, **where** *subnam* **names a subroutine, function, or block data subprogram.**

   To stop at the first executable statement in a main program.

```
(dbx) stop in MAIN
(2) stop in MAIN
```

   Although `MAIN` must be all uppercase, *subnam* can be uppercase or lowercase.

4. **Type the** `run` **command, which runs the program in the executable files named when you started** `dbx`**.**

```
(dbx) run
Running: my_program
stopped in MAIN at line 3 in file "a1.f"
      3       call mkidentity( twobytwo, n )
```

   When the breakpoint is reached, `dbx` displays a message showing where it stopped—in this case, at line 3 of the `a1.f` file.

5. **To print a value, type the** `print` **command.**

   Print value of `n`:

```
(dbx) print n
n = 2
```

   Print the matrix `twobytwo`; the format might vary:

```
(dbx) print twobytwo
twobytwo =
   (1,1)        -1.0
   (2,1)        -1.0
   (1,2)        -1.0
   (2,2)        -1.0
```

   Print the matrix `array`:

```
(dbx) print array
dbx: "array" is not defined in the current scope
(dbx)
```

   The print fails because `array` is not defined here—only in `mkidentity`.

6. **To advance execution to the next line, type the `next` command.**

   Advance execution to the next line:

```
(dbx) next
stopped in MAIN at line 4 in file "a1.f"
    4         print *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo =
    (1,1)       1.0
    (2,1)       0.0
    (1,2)       0.0
    (2,2)       1.0
(dbx) quit
demo%
```

   The `next` command executes the current source line and stops at the next line. It counts subprogram calls as single statements.

   Compare the `next` command with the `step` command. The `step` command executes the next source line or the next step into a subprogram. If the next executable source statement is a subroutine or function call, then:

   - The `step` command sets a breakpoint at the first source statement of the subprogram.
   - The `next` command sets the breakpoint at the first source statement after the call, but still in the calling program.

7. **To quit `dbx`, type the `quit` command.**

```
(dbx)quit
demo%
```

Debugging Segmentation Faults

If a program gets a segmentation fault (`SIGSEGV`), it references a memory address outside of the memory available to it.

The most frequent causes for a segmentation fault are:

- An array index is outside the declared range.
- The name of an array index is misspelled.
- The calling routine has a `REAL` argument, which the called routine has as `INTEGER`.
- An array index is miscalculated.
- The calling routine has fewer arguments than required.
- A pointer is used before it has been defined.

Using dbx to Locate Problems

Use `dbx` to find the source code line where a segmentation fault has occurred.

Use a program to generate a segmentation fault:

```
demo% cat WhereSEGV.f
       INTEGER a(5)
       j = 2000000
       DO 9 i = 1,5
           a(j) = (i * 10)
9      CONTINUE
       PRINT *, a
       END
demo%
```

Use `dbx` to find the line number of a `dbx` segmentation fault:

```
demo% f95 -g -silent WhereSEGV.f
demo% a.out
Segmentation fault
demo% dbx a.out
Reading symbolic information for a.out
program terminated by signal SEGV (segmentation violation)
(dbx) run
Running: a.out
signal SEGV (no mapping at the fault address)
    in MAIN at line 4 in file "WhereSEGV.f"
    4                   a(j) = (i * 10)
(dbx)
```

Locating Exceptions

If a program gets an exception, there are many possible causes. One approach to locating the problem is to find the line number in the source program where the exception occurred, and then look for clues there.

Compiling with `-ftrap=common` forces trapping on all common exceptions.

To find where an exception occurred:

```
demo% cat wh.f
                call joe(r, s)
                print *, r/s
                end
                subroutine joe(r,s)
                r = 12.
                s = 0.
                return
                end
demo% f95 -g -o wh -ftrap=common wh.f
demo% dbx wh
Reading symbolic information for wh
(dbx) catch FPE
(dbx) run
Running: wh
(process id 17970)
signal FPE (floating point divide by zero) in MAIN at line 2 in
file "wh.f"
   2                    print *, r/s
(dbx)
```

Tracing Calls

Sometimes a program stops with a core dump, and you need to know the sequence of calls that led it there. This sequence is called a *stack trace*.

The `where` command shows where in the program flow execution stopped and how execution reached this point—a *stack trace* of the called routines.

`ShowTrace.f` is a program contrived to get a core dump a few levels deep in the call sequence—to show a stack trace.

Show the sequence of calls, starting at where the execution stopped:

Note the reverse order:

```
demo% f77 -silent -g ShowTrace.f
demo% a.out
MAIN called calc, calc called calcb.
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quil 174% dbx a.out
Execution stopped, line 23
Reading symbolic information for a.out
...
(dbx) run
calcB called from calc, line 9
Running: a.out
(process id 1089)
calc called from MAIN, line 3
signal SEGV (no mapping at the fault address) in calcb at
line 23 in file "ShowTrace.f"
   23                   v(j) = (i * 10)
(dbx) where -V
=>[1] calcb(v = ARRAY , m = 2), line 23 in "ShowTrace.f"
  [2] calc(a = ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f"
  [3] MAIN(), line 3 in "ShowTrace.f"
(dbx)
```

Working With Arrays

dbx recognizes arrays and can print them.

```
demo% dbx a.out
Reading symbolic information…
(dbx) list 1,25
    1           DIMENSION IARR(4,4)
    2           DO 90 I = 1,4
    3                   DO 20 J = 1,4
    4                           IARR(I,J) = (I*10) + J
    5   20              CONTINUE
    6   90      CONTINUE
    7           END
(dbx)  stop at 7
(1) stop at "Arraysdbx.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "Arraysdbx.f"
    7           END
(dbx) print IARR
iarr =
       (1,1) 11
       (2,1) 21
       (3,1) 31
       (4,1) 41
       (1,2) 12
       (2,2) 22
       (3,2) 32
       (4,2) 42
       (1,3) 13
       (2,3) 23
       (3,3) 33
       (4,3) 43
       (1,4) 14
       (2,4) 24
       (3,4) 34
       (4,4) 44
(dbx) print IARR(2,3)
       iarr(2, 3) = 23  - Order of user-specified subscripts ok
(dbx) quit
```

For information on array slicing in Fortran, see “Array Slicing Syntax for Fortran” on page 129.

Fortran 95 Allocatable Arrays

The following example shows how to work with allocated arrays in dbx.

```
demo% f95 -g Alloc.f95
  demo% dbx a.out
  (dbx) list 1,99
      1   PROGRAM TestAllocate
      2   INTEGER n, status
      3   INTEGER, ALLOCATABLE :: buffer(:)
      4            PRINT *, 'Size?'
      5            READ *, n
      6            ALLOCATE( buffer(n), STAT=status )
      7           IF ( status /= 0 ) STOP 'cannot allocate buffer'
      8            buffer(n) = n
      9            PRINT *, buffer(n)
     10            DEALLOCATE( buffer, STAT=status)
     11   END
(dbx) stop at 6
 (2) stop at "alloc.f95":6
 (dbx) stop at 9
 (3) stop at "alloc.f95":9
 (dbx) run
 Running: a.out
 (process id 10749)
  Size?
 1000
Unknown size at line 6
 stopped in main at line 6 in file "alloc.f95"
     6           ALLOCATE( buffer(n), STAT=status )
 (dbx) whatis buffer
 integer*4 , allocatable::buffer(:)
 (dbx) next
 continuing
 stopped in main at line 7 in file "alloc.f95"
     7          IF ( status /= 0 ) STOP 'cannot allocate buffer'
 (dbx) whatis buffer
 integer*4 buffer(1:1000)
Known size at line 9
 (dbx) cont
 stopped in main at line 9 in file "alloc.f95"
     9           PRINT *, buffer(n)
 (dbx) print n
buffer(1000) holds 1000
 n = 1000
 (dbx) print buffer(n)
 buffer(n) = 1000
```

Showing Intrinsic Functions

dbx recognizes Fortran intrinsic functions (SPARC™ platforms only).

To show an intrinsic function in dbx, type:

```
demo% cat ShowIntrinsic.f
    INTEGER i
    i = -2
    END
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
Running: shi
(process id 18019)
stopped in MAIN at line 2 in file "shi.f"
    2               i = -2
(dbx) whatis abs
Generic intrinsic function: "abs"
(dbx) print i
i = 0
(dbx) step
stopped in MAIN at line 3 in file "shi.f"
    3               end
(dbx) print i
i = -2
(dbx) print abs(1)
abs(i) = 2
(dbx)
```

Showing Complex Expressions

dbx also recognizes Fortran complex expressions.

To show a complex expression in dbx, type:

```
demo% cat ShowComplex.f
   COMPLEX z
    z = ( 2.0, 3.0 )
    END
demo% f95 -g ShowComplex.f
demo% dbx a.out
(dbx) stop in MAIN
(dbx) run
Running: a.out
(process id 10953)
stopped in MAIN at line 2 in file "ShowComplex.f"
    2       z = ( 2.0, 3.0 )
(dbx) whatis z
complex*8  z
(dbx) print z
z = (0.0,0.0)
(dbx) next
stopped in MAIN at line 3 in file "ShowComplex.f"
    3       END
(dbx) print z
z = (2.0,3.0)
(dbx) print z+(1.0,1.0)
z+(1,1) = (3.0,4.0)
(dbx) quit
demo%
```

Showing Interval Expressions

To show an interval expression in dbx, type:

```
demo% cat ShowInterval.f95
   INTERVAL v
   v = [ 37.1, 38.6 ]
   END
demo% f95 -g -xia ShowInterval.f95
demo% dbx a.out
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
Running: a.out
(process id 5217)
stopped in MAIN at line 2 in file "ShowInterval.f95"
    2      v = [ 37.1, 38.6 ]
(dbx) whatis v
INTERVAL*16  v
(dbx) print v
v = [0.0,0.0]
(dbx) next
stopped in MAIN at line 3 in file "ShowInterval.f95"
    3      END
(dbx) print v
v = [37.1,38.6]
(dbx) print v+[0.99,1.01]
v+[0.99,1.01] = [38.09,39.61]
(dbx) quit
demo%
```

Showing Logical Operators

dbx can locate Fortran logical operators and print them.

To show logical operators in dbx, type:

```
demo% cat ShowLogical.f
        LOGICAL a, b, y, z
        a = .true.
        b = .false.
        y = .true.
        z = .false.
        END
demo% f95 -g ShowLogical.f
demo% dbx a.out
(dbx) list 1,9
    1           LOGICAL a, b, y, z
    2           a = .true.
    3           b = .false.
    4           y = .true.
    5           z = .false.
    6           END
(dbx) stop at 5
(2) stop at "ShowLogical.f":5
(dbx) run
Running: a.out
(process id 15394)
stopped in MAIN at line 5 in file "ShowLogical.f"
    5           z = .false.
(dbx) whatis y
logical*4 y
(dbx) print a .or. y
a.OR.y = true
(dbx) assign z = a .or. y
(dbx) print z
z = true
(dbx) quit
demo%
```

Viewing Fortran 95 Derived Types

You can show structures—Fortran 95 derived types—with `dbx`.

```
demo% f95 -g DebStruc.f95
demo% dbx a.out
(dbx) list 1,99
       1   PROGRAM Struct ! Debug a Structure
       2      TYPE product
       3         INTEGER        id
       4         CHARACTER*16   name
       5         CHARACTER*8    model
       6         REAL           cost
       7 REAL price
       8      END TYPE product
       9
   10      TYPE(product) :: prod1
   11
   12      prod1%id = 82
   13      prod1%name = "Coffee Cup"
   14      prod1%model = "XL"
   15      prod1%cost = 24.0
   16      prod1%price = 104.0
   17      WRITE ( *, * ) prod1%name
   18   END
(dbx) stop at 17
(2) stop at "Struct.f95":17
(dbx) run
Running: a.out
(process id 12326)
stopped in main at line 17 in file "Struct.f95"
   17      WRITE ( *, * ) prod1%name
(dbx) whatis prod1
product prod1
(dbx) whatis -t product
type product
       integer*4 id
       character*16 name
       character*8 model
       real*4 cost
       real*4 price
end type product
```

```
(dbx) n
(dbx) print prod1
       prod1 = (
       id    = 82
       name = 'Coffee Cup'
       model = 'XL'
       cost = 24.0
       price = 104.0
)
```

Pointer to Fortran 95 Derived Type

You can show structures—Fortran 95 derived types—and pointers with dbx.

```
demo% f95 -o debstr -g DebStruc.f95
 demo% dbx debstr
 (dbx) stop in main
 (2) stop in main
 (dbx) list 1,99
     1   PROGRAM DebStruPtr! Debug structures & pointers
Declare a derived type.
     2      TYPE product
     3         INTEGER        id
     4         CHARACTER*16   name
     5         CHARACTER*8    model
     6         REAL           cost
     7         REAL           price
     8      END TYPE product
     9
Declare prod1 and prod2 targets.
    10      TYPE(product), TARGET :: prod1, prod2
Declare curr and prior pointers.
    11      TYPE(product), POINTER :: curr, prior
    12
Make curr point to prod2.
    13      curr => prod2
Make prior point to prod1.
    14      prior => prod1
Initialize prior.
    15      prior%id = 82
    16      prior%name = "Coffee Cup"
    17      prior%model = "XL"
    18      prior%cost = 24.0
    19      prior%price = 104.0
```

```
Set curr to prior.
    20      curr = prior
Print name from curr and prior.
    21      WRITE ( *, * ) curr%name, " ", prior%name
    22   END PROGRAM DebStruPtr
 (dbx) stop at 21
 (1) stop at "DebStruc.f95":21
 (dbx) run
 Running: debstr
(process id 10972)
stopped in main at line 21 in file "DebStruc.f95"
   21      WRITE ( *, * ) curr%name, " ", prior%name
(dbx) print prod1
 prod1 = (
       id = 82
       name = "Coffee Cup"
       model = "XL"
       cost = 24.0
       price = 104.0
)
```

Above, dbx displays all fields of the derived type, including field names.

You can use structures—inquire about an item of an Fortran 95 derived type.

```
Ask about the variable
(dbx) whatis prod1
 product prod1
Ask about the type (-t)
 (dbx) whatis -t product
 type product
    integer*4 id
    character*16 name
    character*8 model
    real cost
    real price
 end type product
```

To print a pointer, type:

dbx displays the contents of a pointer, which is an address. This address can be different with every run.

```
(dbx) print prior
 prior = (
     id    = 82
     name = 'Coffee Cup'
     model = 'XL'
     cost = 24.0
     price = 104.0
 )
```

CHAPTER 17

Debugging a Java Application With dbx

This chapter describes how you can use dbx to debug an application that is a mixture of Java™ code and C JNI (Java™ Native Interface) code or C++ JNI code.

The chapter is organized into the following sections:

Using dbx With Java Code

You can use the Sun Studio dbx to debug mixed code (Java code and C code or C++ code) running under the Solaris™ operating environment.

Capabilities of dbx With Java Code

You can debug several types of Java applications with dbx (see "Starting to Debug a Java Application" on page 231). Most dbx commands operate similarly on native code and Java code.

Limitations of dbx With Java Code

dbx has the following limitations when debugging Java code:

- dbx cannot tell you the state of a Java application from a core file as it can with native code.
- dbx cannot tell you the state of a Java application if the application is hung for some reason and dbx is not able to make procedure calls.
- Fix and continue, runtime checking, and performance data collection do not apply to Java applications.

Environment Variables for Java Debugging

The following environment variables are specific to debugging a Java application with dbx. You can set the JAVASRCPATH, CLASSPATHX, and jvm_invocation environment variables at a shell prompt before starting dbx. The setting of the jdbx_mode environment variable changes as you are debugging your application. You can change its setting with the jon command ("jon Command" on page 340) and the joff command (see "joff Command" on page 340).

jdbx_mode	The jdbx_mode environment variable can have the following settings: java, jni, or native. For descriptions of the Java, JNI, and native modes, and how and when the mode changes, see "dbx Modes for Debugging Java Code" on page 240. Default: java.
JAVASRCPATH	You can use the JAVASRCPATH environment variable to specify the directories in which dbx should look for Java source files. This variable is useful when the Java sources files are not in the same directory as the .class or .jar files. See "Specifying the Location of Your Java Source Files" on page 234 for more information.
CLASSPATHX	The CLASSPATHX environment variable lets you specify to dbx a path for Java class files that are loaded by custom class loaders. For more information, see "Specifying a Path for Class Files That Use Custom Class Loaders" on page 235.
jvm_invocation	The jvm_invocation environment variable lets you customize the way the JVM™ software is started. (The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java™ platform.) For more information, see See "Customizing Startup of the JVM Software" on page 236.

Starting to Debug a Java Application

You can use dbx to debug the following types of Java applications:

- A file with a file name that ends in .class
- A file with a file name that ends in .jar
- A Java application that is started using a wrapper
- A running Java application that was started in debug mode to which you attach dbx
- A C application or C++ application that embeds a Java application using the JNI_CreateJavaVM interface

dbx recognizes that it is debugging a Java application in all of these cases.

Debugging a Class File

You can debug a file that uses the .class file name extension using dbx as in the following example.

```
(dbx) debug myclass.class
```

If the class that defines the application is defined in a package, you need to include the package path just as when running the application under the JVM software, as in the following example.

```
(dbx) debug java.pkg.Toy.class
```

You can also use a full path name for the class file. dbx automatically determines the package portion of the class path by looking in the .class file and adds the remaining portion of the full path name to the class path. For example, given the following path name, dbx determines that pkg/Toy.class is the main class name and adds /home/user/java to the class path.

```
(dbx) debug /home/user/java/pkg/Toy.class
```

Debugging a JAR File

A Java application can be bundled in a JAR (Java Archive) file. You can debug a JAR file using `dbx` as in the following example.

```
(dbx) debug myjar.jar
```

When you start debugging a file that has a file name ending in `.jar`, `dbx` uses the `Main_Class` attribute specified in the manifest of this JAR file to determine the main class. (The main class is the class within the JAR file that is your application's entry point. If you use a full path name or relative path name to specify the JAR file, `dbx` uses the directory name and prefixes it to the class path in the `Main-Class` attribute.

If you debug a JAR file that does not have the `Main-Class` attribute, you can use the JAR URL syntax `jar:<url>!/{entry}` that is specified in the class `JarURLConnection` of the Java™ 2 Platform, Standard Edition to specify the name of the main class, as in the following examples.

```
(dbx) debug jar:myjar.jar!/myclass.class
(dbx) debug jar:/a/b/c/d/e.jar!/x/y/z.class
(dbx) debug jar:file:/a/b/c/d.jar!/myclass.class
```

For each of these examples `dbx` would do the following:

- Treat the class path specified after the `!` character as the main class (for example, `/myclass.class` or `/x/y/z.class`)
- Add the name of the JAR file `./myjar.jar`, `/a/b/c/d/e.jar`, or `/a/b/c/d.jar` to the class pat
- Begin debugging the main class

Note – If you have specified a custom startup of the JVM software using the `jvm_invocation` environment variable (see "Customizing Startup of the JVM Software" on page 236, the file name of the JAR file is not automatically added to the class path In this case, you must add the file name of the JAR file to the class path when you start debugging.

Debugging a Java Application That Has a Wrapper

A Java application usually has a wrapper to set environment variables. If your Java application has a wrapper, you need to tell `dbx` that a wrapper script is being used by setting the `jvm_invocation` environment variable (see "Customizing Startup of the JVM Software" on page 236).

Attaching `dbx` to a Running Java Application

You can attach `dbx` to a running Java application if you specified the options shown in the following example when you started the application. After starting the application, you would use the `dbx` command (see "`dbx` Command" on page 317) with the process ID of the running process to start debugging.

```
$ java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent
myclass.class
$ dbx - 2345
```

For the JVM software to locate `libdbx_agent.so`, you need to add */installation_directory*`/SUNWspro/lib` to `LD_LIBRARY_PATH` before running the Java application, where *installation_directory* is the location where `dbx` was installed. If you are using the 64-bit version of the JVM software, you need to add */installation_directory*`/SUNWspro/lib/v9` to `LD_LIBRARY_PATH`.

When you attach `dbx` to the running application, `dbx` starts debugging the application in Java mode.

If your Java application requires 64-bit object libraries, include the `-d64` option when you start the application. Then when you attach `dbx` to the application, `dbx` will use the 64-bit JVM software on which the application is running.

```
$ java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent -d64
myclass.class
$ dbx - 2345
```

Debugging a C Application or C++ Application That Embeds a Java Application

You can debug a C application or C++ application that embeds a Java application using the `JNI_CreateJavaVM` interface. The C application or C++ application must start the Java application by specifying the following options to the JVM software:

```
-Xdebug -Xnoagent -Xrundbx_agent
```

For the JVM software to locate `libdbx_agent.so`, you need to add *install_directory*`/current/lib` to `LD_LIBRARY_PATH` before running the Java application, where *install_directory* is the location where `dbx` was installed. If you are using the 64-bit version of the JVM software, you need to add *install_directory*`/current/lib/v9` to `LD_LIBRARY_PATH`.

Passing Arguments to the JVM Software

When you use the `run` command in Java mode, the arguments you give are passed to the application and not to the JVM software. To pass arguments to the JVM software, see "Customizing Startup of the JVM Software" on page 236.

Specifying the Location of Your Java Source Files

Sometimes your Java source files are not in the same directory as the `.class` or `.jar` files. You can use the `$JAVASRCPATH` environment variable to specify the directories in which `dbx` should look for Java source files. For example `JAVASRCPATH=.:/mydir/mysrc:/mydir/mylibsrc:/mydir/myutils` causes `dbx` to look in the listed directories for source files that correspond to the class files being debugged.

Specifying the Location of Your C Source Files or C++ Source Files

`dbx` might not be able to find your C source files or C++ source files in the following circumstances:

- If your source files are not in the same location as they were when you compiled them

- If you compiled your source files on a different system than the one on which you are running dbx and the compile directory does not have the same path name

In such cases, use the pathmap command (see "pathmap Command" on page 356) to map one path name to another so that dbx can find your files.

Specifying a Path for Class Files That Use Custom Class Loaders

An application can have custom class loaders that load class files from locations that might not be part of the regular class path. In such situations dbx cannot locate the class files. The CLASSPATHX environment variable lets you specify to dbx a path for the class files that are loaded by their custom class loaders. For example, CLASSPATHX=.:/myloader/myclass:/mydir/mycustom causes dbx to look in the listed directories when it is trying to locate a class file.

Setting Breakpoints on Code That Has Not Yet Been Loaded by the JVM Software

To set a stop breakpoint on a Java method in a class file that has not been loaded by the JVM software, use the full name of the class with a stop in command, or the class name with a stop inmethod command. See the following example.

```
(dbx) stop in Java.Pkg.Toy.myclass.class.mymethod
(dbx) stop inmethod myclass.class.mymethod
```

To set a stop breakpoint on a C function or C++ function in a shared library that has not been loaded by the JVM software, preload the symbol table of the shared library before setting the breakpoint. For example, if you have a library named mylibrary.so that contains a function named myfunc, you could preload the library and set a breakpoint on the function as follows:

```
(dbx) loadobject -load fullpathto/mylibrary.so
(dbx> stop in myfunc
```

You can also load the symbol tables of all dynamically loaded shared objects by running your application once before beginning to debug it with dbx.

Customizing Startup of the JVM Software

You might need to customize startup of the JVM software from `dbx` to do the following:

- Specify a path name for the JVM software (see "Specifying a Path Name for the JVM Software" on page 237)
- Pass some run arguments to the JVM software (see "Passing Run Arguments to the JVM Software" on page 237)
- Specify a custom wrapper instead of the default Java wrapper for running Java applications (see "Specifying a Custom Wrapper for Your Java Application" on page 237)
- Specify 64-bit JVM software (see "Specifying 64-bit JVM Software" on page 240)

You can customize startup of the JVM software using the `jvm_invocation` environment variable. By default, when the `jvm_invocation` environment variable is not defined, `dbx` starts the JVM software as follows:

```
java -Xdebug -Xnoagent -Xrundbx_agent:syncpid
```

When the `jvm_invocation` environment variable is defined, `dbx` uses the value of the variable to start the JVM software.

You must include the `-Xdebug` option in the definition of the `jvm_invocation` environment variable. `dbx` expands `-Xdebug` into the internal options `-Xdebug -Xnoagent -Xrundbxagent::sync`.

If you do not include the `-Xdebug` option in the definition, as in the following example, `dbx` issues an error message.

```
jvm_invocation="/set/java/javasoft/sparc-S2/jdk1.2/bin/java"
```

```
dbx: Value of '$jvm_invocation' must include an option to invoke
the VM in debug mode
```

Specifying a Path Name for the JVM Software

To specify a path name for the JVM software, set the `jvm_invocation` environment variable to the appropriate path name, as in the following example.

```
jvm_invocation="/myjava/java -Xdebug"
```

This causes `dbx` to start the JVM software as follows:

```
/myjava/java -Djava.compiler=NONE -Xdebug -Xnoagent -
Xrundbx_agent:sync
```

Passing Run Arguments to the JVM Software

To pass run arguments to the JVM software, set the `jvm_invocation` environment variable to start the JVM software with those arguments, as in the following example.

```
jvm_invocation="java -Xdebug -Xms512 -Xmx1024 -Xcheck:jni"
```

This causes `dbx` to start the JVM software as follows:

```
java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent:sync=
-Xms512 -Xmx1024 -Xcheck:jni
```

Specifying a Custom Wrapper for Your Java Application

A Java application can use a custom wrapper for startup. If your application uses a custom wrapper, you can use the `jvm_invocation` environment variable to specify the wrapper to be used, as in the following example.

```
jvm_invocation="/export/siva-a/forte4j/bin/forte4j.sh -J-Xdebug"
```

This causes dbx to start the JVM software as follows:

```
/export/siva-a/forte4j/bin/forte4j.sh - -J-Xdebug -J-Xnoagent -J-
Xrundbxagent:sync=process_id
```

Using a Custom Wrapper That Accepts Command-Line Options

The following wrapper script (xyz) sets a few environment variables and accepts command line options:

```
#!/bin/sh
CPATH=/mydir/myclass:/mydir/myjar.jar; export CPATH
JARGS="-verbose:gc -verbose:jni -DXYZ=/mydir/xyz"
ARGS=
while [ $# -gt 0 ] ; do
    case "$1" in
        -userdir) shift; if [ $# -gt 0 ]
; then userdir=$1; fi;;
        -J*) jopt=`expr $1 : '-J<.*>'`
; JARGS="$JARGS '$jopt'";;
        *) ARGS="$ARGS '$1'" ;;
    esac
    shift
done
java $JARGS -cp $CPATH $ARGS
```

This script accepts some command line options for the JVM software and the user application. For wrapper scripts of this form, you would set the jvm_invocation environment variable and start dbx as follows:

```
% jvm_invocation="xyz -J-Xdebug -Jany other java options"
% dbx myclass.class -Dide=visual
```

Using a Custom Wrapper That Does Not Accept Command-Line Options

The following wrapper script (`xyz`) sets a few environment variables and starts the JVM software, but does not accept any command line options or a class name:

```
#!/bin/sh
CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
ABC=/mydir/abc; export ABC
java <options> myclass
```

You could use such a script to debug a wrapper using `dbx` in one of two ways:

- You could modify the script to start `dbx` from inside the wrapper script itself by adding the definition of the `jvm_invocation` variable to the script and starting `dbx`:

```
#!/bin/sh
CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
ABC=/mydir/abc; export ABC
jvm_invocation="java -Xdebug <options>"; export jvm_invocation
dbx myclass.class
```

  Once you have made this modification, you could start the debugging session by running the script.

- You could modify the script slightly to accept some command line options as follows:

```
#!/bin/sh
CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
ABC=/mydir/abc; export ABC
JAVA_OPTIONS="$1 <options>"
java $JAVA_OPTIONS $2
```

  Once you made this modification, you would set the `jvm_invocation` environment variable and start `dbx` as follows:

```
% jvm_invocation="xyz -Xdebug"; export jvm_invocation
% dbx myclass.class
```

Specifying 64-bit JVM Software

If you want dbx to start 64-bit JVM software to debug an application that requires 64-bit object libraries, include the -d64 option when you set the jvm_invocation environment variable:

```
jvm_invocation="/myjava/java -Xdebug -d64"
```

dbx Modes for Debugging Java Code

When debugging a Java application, dbx is in one of three modes:

- Java mode
- JNI mode
- Native mode

When dbx is Java mode or JNI (Java Native Interface) mode, you can inspect the state of your Java application, including JNI code, and control execution of the code. When dbx is in native mode, you can inspect the state of your C or C++ JNI code. The current mode (java, jni, native) is stored in the environment variable jdbx_mode.

In Java mode, you interact with dbx using Java syntax and dbx uses Java syntax to present information to you. This mode is used for debugging pure Java code, or the Java code in an application that is a mixture of Java code and C JNI code or C++ JNI code.

In JNI mode, dbx commands use native syntax and affect native code, but the output of commands shows Java-related status as well as native status, so JNI mode is a "mixed" mode. This mode is used for debugging the native parts of an application that is a mixture of Java code and C JNI code or C++ JNI code.

In native mode, dbx commands affect only a native program, and all Java-related features are disabled. This mode is used for debugging non-Java related programs.

As you execute your Java application, dbx switches automatically between Java mode and JNI mode as appropriate. For example, when it encounters a Java breakpoint, dbx switches into Java mode, and when you step from Java code into JNI code, it switches into JNI mode.

Switching from Java or JNI Mode to Native Mode

dbx does not switch automatically into native mode. You can switch explicitly from Java or JNI Mode to native mode with the joff command, and from native mode to Java mode with the jon command.

Switching Modes When You Interrupt Execution

If you interrupt execution of your Java application (for example, with a control-C), dbx tries to set the mode automatically to Java/JNI mode by bringing the application to a safe state and suspending all threads.

If dbx cannot suspend the application and switch to Java/JNI mode, dbx switches to native mode. You can then use the jon command to switch to Java mode so that you inspect the state of the program.

Using dbx Commands in Java Mode

When you are using dbx to debug a mixture of Java and native code, dbx commands fall into several categories:

- Commands that accept the same arguments and operate the same way in Java mode or JNI mode as in native mode (see "Commands With Identical Syntax and Functionality in Java Mode and Native Mode" on page 243).
- Commands have arguments that are valid only in Java mode or JNI mode, as well as arguments that are valid only in native mode (see "Commands With Different Syntax in Java Mode" on page 244).
- Commands that are valid only in Java mode or JNI mode (see "Commands Valid Only in Java Mode" on page 245).

Any commands not included in one of these categories work only in native mode.

The Java Expression Evaluation in dbx Commands

The Java expression evaluator used in most dbx commands supports the following constructs:

- All literals

- All names and field accesses
- `this` and `super`
- Array accesses
- Casts
- Conditional binary operations
- Method calls
- Other unary/binary operations
- Assignment to variables or fields
- `instanceof` operator
- Array length operator

The Java expression evaluator does not support the following constructs:

- Qualified `this`, for example, <*ClassName*>`.this`
- Class instance creation expressions
- Array creation expressions
- String concatenation operator
- Conditional operator ? :
- Compound assignment operators, for example, `x += 3`

A particularly useful way of inspecting the state of your Java application is using the display facility in the dbx Debugger.

Depending on precise value semantics in expressions that do more than just inspect data is not recommended.

Static and Dynamic Information Used by `dbx` Commands

Much of the information about a Java application is normally available only after the JVM software has started, and is unavailable after the Java application has finished executing. However, when you debug a Java application with `dbx`, `dbx` gleans some of the information it needs from class files and JAR files that are part of the system class path and user class path before it starts the JVM software. This allows `dbx` to do better error checking on breakpoints before you run the application.

Some Java classes and their attributes might not be accessible through the class path. `dbx` can inspect and step through these classes, and the expression parser can access them, once they are loaded. However, the information it gathers is temporary and is no longer available after the JVM software terminates.

Some information that `dbx` needs to debug your Java application is not recorded anywhere so `dbx` skims Java source files to derive this information as it is debugging your code.

Commands With Identical Syntax and Functionality in Java Mode and Native Mode

The following dbx commands have the same syntax and perform the same operations in Java mode as in native mode.

Command	Functionality
attach	Attaches dbx to a running process, stopping execution and putting the program under debugging control
cont	Causes the process to continue execution
dbxenv	List or set dbx environment variables
delete	Deletes breakpoints and other events
down	Moves down the call stack (away from main)
dump	Prints all variables local to a procedure or method
file	Lists or changes the current file
frame	Lists or changes the current stack frame number
handler	Modifies event handlers (breakpoints)
import	Import commands from a dbx command library
line	Lists or changes the current line number
list	Lists or changes the current line number
next	Steps one source line (steps over calls)
pathmap	Maps one path name to another for finding source files, etc.
proc	Displays the status of the current process
prog	Manages programs being debugged and their attributes
quit	Exits dbx
rerun	Runs the program with no arguments
runargs	Changes the arguments of the target process
status	Lists the event handlers (breakpoints)
step up	Steps up and out of the current function or method
stepi	Steps one machine instruction (steps into calls)
up	Moves up the call stack (toward main)
whereami	Displays the current source line

Commands With Different Syntax in Java Mode

The following `dbx` commands have different syntax for Java debugging than for native code debugging, and operate differently in Java mode than in native mode.

Command	Native Mode Functionality	Java Mode Functionality
`assign`	Assigns a new value to a program variable	Assigns a new value to a local variable or parameter
`call`	Calls a procedure	Calls a method
`dbx`	Starts `dbx`	Starts `dbx`
`debug`	Loads the specified application and begins debugging the application	Loads the specified Java application, checks for the existence of the class file, and begins debugging the application
`detach`	Releases the target process from `dbx`'s control	Releases the target process from `dbx`'s control
`display`	Evaluates and prints expressions at every stopping point.	Evaluates and prints expressions, local variables, or parameters at every stopping point
`files`	Lists file names that match a regular expression	Lists all of the Java source files known to `dbx`
`func`	Lists or changes the current function	Lists or changes the current method
`next`	Steps one source line (stepping over calls)	Steps one source line (stepping over calls)
`print`	Prints the value of an expression	Prints the value of an expression, local variable, or parameter.
`run`	Runs the program with arguments	Runs the program with arguments
`step`	Steps one source line or statement (stepping into calls)	Steps one source line or statement (stepping into calls)
`stop`	Sets a source-level breakpoint	Sets a source-level breakpoint
`thread`	Lists or changes the current thread	Lists or changes the current thread
`threads`	Lists all threads	Lists all threads
`trace`	Shows executed source lines, function calls, or variable changes	Shows executed source lines, function calls, or variable changes
`undisplay`	Undoes `display` commands	Undoes `display` commands

Command	Native Mode Functionality	Java Mode Functionality
`whatis`	Prints the type of expression or declaration of type	Prints the declaration of an identifier
`when`	Executes commands when a specified event occurs	Executes commands when a specified event occurs
`where`	Prints the call stack	Prints the call stack

Commands Valid Only in Java Mode

The following `dbx` commands are valid only in Java mode or JNI mode.

Command	Functionality
`java`	Used when `dbx` is in JNI mode to indicate that the Java version of a specified command is to be executed
`javaclasses`	Prints the names of all Java classes known to `dbx` when you give the command
`joff`	Switches `dbx` from Java mode or JNI mode to native mode
`jon`	Switches `dbx` from native mode to Java mode
`jpkgs`	Prints the names of all Java packages known to `dbx` when you give the command
native	Used when `dbx` is in Java mode to indicate that the native version of a specified command is to be executed

CHAPTER 18

Debugging at the Machine-Instruction Level

This chapter describes how to use event management and process control commands at the machine-instruction level, how to display the contents of memory at specified addresses, and how to display source lines along with their corresponding machine instructions. The `next`, `step`, `stop` and `trace` commands each support a machine-instruction level variant: `nexti`, `stepi`, `stopi`, and `tracei`. Use the `regs` command to print out the contents of machine registers or the `print` command to print out individual registers.

This chapter is organized into the following sections:

Examining the Contents of Memory

Using addresses and the `examine` or `x` command, you can examine the content of memory locations as well as print the assembly language instruction at each address. Using a command derived from `adb`(1), the assembly language debugger, you can query for:

- The *address*, using the `=` (equal sign) character, or,
- The *contents* stored at an address, using the `/` (slash) character.

You can print the assembly commands using the `dis` and `listi` commands. (See "Using the `dis` Command" on page 251 and "Using the `listi` Command" on page 251.)

Using the `examine` or `x` Command

Use the `examine` command, or its alias `x`, to display memory contents or addresses.

Use the following syntax to display the contents of memory starting at *address* for *count* items in format *format*. The default *address* is the next one after the last address previously displayed. The default *count* is 1. The default *format* is the same as was used in the previous `examine` command, or `X` if this is the first command given.

The syntax for the `examine` command is:

```
examine [address] [/ [count] [format]]
```

To display the contents of memory from *address1* through *address2* inclusive, in format *format*, type:

```
examine address1, address2 [/ [format]]
```

Display the address, instead of the contents of the address in the given format by typing:

```
examine address = [format]
```

To print the value stored at the next address after the one last displayed by `examine`, type:

```
examine +/ i
```

To print the value of an expression, enter the expression as an address:

```
examine address=format
examine address=
```

Addresses

The *address* is any expression resulting in or usable as an address. The *address* may be replaced with a + (plus sign), which displays the contents of the next address in the default format.

For example, the following are valid addresses.:

`0xff99`	An absolute address
`main`	Address of a function
`main+20`	Offset from a function address
`&errno`	Address of a variable
`str`	A pointer-value variable pointing to a string

Symbolic addresses used to display memory are specified by preceding a name with an ampersand (&). Function names can be used without the ampersand; `&main` is equal to `main`. Registers are denoted by preceding a name with a dollar sign ($).

Formats

The *format* is the address display format in which `dbx` displays the results of a query. The output produced depends on the current display *format*. To change the display format, supply a different *format* code.

The default format set at the start of each `dbx` session is `X`, which displays an address or value as a 32-bit word in hexadecimal. The following memory display formats are legal.

i	Display as an assembly instruction.
d	Display as 16 bits (2 bytes) in decimal.
D	Display as 32 bits (4 bytes) in decimal.
o	Display as 16 bits (2 bytes) in octal.
O	Display as 32 bits (4 bytes) in octal.
x	Display as 16 bits (2 bytes) in hexadecimal.
X	Display as 32 bits (4 bytes) in hexadecimal. (default format)
b	Display as a byte in octal.
c	Display as a character.
w	Display as a wide character.
s	Display as a string of characters terminated by a null byte.
W	Display as a wide character.
f	Display as a single-precision floating point number.
F, g	Display as a double-precision floating point number.

E	Display as an extended-precision floating point number.
ld, lD	Display 32 bits (4 bytes) in decimal (same as D).
lo, lO	Display 32 bits (4 bytes) in octal (same as O).
lx, LX	Display 32 bits (4 bytes) in hexadecimal (same as X).
Ld, LD	Display 64 bits (8 bytes) in decimal.
Lo, LO	Display 64 bits (8 bytes) in octal.
Lx, LX	Display 64 bits (8 bytes) in hexadecimal.

Count

The *count* is a repetition count in decimal. The increment size depends on the memory display format.

Examples of Using an Address

The following examples show how to use an address with *count* and *format* options to display five successive disassembled instructions starting from the current stopping point.

For SPARC:

```
(dbx) stepi
stopped in main at 0x108bc
0x000108bc: main+0x000c: st    %l0, [%fp - 0x14]
(dbx) x 0x108bc/5i
0x000108bc: main+0x000c: st    %l0, [%fp - 0x14]
0x000108c0: main+0x0010: mov   0x1,%l0
0x000108c4: main+0x0014: or    %l0,%g0, %o0
0x000108c8: main+0x0018: call  0x00020b90 [unresolved PLT 8:
malloc]
0x000108cc: main+0x001c: nop
```

For Intel:

```
(dbx) x &main/5i
0x08048988: main       :  pushl  %ebp
0x08048989: main+0x0001:  movl   %esp,%ebp
0x0804898b: main+0x0003:  subl   $0x28,%esp
0x0804898e: main+0x0006:  movl   0x8048ac0,%eax
0x08048993: main+0x000b:  movl   %eax,-8(%ebp)
```

Using the `dis` Command

The `dis` command is equivalent to the `examine` command with `i` as the default display format.

Here is the syntax for the `dis` command.

```
dis [address] [address1, address2] [/count]
```

The `dis` command:

- Without arguments displays 10 instructions starting at +.
- With the *address* argument only, disassembles 10 instructions starting at *address*.
- With the *address1* and *address2* arguments, disassembles instructions from *address1* through *address2*.
- With only a *count*, displays count instructions starting at +.

Using the `listi` Command

To display source lines with their corresponding assembly instructions, use the `listi` command, which is equivalent to the command `list -i`. See the discussion of `list -i` in "Printing a Source Listing" on page 75.

For SPARC:

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x0001083c: main+0x0014:  ld      [%fp + 0x48], %l0
0x00010840: main+0x0018:  add     %l0, 0x4, %l0
0x00010844: main+0x001c:  ld      [%l0], %l0
0x00010848: main+0x0020:  or      %l0, %g0, %o0
0x0001084c: main+0x0024:  call    0x000209e8 [unresolved PLT 7:
atoi]
0x00010850: main+0x0028:  nop
0x00010854: main+0x002c:  or      %o0, %g0, %l0
0x00010858: main+0x0030:  st      %l0, [%fp - 0x8]
   14       j = foo(i);
0x0001085c: main+0x0034:  ld      [%fp - 0x8], %l0
0x00010860: main+0x0038:  or      %l0, %g0, %o0
0x00010864: main+0x003c:  call    foo
0x00010868: main+0x0040:  nop
0x0001086c: main+0x0044:  or      %o0, %g0, %l0
0x00010870: main+0x0048:  st      %l0, [%fp - 0xc]
```

For Intel:

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x080488fd: main+0x000d:  movl   12(%ebp),%eax
0x08048900: main+0x0010:  movl   4(%eax),%eax
0x08048903: main+0x0013:  pushl  %eax
0x08048904: main+0x0014:  call   atoi <0x8048798>
0x08048909: main+0x0019:  addl   $4,%esp
0x0804890c: main+0x001c:  movl   %eax,-8(%ebp)
   14       j = foo(i);
0x0804890f: main+0x001f:  movl   -8(%ebp),%eax
0x08048912: main+0x0022:  pushl  %eax
0x08048913: main+0x0023:  call   foo <0x80488c0>
0x08048918: main+0x0028:  addl   $4,%esp
0x0804891b: main+0x002b:  movl   %eax,-12(%ebp)
```

Stepping and Tracing at Machine-Instruction Level

Machine-instruction level commands behave the same as their source level counterparts except that they operate at the level of single instructions instead of source lines.

Single Stepping at the Machine-Instruction Level

To single step from one machine instruction to the next machine instruction, use the `nexti` command or the `stepi` command

The `nexti` command and the `stepi` command behave the same as their source-code level counterparts: the `nexti` command steps *over* functions, the `stepi` command steps into a function called by the next instruction (stopping at the first instruction in the called function). The command forms are also the same. See "`next` Command" on page 353 and "`step` Command" on page 372 for a description.

The output from the `nexti` command and the `stepi` command differs from the corresponding source level commands in two ways:

- The output includes the *address* of the instruction at which the program is stopped (instead of the source code line number).

- The default output contains the *disassembled instruction* instead of the source code line.

For example:

```
(dbx) func
hand::ungrasp
(dbx) nexti
ungrasp +0x18:  call support
(dbx)
```

For more information, see "nexti Command" on page 355 and "stepi Command" on page 374.

Tracing at the Machine-Instruction Level

Tracing techniques at the machine-instruction level work the same as at the source code level, except you use the `tracei` command For the `tracei` command, `dbx` executes a single instruction only after each check of the address being executed or the value of the variable being traced. The `tracei` command produces automatic `stepi`-like behavior: the program advances one instruction at a time, stepping into function calls.

When you use the `tracei` command, it causes the program to stop for a moment after each instruction while `dbx` checks for the address execution or the value of the variable or expression being traced. Using the `tracei` command can slow execution considerably.

For more information on trace and its event specifications and modifiers, see "Tracing Execution" on page 112 and "tracei Command" on page 390.

Here is the general syntax for `tracei`:

```
tracei event-specification [modifier]
```

Commonly used forms of `tracei` are:

`tracei step`	Trace each instruction.
`tracei next`	Trace each instruction, but skip over calls.
`tracei at` *address*	Trace the given code address.

For more information, see "tracei Command" on page 390.

For SPARC:

```
(dbx) tracei next -in main
(dbx) cont
0x00010814: main+0x0004:  clr     %l0
0x00010818: main+0x0008:  st      %l0, [%fp - 0x8]
0x0001081c: main+0x000c:  call    foo
0x00010820: main+0x0010:  nop
0x00010824: main+0x0014:  clr     %l0
....
....
(dbx) (dbx) tracei step -in foo -if glob == 0
(dbx) cont
0x000107dc: foo+0x0004:  mov     0x2, %l1
0x000107e0: foo+0x0008:  sethi   %hi(0x20800), %l0
0x000107e4: foo+0x000c:  or      %l0, 0x1f4, %l0     ! glob
0x000107e8: foo+0x0010:  st      %l1, [%l0]
0x000107ec: foo+0x0014:  ba      foo+0x1c
....
....
```

Setting Breakpoints at the Machine-Instruction Level

To set a breakpoint at the machine-instruction level, use the `stopi` command. The command accepts any *event specification*, using the syntax:

```
stopi event-specification [modifier]
```

Commonly used forms of the `stopi` command are:

```
stopi [at address] [-if cond]
stopi in function [-if cond]
```

For more information, see "`stopi` Command" on page 380.

Setting a Breakpoint at an Address

To set a breakpoint at a specific address, type:

```
(dbx) stopi at address
```

For example:

```
(dbx) nexti
stopped in hand::ungrasp at 0x12638
(dbx) stopi at &hand::ungrasp
(3) stopi at &hand::ungrasp
(dbx)
```

Using the adb Command

The adb command lets you enter commands in an adb(1) syntax. You can also enter adb mode which interprets every command as adb syntax. Most adb commands are supported.

For more information, see "adb Command" on page 299.

Using the regs Command

The regs command lets you print the value of all the registers.

Here is the syntax for the regs command:

```
regs [-f][-F]
```

-f includes floating point registers (single precision). -F includes floating point registers (double precision). These are SPARC-only options.

For more information, see "regs Command" on page 363.

For SPARC:

```
dbx[13] regs -F
current thread: t@1
current frame:  [1]
g0-g3    0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7    0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3    0x00000003 0x00000014 0xef7562b4 0xeffff420
o4-o7    0xef752f80 0x00000003 0xeffff3d8 0x000109b8
l0-l3    0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7    0xeffff438 0x00000001 0x00000007 0xef74df54
i0-i3    0x00000001 0xeffff4a4 0xeffff4ac 0x00020c00
i4-i7    0x00000001 0x00000000 0xeffff440 0x000108c4
y        0x00000000
psr      0x40400086
pc       0x000109c0:main+0x4    mov     0x5, %l0
npc      0x000109c4:main+0x8    st      %l0, [%fp - 0x8]
f0f1     +0.00000000000000e+00
f2f3     +0.00000000000000e+00
f4f5     +0.00000000000000e+00
f6f7     +0.00000000000000e+00
...
```

Platform-Specific Registers

The following tables list platform-specific register names for SPARC, Intel, and AMD64 architectures that can be used in expressions.

SPARC Register Information

The following register information is for SPARC architecture.

Register	Description
`$g0 through $g7`	Global registers
`$o0 through $o7`	“out” registers
`$l0 through $l7`	“local” registers
`$i0 through $i7`	“in” registers
`$fp`	Frame pointer, equivalent to register $i6
`$sp`	Stack pointer, equivalent to register $o6
`$y`	Y register

Register	Description
`$psr`	Processor state register
`$wim`	Window invalid mask register
`$tbr`	Trap base register
`$pc`	Program counter
`$npc`	Next program counter
`$f0 through $f31`	FPU “f” registers
`$fsr`	FPU status register
`$fq`	FPU queue

The `$f0f1` `$f2f3` ... `$f30f31` pairs of floating-point registers are treated as having C “double” type (normally `$fN` registers are treated as C “float” type). These pairs can also be referred to as `$d0` ... `$d30`.

The following additional registers are available on SPARC V9 and V8+ hardware:

```
$g0g1 through $g6g7
$o0o1 through $o6o7
$xfsr $tstate $gsr
$f32f33 $f34f35 through $f62f63 ($d32 ... $$d62)
```

See the *SPARC Architecture Reference Manual* and the *SPARC Assembly Language Reference Manual* for more information on SPARC registers and addressing.

Intel Register Information

The following register information is for Intel architecture.

Register	Description
`$gs`	Alternate data segment register
`$fs`	Alternate data segment register
`$es`	Alternate data segment register
`$ds`	Data segment register
`$edi`	Destination index register
`$esi`	Source index register
`$ebp`	Frame pointer

Register	Description
`$esp`	Stack pointer
`$ebx`	General register
`$edx`	General register
`$ecx`	General register
`$eax`	General register
`$trapno`	Exception vector number
`$err`	Error code for exception
`$eip`	Instruction pointer
`$cs`	Code segment register
`$eflags`	Flags
`$uesp`	User stack pointer
`$ss`	Stack segment register

Commonly used registers are also aliased to their machine independent names.

Register	Description
`$SP`	Stack pointer; equivalent of `$uesp`
`$pc`	Program counter; equivalent of `$eip`
`$fp`	Frame pointer; equivalent of `$ebp`

Registers for the 80386 lower halves (16 bits) are:

Register	Description
`$ax`	General register
`$cx`	General register
`$dx`	General register
`$bx`	General register
`$si`	Source index register
`$di`	Destination index register
`$ip`	Instruction pointer, lower 16 bits
`$flags`	Flags, lower 16 bits

The first four 80386 16-bit registers can be split into 8-bit parts:

Register	Description
`$al`	Lower (right) half of register `$ax`
`$ah`	Higher (left) half of register `$ax`
`$cl`	Lower (right) half of register `$cx`
`$ch`	Higher (left) half of register `$cx`
`$dl`	Lower (right) half of register `$dx`
`$dh`	Higher (left) half of register `$dx`
`$bl`	Lower (right) half of register `$bx`
`$bh`	Higher (left) half of register `$bx`

Registers for the 80387 are:

Register	Description
`$fctrl`	Control register
`$fstat`	Status register
`$ftag`	Tag register
`$fip`	Instruction pointer offset
`$fcs`	Code segment selector
`$fopoff`	Operand pointer offset
`$fopsel`	Operand pointer selector
`$st0 through $st7`	Data registers

AMD64 Register Information

The following register information is for AMD64 architecture:

Register	Description
`RAX`	General purpose register - argument passing for function calls
`RBX`	General purpose register - callee-saved
`RCX`	General purpose register - argument passing for function calls
`RDX`	General purpose register - argument passing for function calls

Register	Description
RBP	General purpose register - stack management
RSI	General purpose register - argument passing for function calls
RDI	General purpose register - argument passing for function calls
RSP	General purpose register - stack management
R8	General purpose register - argument passing for function calls
R9	General purpose register - argument passing for function calls
R10	General purpose register - temporary
R11	General purpose register - temporary
R12	General purpose register - callee-saved
R13	General purpose register - callee-saved
R14	General purpose register - callee-saved
R15	General purpose register - callee-saved
RFLAGS	Flags register
RIP	Instruction pointer
MMX0	64-bit media and floating point register
MMX1	64-bit media and floating point register
MMX2	64-bit media and floating point register
MMX3	64-bit media and floating point register
MMX4	64-bit media and floating point register
MMX5	64-bit media and floating point register
MMX6	64-bit media and floating point register
MMX7	64-bit media and floating point register
XMM0	128-bit media register
XMM1	128-bit media register
XMM2	128-bit media register
XMM3	128-bit media register
XMM4	128-bit media register
XMM5	128-bit media register
XMM6	128-bit media register
XMM7	128-bit media register
XMM8	128-bit media register

Register	Description
XMM9	128-bit media register
XMM10	128-bit media register
XMM11	128-bit media register
XMM12	128-bit media register
XMM13	128-bit media register
XMM14	128-bit media register
XMM15	128-bit media register
CS	Segment register
DS	Segment register
ES	Segment register
FS	Segment register
GS	Segment register
SS	Segment register

CHAPTER 19

Using dbx With the Korn Shell

The dbx command language is based on the syntax of the Korn Shell (ksh 88), including I/O redirection, loops, built-in arithmetic, history, and command-line editing. This chapter lists the differences between ksh-88 and dbx command language.

If no dbx initialization file is located on startup, dbx assumes ksh mode.

This chapter is organized into the following sections:

ksh-88 Features Not Implemented

The following features of ksh-88 are not implemented in dbx:

- `set -A` *name* for assigning values to array *name*
- `set -o` particular options: `allexport bgnice gmacs markdirs noclobber nolog privileged protected viraw`
- `typeset -l -u -L -R -H` attributes
- backquote (`` `...` ``) for command substitution (use `$(...)` instead)
- [[*expression*]] compound command for expression evaluation
- `@(pattern[|pattern] ...)` extended pattern matching
- co-processes (command or pipeline running in the background that communicates with your program)

Extensions to ksh-88

`dbx` adds the following features as extensions:

- `$[ p -> flags ]` language expression
- `typeset -q` enables special quoting for user-defined functions
- csh-like `history` and `alias` arguments
- `set +o path` disables path searching
- `0xabcd` C syntax for octal and hexadecimal numbers
- `bind` to change Emacs-mode bindings
- `set -o hashall`
- `set -o ignore suspend`
- `print -e` and `read -e` (opposite of `-r`, raw)
- built-in `dbx` commands

Renamed Commands

Particular `dbx` commands have been renamed to avoid conflicts with ksh commands.

- The `dbx print` command retains the name `print`; the ksh `print` command has been renamed `kprint`.
- The ksh `kill` command has been merged with the `dbx kill` command.
- The `alias` command is the ksh `alias`, unless in `dbx` compatibility mode.
- *address*`/`*format* is now `examine` *address*`/`*format*.
- `/`*pattern* is now `search` *pattern*.
- `?`*pattern* is now `bsearch` *pattern*.

Rebinding of Editing Functions

The `bind` command allows rebinding of editing functions. You can use the command to display or modify the key bindings for EMacs-style editors and vi-style editors. The syntax of the `bind` command is:

`bind`	Display the current editing key bindings
`bind` *key=definition*	Bind *key* to *definition*
`bind` *key*	Display the current definition for *key*
`bind` *key=*	Remove binding of *key*
`bind -m` *key=definition*	Define *key* to be a macro with *definition*
`bind -m`	Same as `bind`

where:

key is the name of a key.

definition is the definition of the macro to be bound to the key.

The following are some of the more important default key bindings for EMacs-style editors:

^A = beginning-of-line	^B = backward-char
^D = eot-or-delete	^E = end-of-line
^F = forward-char	^G = abort
^K = kill-to-eo	^L = redraw
^N = down-history	^P = up-history
^R = search-history	^^ = quote
^? = delete-char-backward	^H = delete-char-backward
^[b = backward-word	^[d = delete-word-forward
^[f = forward-word	^[^H = delete-word-backward
^[^[= complete	^[? = list-command

The following are some of the more important default key bindings for vi-style editors:

a = append	A = append at EOL
c = change	d = delete
G = go to line	h = backward character
i = insert	I = insert at BOL
j = next line	k = previous line
l = forward line	n = next match
N = prev match	p = put after
P = put before	r = repeat
R = replace	s = substitute
u = undo	x = delete character
X = delete previous character	y = yank
~ = transpose case	_ = last argument
* = expand	= = list expansion
- = previous line	+ = next line
sp = forward char	# = comment out command
? = search history from beginning	
/ = search history from current	

In insert mode, the following keystrokes are special:

^? = delete character	^H = delete character
^U = kill line	^W = delete word

CHAPTER 20

Debugging Shared Libraries

dbx provides full debugging support for programs that use dynamically-linked, shared libraries, provided that the libraries are compiled using the -g option.

This chapter is organized into the following sections:

Dynamic Linker

The dynamic linker, also known as rtld, Runtime ld, or ld.so, arranges to bring shared objects (load objects) into an executing application. There are two primary areas where rtld is active:

- Program startup – At program startup, rtld runs first and dynamically loads all shared objects specified at link time. These are *preloaded* shared objects and may include libc.so, libC.so, or libX.so. Use ldd(1) to find out which shared objects a program will load.
- Application requests – The application uses the function calls dlopen(3) and dlclose(3) to dynamically load and unload shared objects or executables.

dbx uses the term *loadobject* to refer to a shared object (.so) or executable (a.out). You can use the loadobject command (see "loadobject Command" on page 345) to list and manage symbolic information from loadobjects.

Link Map

The dynamic linker maintains a list of all loaded objects in a list called a *link map*. The link map is maintained in the memory of the program being debugged, and is indirectly accessed through `librtld_db.so`, a special system library for use by debuggers.

Startup Sequence and `.init` Sections

A `.init` section is a piece of code belonging to a shared object that is executed when the shared object is loaded. For example, the `.init` section is used by the C++ runtime system to call all static initializers in a `.so`.

The dynamic linker first maps in all the shared objects, putting them on the link map. Then, the dynamic linker traverses the link map and executes the `.init` section for each shared object. The `syncrtld` event (see "`syncrtld`" on page 288) occurs between these two phases.

Procedure Linkage Tables

Procedure linkage tables (PLTs) are structures used by the `rtld` to facilitate calls across shared object boundaries. For instance, calls to `printf` go through this indirect table. The details of how this is done can be found in the generic and processor specific SVR4 ABI reference manuals.

For `dbx` to handle `step` and `next` commands across PLTs, it has to keep track of the PLT table of each load object. The table information is acquired at the same time as the `rtld` handshake.

Fix and Continue

Using fix and continue with shared objects loaded with `dlopen()` requires a change in how they are opened for fix and continue to work correctly. Use mode `RTLD_NOW|RTLD_GLOBAL` or `RTLD_LAZY|RTLD_GLOBAL`.

Setting Breakpoints in Shared Libraries

To set a breakpoint in a shared library, dbx needs to know that a program will use that library when it runs, and dbx needs to load the symbol table for the library. To determine which libraries a newly-loaded program will use when it runs, dbx executes the program just long enough for the runtime linker to load all of the starting libraries. dbx then reads the list of loaded libraries and kills the process. The libraries remain loaded and you can set breakpoints in them before rerunning the program for debugging.

dbx follows the same procedure for loading the libraries whether the program is loaded from the command line with the dbx command, from the dbx prompt with the debug command, or from the dbx Debugger in the IDE.

Setting a Breakpoint in a Explicitly Loaded Library

dbx automatically detects that a dlopen() or a dlclose() has occurred and loads the symbol table of the loaded object. Once a shared object has been loaded with dlopen() you can place breakpoints in it and debug it as you would any part of your program.

If a shared object is unloaded using dlclose(), dbx remembers the breakpoints placed in it and replaces them if the shared object is again loaded with dlopen(), even if the application is run again.

However, you do not need to wait for the loading of a shared object with dlopen() to place a breakpoint in it, or to navigate its functions and source code. If you know the name of the shared object that the program being debugged will be loading with dlopen(), you can request that dbx preload its symbol table by using the loadobject -load command:

```
loadobject -load /usr/java1.1/lib/libjava_g.so
```

You can now navigate the modules and functions in this loadobject and place breakpoints in it before it has been loaded with dlopen(). Once the loadobject is loaded by your program, dbx automatically places the breakpoints.

Setting a breakpoint in a dynamically linked library is subject to the following limitations:

- You cannot set a breakpoint in a "filter" library loaded with `dlopen()` until the first function in it is called.
- When a library is loaded by `dlopen()`, an initialization routine named `_init()` is called. This routine might call other routines in the library. `dbx` cannot place breakpoints in the loaded library until after this initialization is completed. In specific terms, this means you cannot have `dbx` stop at `_init()` in a library loaded by `dlopen()`.

APPENDIX **A**

Modifying a Program State

This appendix focuses on `dbx` usage and commands that change your program or change the behavior of your program when you run it under `dbx`, as compared to running it without `dbx`. It is important to understand which commands might make modifications to your program.

The chapter is divided into the following sections:

Impacts of Running a Program Under `dbx`

Your application might behave differently when run under `dbx`. Although `dbx` strives to minimize its impact on the program being debugged, you should be aware of the following:

- You might have forgotten to take out a `-C` or disable RTC. Having the RTC support library `librtc.so` loaded into a program can cause the program to behave differently.
- Your `dbx` initialization scripts might have some environment variables set that you've forgotten about. The stack base starts at a different address when running under `dbx`. This is also different based on your environment and the contents of `argv[]`, forcing local variables to be allocated differently. If they're not initialized, they will get different random numbers. This problem can be detected using runtime checking.
- The program does not initialize memory allocated with `malloc()` before use; a situation similar to the previous one. This problem can be detected using runtime checking.

- `dbx` has to catch LWP creation and `dlopen` events, which might affect timing-sensitive multithreaded applications.
- `dbx` does context switching on signals, so if your application makes heavy use of signals, things might work differently.
- Your program might be expecting that `mmap()` always returns the same base address for mapped segments. Running under `dbx` perturbs the address space sufficiently to make it unlikely that `mmap()` returns the same address as when the program is run without `dbx`. To determine if this is a problem, look at all uses of `mmap()` and ensure that the address returned is used by the program, rather than a hard-coded address.
- If the program is multithreaded, it might contain data races or be otherwise dependent upon thread scheduling. Running under `dbx` perturbs thread scheduling and may cause the program to execute threads in a different order than normal. To detect such conditions, use `lock_lint`.

Otherwise, determine whether running with `adb` or `truss` causes the same problems.

To minimize perturbations imposed by `dbx`, try attaching to the application while it is running in its natural environment.

Commands That Alter the State of the Program

`assign` Command

The `assign` command assigns a value of the *expression* to *variable*. Using it in `dbx` permanently alters the value of *variable*.

```
assign variable = expression
```

`pop` Command

The `pop` command pops a frame or frames from the stack:

`pop`	Pop current frame.
`pop` *number*	Pop *number* frames.
`pop -f` *number*	Pop frames until specified frame *number*.

Any calls popped are re-executed upon resumption, which might result in unwanted program changes. `pop` also calls destructors for objects local to the popped functions.

For more information, see "`pop` Command" on page 357.

`call` Command

When you use the `call` command in `dbx`, you call a procedure and the procedure performs as specified:

```
call proc([params])
```

The procedure could modify something in your program. `dbx` is making the call as if you had written it into your program source.

For more information, see "`call` Command" on page 302.

`print` Command

To print the value of the expression(s), type:

```
print expression, ...
```

If an expression has a function call, the same considerations apply as with the `call` command. With C++, you should also be careful of unexpected side effects caused by overloaded operators.

For more information, see "`print` Command" on page 358.

`when` Command

The `when` command has a general syntax as follows:

```
when event-specification [modifier] {command; ... }
```

When the event occurs, the *command*s are executed.

When you get to a line or to a procedure, a command is performed. Depending upon which command is issued, this could alter your program state.

For more information, see "`when` Command" on page 397.

`fix` Command

You can use the `fix` command to make immediate changes to your program:

```
fix
```

Although is a very useful tool, the `fix` command recompiles modified source files and dynamically links the modified functions into the application.

Make sure to check the restrictions for fix and continue. See Chapter 10.

For more information, see "`fix` Command" on page 332.

`cont at` Command

The `cont at` command alters the order in which the program runs. Execution is continued at line *line. id* is required if the program is multithreaded.

```
cont at line id
```

This could change the outcome of the program.

APPENDIX **B**

Event Management

Event management refers to the capability of `dbx` to perform actions when events take place in the program being debugged. When an event occurs, `dbx` allows you to stop a process, execute arbitrary commands, or print information. The simplest example of an event is a breakpoint (see Chapter 6). Examples of other events are faults, signals, system calls, calls to `dlopen()`, and data changes (see "Setting Data Change Breakpoints" on page 106).

This appendix is organized into the following sections:

Event Handlers

Event management is based on the concept of a *handler*. The name comes from an analogy with hardware interrupt handlers. Each event management command typically creates a handler, which consists of an *event specification* and a series of side-effect actions. (See "Setting Event Specifications" on page 277.) The event specification specifies the event that will trigger the handler.

When the event occurs and the handler is triggered, the handler evaluates the event according to any modifiers included in the event specification. (See "Event Specification Modifiers" on page 289.) If the event meets the conditions imposed by the modifiers, the handler's side-effect actions are performed (that is, the handler "fires").

An example of the association of a program event with a `dbx` action is setting a breakpoint on a particular line.

The most generic form of creating a handler is by using the `when` command.

```
when event-specification {action; ... }
```

Examples in this chapter show how you can write a command (like `stop`, `step`, or `ignore`) in terms of `when`. These examples are meant to illustrate the flexibility of `when` and the underlying *handler* mechanism, but they are not always exact replacements.

Creating Event Handlers

Use the commands `when`, `stop`, and `trace` to create event handlers. (For detailed information, see "`when` Command" on page 397, "`stop` Command" on page 375, and "`trace` Command" on page 387.)

`stop` is shorthand for a common `when` idiom.

```
when event-specification { stop -update; whereami; }
```

An *event-specification* is used by the event management commands `stop`, `when`, and `trace` to specify an event of interest. (see "Setting Event Specifications" on page 277).

Most of the `trace` commands can be handcrafted using the `when` command, `ksh` functionality, and event variables. This is especially useful if you want stylized tracing output.

Every command returns a number known as a handler id (*hid*). You can access this number using the predefined variable `$newhandlerid`.

Manipulating Event Handlers

You can use the following commands to manipulate event handlers. For more information on each command, see the cited section.

- `status` – lists handlers (see "`status` Command" on page 371).
- `delete` – deletes all handlers including temporary handlers (see "`delete` Command" on page 323).
- `clear` – deletes handlers based on breakpoint position (see "`clear` Command" on page 307).
- `handler -enable` – enables handlers (see "`handler` Command" on page 336).
- `handler -disable` – disables handlers.
- `cancel` – cancels signals and lets the process continue (see "`cancel` Command" on page 303).

Using Event Counters

An event handler has a trip counter, which has a count limit. Whenever the specified event occurs, the counter is incremented. The action associated with the handler is performed only if the count reaches the limit, at which point the counter is automatically reset to 0. The default limit is 1. Whenever a process is rerun, all event counters are reset.

You can set the count limit using the `-count` modifier with a `stop`, `when`, or `trace` command (see "`-count n -count infinity`" on page 290). Otherwise, use the `handler` command to individually manipulate event handlers:.

```
handler [ -count | -reset ] hid new-count new-count-limit
```

Setting Event Specifications

Event specifications are used by the `stop`, `when`, and `trace` commands to denote event types and parameters. The format consists of a keyword representing the event type and optional parameters. The meaning of an event specification is

generally identical for all three commands; exceptions are documented in the command descriptions (see "stop Command" on page 375, "trace Command" on page 387, and "when Command" on page 397).

Breakpoint Event Specifications

A breakpoint is a location where an action occurs, at which point the program stops executing. The following are event specifications for breakpoint events.

in *function*

The function has been entered, and the first line is about to be executed. The first executable code after the prolog is used as the actual breakpoint location. This may be a line where a local variable is being initialized. In the case of C++ constructors, execution stops after all base class constructors have executed. If the `-instr` modifier is used (see "-instr" on page 290), it is the first instruction of the function about to be executed. The *function* specification can take a formal parameter signature to help with overloaded function names or template instance specification. For example:

```
stop in mumble(int, float, struct Node *)
```

Note – Do not confuse `in` *function* with the `-in` *function* modifier.

at [*filename:*]*line_number*

The designated line is about to be executed. If you specify *filename*, then the designated line in the specified file is about to be executed. The file name can be the name of a source file or an object file. Although quotation marks are not required, they may be necessary if the file name contains special characters. If the designated line is in template code, a breakpoint is placed on all instances of that template.

at *address_expression*

The instruction at the given address is about to be executed. This event is available only with the `stopi` command (see "stopi Command" on page 380) or with the `-instr` event modifier (see "-instr" on page 290).

`infunction` *function*

Equivalent to `in` *function* for all overloaded functions named *function* or all template instantiations thereof.

`inmember` *function*
`inmethod` *function*

Equivalent to `in` *function* or the member function named *function* for every class.

`inclass` *classname* `[-recurse | -norecurse]`

Equivalent to `in` *function* for all member functions that are members of *classname*, but not any of the bases of *classname*. `-norecurse` is the default. If `-recurse` is specified, the base classes are included.

`inobject` *object-expression* `[-recurse | -norecurse]`

A member function called on the specific object at the address denoted by *object-expression* has been called. `stop inobject` *ox* is roughly equivalent to the following, but unlike `inclass`, bases of the dynamic type of *ox* are included. `-recurse` is the default. If `-norecurse` is specified, the base classes are not included.

```
stop inclass dynamic_type(ox) -if this==ox
```

Data Change Event Specifications

The following are event specifications for events that involve access or change to the contents of a memory address.

`access` *mode address-expression* [, *byte-size-expression*]

The memory specified by *address-expression* has been accessed.

mode specifies how the memory was accessed. It can be composed of one or all of the letters:

`r`	The memory at the specified address has been read.
`w`	The memory has been written to.
`x`	The memory has been executed.

mode can also contain either of the following:

`a`	Stops the process after the access (default).
`b`	Stops the process before the access.

In both cases the program counter will point at the offending instruction. The "before" and "after" refer to the side effect.

address-expression is any expression that can be evaluated to produce an address. If you give a symbolic expression, the size of the region to be watched is automatically deduced; you can override it by specifying *byte-size-expression*. You can also use nonsymbolic, typeless address expressions; in which case, the size is mandatory. For example:

```
stop access w 0x5678, sizeof(Complex)
```

The `access` command has the limitation that no two matched regions may overlap.

Note – The `access` event specification is a replacement for the `modify` event specification. While both syntaxes work on Solaris 2.6, Solaris 7, and Solaris 8, on all of these operating environments except Solaris 2.6, `access` suffers the same limitations as `modify` and accepts only a mode of `wa`.

`change` *variable*

The value of *variable* has changed. The `change` event is roughly equivalent to:

```
when step { if [ $last_value !=$[variable]] then
                  stop
            else
                  last_value=$[variable]
          }
```

This event is implemented using single-stepping. For faster performance, use the `access` event (see "`access` mode address-expression [, byte-size-expression]" on page 279).

The first time *variable* is checked causes one event, even though no change is detected. This first event provides access to the initial value of *variable*. Subsequent detected changes in the value of *variable* trigger additional events.

`cond` *condition-expression*

The condition denoted by *condition-expression* evaluates to true. You can specify any expression for *condition-expression*, but it must evaluate to an integral type. The cond event is roughly equivalent to:

`stop step -if` *conditional_expression*

System Event Specifications

The following are event specifications for system events.

`dlopen [` *lib-path* `] | dlclose [` *lib-path* `]`

These events occur after a `dlopen`() or a `dlclose`() call succeeds. A `dlopen`() or `dlclose`() call can cause more than one library to be loaded. The list of these libraries is always available in the predefined variable `$dllist`. The first shell word in `$dllist` is a "+" or a "-", indicating whether the list of libraries is being added or deleted.

lib-path is the name of a shared library. If it is specified, the event occurs only if the given library was loaded or unloaded. In that case, `$dlobj` contains the name of the library. `$dllist` is still available.

If *lib-path* begins with a `/`, a full string match is performed. Otherwise, only the tails of the paths are compared.

If *lib-path* is not specified, then the events always occur whenever there is any `dl`-activity. In this case, `$dlobj` is empty but `$dllist` is valid.

fault *fault*

The fault event occurs when the specified fault is encountered. The faults are architecture-dependent. The following set of faults known to dbx is defined in the proc(4) man page.

Fault	Description
FLTILL	Illegal instruction
FLTPRIV	Privileged instruction
FLTBPT*	Breakpoint trap
FLTTRACE*	Trace trap (single step)
FLTACCESS	Memory access (such as alignment)
FLTBOUNDS	Memory bounds (invalid address)
FLTIOVF	Integer overflow
FLTIZDIV	Integer zero divide
FLTPE	Floating-point exception
FLTSTACK	Irrecoverable stack fault
FLTPAGE	Recoverable page fault
FLTWATCH*	Watchpoint trap
FLTCPCOVF	CPU performance counter overflow

Note – BPT, TRACE, and BOUNDS are used by dbx to implement breakpoints and single-stepping. Handling them might interfere with how dbx works.

These faults are taken from /sys/fault.h. *fault* can be any of those listed above, in uppercase or lowercase, with or without the FLT- prefix, or the actual numerical code.

Note – The fault event is not available on Linux platforms.

lwp_exit

The lwp_exit event occurs when lwp has been exited. $lwp contains the id of the exited LWP (lightweight process) for the duration of the event handler.

Note – The `lwpexit` event is not available on Linux platforms.

`sig` *signal*

The `sig` *signal* event occurs when the signal is first delivered to the program being debugged. *signal* can be either a decimal number or the signal name in uppercase or lowercase; the prefix is optional. This is completely independent of the `catch` and `ignore` commands, although the `catch` command can be implemented as follows:

```
function simple_catch {
       when sig $1 {
              stop;
              echo Stopped due to $sigstr $sig
              whereami
       }
}
```

Note – When the `sig` event is received, the process has not seen it yet. Only if you continue the process with the specified signal is the signal forwarded to it.

`sig` *signal sub-code*

When the specified signal with the specified *sub-code* is first delivered to the child, the `sig` *signal sub-code* event occurs. As with signals, you can type the *sub-code* as a decimal number, in uppercase or lowercase; the prefix is optional.

`sysin` *code* | *name*

The specified system call has just been initiated, and the process has entered kernel mode.

The concept of system call supported by `dbx` is that provided by traps into the kernel as enumerated in `/usr/include/sys/syscall.h`.

This is not the same as the ABI notion of system calls. Some ABI system calls are partially implemented in user mode and use non-ABI kernel traps. However, most of the generic system calls (the main exception being signal handling) are the same between `syscall.h` and the ABI.

Note – The `sysin` event is not available on Linux platforms.

Note – The list of kernel system call traps in `/usr/include/sys/syscall.h` is part of a private interface in the Solaris operating environment that changes from release to release. The list of trap names (codes) and trap numbers that `dbx` accepts includes all of those supported by any of the versions of the Solaris operating environment that `dbx` supports. It is unlikely that the names supported by `dbx` exactly match those of any particular release of the Solaris operating environment, and some of the names in `syscall.h` might not be available. Any trap number (code) is accepted by `dbx` and works as expected, but a warning is issued if it does not correspond to a known system call trap.

`sysout` *code* | *name*

The specified system call is finished, and the process is about to return to user mode.

Note – The `sysout` event is not available on Linux platforms.

`sysin` | `sysout`

Without arguments, all system calls are traced. Certain `dbx` features, for example, the `modify` event and runtime checking, cause the child to execute system calls for its own purposes and show up if traced.

Execution Progress Event Specifications

The following are event specifications for events pertaining to execution progress.

`exit` *exitcode*

The `exit` event occurs when the process has exited.

next

The `next` event is similar to the `step` event except that functions are not stepped into.

returns

The `returns` event is a breakpoint at the return point of the current *visited* function. The visited function is used so that you can use the `returns` event specification after giving a number of `step up` commands. The `returns` event is always `-temp` and can only be created in the presence of a live process.

returns *function*

The `returns` *function* event executes each time the given function returns to its call site. This is not a temporary event. The return value is not provided, but you can find integral return values by accessing the following registers:

Sparc	`$o0`
Intel	`$eax`

The event is roughly equivalent to:

```
when in func { stop returns; }
```

step

The `step` event occurs when the first instruction of a source line is executed. For example, you can get simple tracing with:

```
when step { echo $lineno: $line; }; cont
```

When enabling a `step` event, you instruct `dbx` to single step automatically next time the `cont` command is used.

Note – The `step` (and `next`) events do not occur upon the termination of the `step` command. The `step` command is implemented in terms of the `step` event roughly as follows:

```
alias step="when step -temp { whereami; stop; }; cont"
```

Other Event Specifications

The following are event specifications for other types of events.

attach

`dbx` has successfully attached to a process.

detach

`dbx` has successfully detached from the program being debugged.

lastrites

The process being debugged is about to expire, which can happen for the following reasons:

- The `_exit`(2) system call has been called. (This happens either through an explicit call or when `main`() returns.)
- A terminating signal is about to be delivered.
- The process is being killed by the `kill` command.

The final state of the process is usually, but not always, available when this event is triggered, giving you your last opportunity to examine the state of the process. Resuming execution after this event terminates the process.

Note – The `lastrites` event is not available on Linux platforms.

proc_gone

The proc_gone event occurs when dbx is no longer associated with a debugged process. The predefined variable $reason may be signal, exit, kill, or detach.

prog_new

The prog_new event occurs when a new program has been loaded as a result of follow exec.

Note – Handlers for this event are always permanent.

---

stop

The process has stopped. The stop event occurs whenever the process stops such that the user receives a prompt, particularly in response to a stop handler. For example, the following commands are equivalent:

```
display x
when stop {print x;}
```

sync

The process being debugged has just been executed with exec(). All memory specified in a.out is valid and present, but preloaded shared libraries have not been loaded. For example, printf, although available to dbx, has not been mapped into memory.

A stop on this event is ineffective; however, you can use the sync event with the when command.

Note – The sync event is not available on Linux platforms.

---

syncrtld

The `syncrtld` event occurs after a `sync` (or `attach` if the process being debugged has not yet processed shared libraries). It executes after the dynamic linker startup code has executed and the symbol tables of all preloaded shared libraries have been loaded, but before any code in the `.init` section has run.

A `stop` on this event is ineffective; however, you can use the `syncrtld` event with the `when` command.

throw

The `throw` event occurs whenever any exception that is not unhandled or unexpected is thrown by the application.

Note – The `throw` event is not available on Linux platforms.

---

throw *type*

If an exception *type* is specified with the `throw` event, only exceptions of that type cause the `throw` event to occur.

throw -unhandled

`-unhandled` is a special exception type signifying an exception that is thrown but for which there is no handler.

throw -unexpected

`-unexpected` is a special exception type signifying an exception that does not satisfy the exception specification of the function that threw it.

timer *seconds*

The `timer` event occurs when the program being debugged has been running for *seconds*. The timer used with this event is shared with `collector` command. The resolution is in milliseconds, so a floating point value for *seconds*, for example `0.001`, is acceptable.

Event Specification Modifiers

An event specification modifier sets additional attributes of a handler, the most common kind being event filters. Modifiers must appear after the keyword portion of an event specification. A modifier begins with a dash (-). The following are the valid event specification modifiers.

`-if` *condition*

The condition is evaluated when the event specified by the event specification occurs. The side effect of the handler is allowed only if the condition evaluates to nonzero.

If the `-if` modifier is used with an event that has an associated singular source location, such as `in` or `at`, *condition* is evaluated in the scope corresponding to that location. Otherwise, qualify it with the desired scope.

`-resumeone`

The `-resumeone` modifier can be used with the `-if` modifier in an event specification for a multithreaded program, and causes only one thread to be resumed if the condition contains function calls. For more information, see "Using a Filter With a Conditional Event" on page 110.

`-in` *function*

The event triggers only if it occurs between the time the first instruction of the given *function* is reached and the time the function returns. Recursion on the function are ignored.

`-disable`

The `-disable` modifier creates the handler in the disabled state.

-count *n*
-count infinity

The -count *n* and -count infinity modifiers have the handler count from 0 (see "Using Event Counters" on page 277). Each time the event occurs, the count is incremented until it reaches *n*. Once that happens, the handler fires and the counter is reset to zero.

Counts of all enabled handlers are reset when a program is run or rerun. More specifically, they are reset when the sync event occurs.

The count is reset when you begin debugging a new program with the debug -r command (see "debug Command" on page 320) or the attach -r command (see "attach Command" on page 300).

-temp

Creates a temporary handler. Once the event has occurred it is automatically deleted. By default, handlers are not temporary. If the handler is a counting handler, it is automatically deleted only when the count reaches 0 (zero).

Use the delete -temp command to delete all temporary handlers.

-instr

Makes the handler act at an instruction level. This event replaces the traditional 'i' suffix of most commands. It usually modifies two aspects of the event handler:

- Any message prints assembly-level rather than source-level information.
- The granularity of the event becomes instruction level. For instance, step -instr implies instruction-level stepping.

-thread *thread_id*

The action is executed only if the thread that caused the event matches *thread_id*. The specific thread you have in mind might be assigned a different *thread_id* from one execution of the program to the next.

-lwp *lwp_id*

The action is executed only if the thread that caused the event matches *lwp_id*. The action is executed only if the thread that caused the event matches *lwp_id*. The specific thread you have in mind might be assigned a different *lwp_id* from one execution of the program to the next.

-hidden

Hides the handler in a regular `status` command. Use `status -h` to see hidden handlers.

-perm

Normally all handlers are thrown away when a new program is loaded. Using the `-perm` modifier retains the handler across debugging sessions. A plain `delete` command does not delete a permanent handler. Use `delete -p` to delete a permanent handler.

Parsing and Ambiguity

The syntax for event specifications and modifiers is:

- Keyword driven
- Based on `ksh` conventions; everything is split into words delimited by spaces

Expressions can have spaces embedded in them, causing ambiguous situations. For example, consider the following two commands:

```
when a -temp
when a-temp
```

In the first example, even though the application might have a variable named *temp*, the `dbx` parser resolves the event specification in favor of `-temp` being a modifier. In the second example, `a-temp` is collectively passed to a language-specific expression parser. There must be variables named *a* and *temp* or an error occurs. Use parentheses to force parsing.

Using Predefined Variables

Certain read-only `ksh` predefined variables are provided. The following variables are always valid:

Variable	Definition
`$ins`	Disassembly of the current instruction.
`$lineno`	Current line number in decimal.
`$vlineno`	Current “visiting” line number in decimal.
`$line`	Contents of the current line.
`$func`	Name of the current function.
`$vfunc`	Name of the current “visiting” function.
`$class`	Name of the class to which `$func` belongs.
`$vclass`	Name of the class to which `$vfunc` belongs.
`$file`	Name of the current file.
`$vfile`	Name of the current file being visited.
`$loadobj`	Name of the current loadable object.
`$vloadobj`	Name of the current loadable object being visited.
`$scope`	Scope of the current PC in back-quote notation.
`$vscope`	Scope of the visited PC in back-quote notation.
`$funcaddr`	Address of `$func` in hex.
`$caller`	Name of the function calling `$func`.
`$dllist`	After a `dlopen` or `dlclose` event, contains the list of load objects just loaded or unloaded. The first word of `dllist` is a “+” or a “-” depending on whether a `dlopen` or a `dlclose` has occurred.
`$newhandlerid`	ID of the most recently created handler. This variable has an undefined value after any command that deletes handlers. Use the variable immediately after creating a handler. `dbx` cannot capture all of the handler IDs for a command that creates multiple handlers.
`$firedhandlers`	List of handler ids that caused the most recent stoppage. The handlers on the list are marked with “*” in the output of the `status` command.
`$proc`	Process ID of the current process being debugged.
`$lwp`	Lwp ID of the current LWP.

Variable	Definition
`$thread`	Thread ID of the current thread.
`$prog`	Full path name of the program being debugged.
`$oprog`	Previous value of `$prog`, which is used to get back to what you were debugging following an `exec()`, when `$prog` reverts to "-". While `$prog` is expanded to a full path name, `$oprog` contains the program path as specified on the command line or to the `debug` command. If `exec()` is called more than once, there is no way to return to the original program.
`$exitcode`	Exit status from the last run of the program. The value is an empty string if the process has not exited.

As an example, consider that `whereami` can be implemented as:

```
function whereami {
  echo Stopped in $func at line $lineno in file $(basename $file)
  echo "$lineno\t$line"
}
```

Variables Valid for `when` Command

The following variables are valid only within the body of a `when` command.

`$handlerid`

During the execution of the body, `$handlerid` is the id of the `when` command to which the body belongs. These commands are equivalent:

```
when X -temp { do_stuff; }
when X  { do_stuff; delete $handlerid; }
```

`$booting`

`$booting` is set to `true` if the event occurs during the *boot* process. Whenever a new program is debugged, it is first run without the user's knowledge so that the list and location of shared libraries can be ascertained. The process is then killed. This sequence is termed booting.

While booting is occurring, all events are still available. Use this variable to distinguish the `sync` and the `syncrtld` events occurring during a `debug` and the ones occurring during a normal `run`.

Variables Valid for Specific Events

Certain variables are valid only for specific events as shown in the following tables.

TABLE B-1 Variables Valid for `sig` Event

Variable	Description
`$sig`	Signal number that caused the event
`$sigstr`	Name of `$sig`
`$sigcode`	Subcode of `$sig` if applicable
`$sigcodestr`	Name of `$sigcode`
`$sigsender`	Process ID of sender of the signal, if appropriate

TABLE B-2 Variable Valid for `exit` Event

Variable	Description
`$exitcode`	Value of the argument passed to `_exit(2)` or `exit(3)` or the return value of `main`

TABLE B-3 Variable Valid for `dlopen` and `dlclose` Events

Variable	Description
`$dlobj`	Pathname of the load object dlopened or dlclosed

TABLE B-4 Variables Valid for `sysin` and `sysout` Events

Variable	Description
`$syscode`	System call number
`$sysname`	System call name

TABLE B-5 Variable Valid for `proc_gone` Events

Variable	Description
`$reason`	One of signal, exit, kill, or detach

Setting Event Handler Examples

The following are some examples of setting event handlers.

Setting a Breakpoint for Store to an Array Member

To set a breakpoint on `array[99]`, type:

```
(dbx) stop access w &array[99]
(2) stop access w &array[99], 4
(dbx) run
Running: watch.x2
watchpoint array[99] (0x2ca88[4]) at line 22 in file "watch.c"
   22  array[i] = i;
```

Implementing a Simple Trace

To implement a simple trace, type:

```
(dbx) when step { echo at line $lineno; }
```

Enabling a Handler While Within a Function (`in` *function*)

To enable a handler while within a function, type:

```
<dbx> trace step -in foo
```

This is equivalent to:

```
# create handler in disabled state
when step -disable { echo Stepped to $line; }
t=$newhandlerid    # remember handler id
when in foo {
 # when entered foo enable the trace
 handler -enable "$t"
 # arrange so that upon returning from foo,
 # the trace is disabled.
 when returns { handler -disable "$t"; };
}
```

Determining the Number of Lines Executed

To see how many lines have been executed in a small program, type:

```
(dbx) stop step -count infinity # step and stop when count=inf
(2) stop step -count 0/infinity
(dbx) run
...
(dbx) status
(2) stop step -count 133/infinity
```

The program never stops—the program terminates. The number of lines executed is 133. This process is very slow. It is most useful with breakpoints on functions that are called many times.

Determining the Number of Instructions Executed by a Source Line

To count how many instructions a line of code executes, type:

```
(dbx) ...                          # get to the line in question
(dbx) stop step -instr -count infinity
(dbx) step ...
(dbx) status
(3) stop step -count 48/infinity # 48 instructions were executed
```

If the line you are stepping over makes a function call, the lines in the function are counted as well. You can use the `next` event instead of `step` to count instructions, excluding called functions.

Enabling a Breakpoint After an Event Occurs

Enable a breakpoint only after another event has occurred. For example, if your program begins to execute incorrectly in function `hash`, but only after the 1300'th symbol lookup, you would type:

```
(dbx) when in lookup -count 1300 {
       stop in hash
       hash_bpt=$newhandlerid
       when proc_gone -temp { delete $hash_bpt; }
}
```

Note – `$newhandlerid` is referring to the just executed `stop in` command.

Resetting Application Files for `replay`

If your application processes files that need to be reset during a `replay`, you can write a handler to do that each time you run the program:

```
(dbx) when sync { sh regen ./database; }
(dbx) run < ./database...# during which database gets clobbered
(dbx) save
...             # implies a RUN, which implies the SYNC event which
(dbx) restore   # causes regen to run
```

Checking Program Status

To see quickly where the program is while it is running, type:

```
(dbx) ignore sigint
(dbx) when sig sigint { where; cancel; }
```

Then type ^C to see a stack trace of the program without stopping it.

This is basically what the collector hand sample mode does (and more). Use SIGQUIT (^\) to interrupt the program because ^C is now used up.

Catch Floating Point Exceptions

To catch only specific floating point exceptions, for example, IEEE underflow, type:

```
(dbx) ignore FPE                # turn off default handler
(dbx) help signals | grep FPE   # can't remember the subcode name
...
(dbx) stop sig fpe FPE_FLTUND
...
```

APPENDIX C

Command Reference

This appendix gives detailed syntax and functional descriptions of all of the `dbx` commands.

`adb` Command

The `adb` command executes an adb-style command or sets `adb` mode. It is valid only in native mode.

Syntax

`adb` *adb-command*	Executes an adb-style command.
`adb`	Sets adb mode; use `$q` to exit adb mode.

`assign` Command

In native mode, the `assign` command assigns a new value to a program variable. In Java mode, the `assign` command assigns a new value to a local variable or parameter.

Native Mode Syntax

`assign` *variable* `=` *expression*

where:

expression is the value to be assigned to *variable*.

Java Mode Syntax

`assign` *identifier* `=` *expression*

where:

class_name is the name of a Java class, using either the package path (using period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`) or the full path name (preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers; for example, `#test1/extra/T1$Inner`). Enclose *class_name* in quotation marks if you use the `$` qualifier.

expression is a valid Java expression.

field_name is the name of a field in the class.

identifier is a local variable or parameter, including `this`, the current class instance variable (*object_name.field_name*) or a class (static) variable (*class_name.field_name*).

object_name is the name of a Java object.

`attach` Command

The `attach` command attaches `dbx` to a running process, stopping execution and putting the program under debugging control. It has identical syntax and identical functionality in native mode and Java mode.

Syntax

`attach` *process_id*	Begin debugging the program with process ID *process_id*. `dbx` finds the program using `/proc`.
`attach -p` *process_id program_name*	Begin debugging *program* with process ID *process_id*.
`attach` *program_name process_id*	Begin debugging *prog*ram with process ID *process_id*. *program* may be `-`; `dbx` finds it using `/proc`.
`attach -r ...`	The `-r` option causes `dbx` to retain all `display`, `trace`, `when`, and `stop` commands. With no `-r` option, an implicit `delete all` and `undisplay 0` are performed.

where:

process_id is the process ID of a running process.

program_name is the path name of the running program.

To attach to a running Java process:

1. **Ensure that the JVM™ software can find `libdbxagent.so` by adding `libdbxagent.so` to your `LD_LIBRARY_PATH`. `libdbxagent.so` is located in your installation directory at:**

   installation_directory`/SUNWspro/lib` for 32-bit applications

   installation_directory`/SUNWspro/lib/v9` for 64-bit applications.

2. **Start your Java application by typing:**

   `java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent` *myclass*`.class`

3. **Then you can attach to the process by starting `dbx` with the process id:**

   `dbx -`*process_id*

`bsearch` Command

The `bsearch` command searches backward in the current source file. It is valid only in native mode.

Syntax

`bsearch` *string*	Search backward for *string* in the current file.
`bsearch`	Repeat search, using the last search string.

where:

string is a character string.

`call` Command

In native mode, the `call` command calls a procedure. In Java mode, the `call` command calls a method.

Native Mode Syntax

`call` *procedure*`([`*parameters*`])`

where:

procedure is the name of the procedure.

parameters are the procedure's parameters.

You can also use the `call` command to call a function; to see the return value use the `print` command (see "`print` Command" on page 358).

Occasionally the called function hits a breakpoint. You can choose to continue using the `cont` command (see "`cont` Command" on page 316), or abort the call by using `pop -c` (see "`pop` Command" on page 357). The latter is useful also if the called function causes a segmentation fault.

Java Mode Syntax

`call [`*class_name*`.|`*object_name*`.]`*method_name*`([`*parameters*`])`

where:

class_name is the name of a Java class, using either the package path (using period (`.`) as a qualifier for example, `test1.extra.T1.Inner`) or the full path name (preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers; for example, `#test1/extra/T1$Inner`). Enclose *class_name* in quotation marks if you use the `$` qualifier.

object_name is the name of a Java object.

method_name is the name of a Java method.

parameters are the method's parameters.

Occasionally the called method hits a breakpoint. You can choose to continue using the `cont` command (see "`cont` Command" on page 316), or abort the call by using `pop -c` (see "`pop` Command" on page 357). The latter is useful also if the called method causes a segmentation fault.

`cancel` Command

The `cancel` command cancels the current signal. It is primarily used within the body of a `when` command (see "`when` Command" on page 397). It is valid only in native mode.

Signals are normally cancelled when `dbx` stops because of a signal. If a `when` command is attached to a signal event, the signal is not automatically cancelled. The `cancel` command can be used to explicitly cancel the signal.

Syntax

```
cancel
```

`catch` Command

The `catch` command catches the given signal(s). It is valid only in native mode.

Catching a given signal causes `dbx` to stop the program when the process receives that signal. If you continue the program at that point, the signal is not processed by the program.

Syntax

`catch`	Print a list of the caught signals.
`catch` *number number*`...`	Catch signal(s) numbered *number.*
`catch` *signal signal*`...`	Catch signal(s) named by *signal*. SIGKILL cannot be caught or ignored.
`catch $(ignore)`	Catch all signals.

where:

number is the number of a signal.

signal is the name of a signal.

`check` Command

The `check` command enables checking of memory access, leaks, or usage, and prints the current status of runtime checking (RTC). It is valid only in native mode.

Note – The `check` command is available only on Solaris platforms.

The features of runtime checking that are enabled by this command are reset to their initial state by the `debug` command.

Syntax

`check -access`

Turn on access checking. RTC reports the following errors:

baf	Bad free
duf	Duplicate free
maf	Misaligned free
mar	Misaligned read
maw	Misaligned write
oom	Out of memory
rua	Read from unallocated memory
rui	Read from uninitialized memory
wro	Write to read-only memory
wua	Write to unallocated memory

The default behavior is to stop the process after detecting each access error, which can be changed using the `rtc_auto_continue` `dbx` environment variable. When set to `on` access errors are logged to a file (the file name is controlled by the `dbx` environment variable `rtc_error_log_file_name`). See "`dbxenv` Command" on page 320.

By default each unique access error is only reported the first time it happens. Change this behavior using the `dbx` environment variable `rtc_auto_suppress` (the default setting of this variable is `on`). See "`dbxenv` Command" on page 320.

`check -leaks [-frames n] [-match m]`

Turn on leak checking. RTC reports the following errors:

aib	Possible memory leak - only pointer points in the middle of the block
air	Possible memory leak - pointer to the block exists only in register
mel	Memory leak - no pointers to the block

With leak checking turned on, you get an automatic leak report when the program exits. All leaks including possible leaks are reported at that time. By default, a non-verbose report is generated (controlled by the `dbx` environment variable `rtc_mel_at_exit`). However, you can ask for a leak report at any time (see "`showleaks` Command" on page 369).

`-frames` *n* implies that up to *n* distinct stack frames are displayed when reporting leaks. `-match` *m* is used for combining leaks; if the call stack at the time of allocation for two or more leaks matches *n* frames, then these leaks are reported in a single combined leak report.

The default value of *n* is 8 or the value of *m* (whichever is larger). Maximum value of *n* is 16. The default value of *m* is 3 for C++, and 2 otherwise.

`check -memuse [-frames n] [-match m]`

Turn on memory use (memuse) checking. `check -memuse` also implies check -leaks. In addition to a leak report at program exit, you also get a blocks in use (biu) report. By default a non-verbose blocks in use report is generated (controlled by the `dbx` environment variable `rtc_biu_at_exit`) At any time during program execution you can see where the memory in your program has been allocated (see "`showmemuse` Command" on page 370).

`-frames` *n* implies that up to *n* distinct stack frames will be displayed while reporting memory use and leaks. `-match` *m* is used for combining these reports; if the call stack at the time of allocation for two or more leaks matches *m* frames, then these are reported in a single combined memory leak report.

The default value of *n* is 8 or the value of *m* (whichever is larger). Maximum value of *n* is 16. The default value of *m* is 3 for C++, and 2 otherwise. See `check -leaks` as well.

`check -all [-frames n] [-match m]`

Equivalent to `check -access` or `check -memuse [-frames n] [-match m]'`

The value of the `dbx` environment variable `rtc_biu_at_exit` is not changed with `check -all`, so by default no memory use report is generated at exit. See "`dbx` Command" on page 317 for the description of the `rtc_biu_at_exit` environment variable.

`check [functions] [files] [loadobjects]`

Equivalent to `check -all` or `suppress all` or `unsuppress all` in *functions*, *files*, and *loadobjects*

where:

functions is one or more function names.

files is one or more file names.

loadobjects is one or more loadobject names

You can use this to focus runtime checking on places of interest.

Note – To detect all errors, RTC does not require the program be compiled `-g`. However, symbolic (`-g`) information is sometimes needed to guarantee the correctness of certain errors (mostly read from uninitialized memory). For this reason certain errors (`rui` for `a.out` and `rui + aib + air` for shared libraries) are suppressed if no symbolic information is available. This behavior can be changed by using `suppress` and `unsuppress`.

`clear` Command

The `clear` command clears breakpoints. It is valid only in native mode.

Event handlers created using the `stop, trace`, or `when` command with the `inclass`, `inmethod`, or `infunction` argument create sets of breakpoints. If the *line* you specify in the `clear` command matches one of these breakpoints, only that breakpoint is cleared. Once cleared in this manner an individual breakpoint belonging to a set cannot be enabled again. However, disabling and then enabling the relevant event handler will reestablish all the breakpoints.

Syntax

`clear`	Clear all breakpoints at the current stopping point.
`clear` *line*	Clear all breakpoints at *line*.
`clear` *filename*`:`*line*	Clear all breakpoints at *line* in *filename*.

where:

line is the number of a source code line.

filename is the name of a source code file.

`collector` Command

The `collector` command collects performance data for analysis by the Performance Analyzer. It is valid only in native mode.

Note – The `collector` command is available only on Solaris platforms.

Syntax

`collector` *command_list*	Specify one or more of the collector commands
`archive` *options*	Specify the mode for archiving an experiment when it terminates (see "`collector archive` Command" on page 309).
`dbxsample` *options*	Control the collection of samples when `dbx` stops the target process (see "`collector dbxsample` Command" on page 310).
`disable`	Stop data collection and close the current experiment (see "`collector disable` Command" on page 310).
`enable`	Enable the collector and open a new experiment (see "`collector enable` Command" on page 310).
`heaptrace` *options*	Enable or disable collection of heap tracing data (see "`collector heaptrace` Command" on page 311)
`hwprofile` *options*	Specify hardware counter profiling settings (see "`collector hwprofile` Command" on page 311).
`limit` *options*	Limit the amount of profiling data recoreded (see "`collector limit` Command" on page 312).
`mpitrace` *options*	Enables or disables collection of MPI tracing data (see "`collector mpitrace` Command" on page 312).
`pause`	Stop collecting performance data but leave experiment open (see "`collector pause` Command" on page 312).

`profile` *options*	Specify settings for collecting callstack profiling data (see "`collector profile` Command" on page 313).
`resume`	Start performance data collection after pause (see "`collector resume` Command" on page 313).
`sample` *options*	Specify sampling settings (see "`collector sample` Command" on page 313).
`show` *options*	Show current collector settings (see "`collector show` Command" on page 314).
`status`	Inquire status about current experiment (see "`collector status` Command" on page 314).
`store` *options*	Experiment file control and settings (see "`collector store` Command" on page 315).
`synctrace` *options*	Specify settings for collecting thread synchronization wait tracing data (see "`collector synctrace` Command" on page 315).
`version`	Report the version of `libcollector.so` that would be used to collect data (see).

where:

options are the settings that can be specified with each command.

To start collecting data, type either `collector enable`.

To turn off data collection, type `collector disable`.

`collector archive` Command

The `collector archive` command specifies the archiving mode to be used when the experiment terminates.

Syntax

`collector archive on|off|copy`	By default, normal archiving is used. For no archiving, specify `off`. To copy loadobjects into the experiment for portability, specify `copy`.

`collector dbxsample` Command

The `collector dbxsample` command specifies whether or not to record a sample when the process is stopped by `dbx`.

Syntax

`collector dbxsample on|off`	By default, a sample is collected when the process is stopped by `dbx`. To not collect a sample at this time, specify `off`.

`collector disable` Command

The `collector disable` command causes the data collection to stop and the current experiment to be closed.

Syntax

```
collector disable
```

`collector enable` Command

The `collector enable` command enables the collector and opens a new experiment.

Syntax

```
collector enable
```

`collector heaptrace` Command

The `collector heaptrace` command specifies options for collecting heap tracing (memory allocation) data.

Syntax

`collector heaptrace on|off`	By default, heap tracing data is not collected. To collect this data, specify `on`.

`collector hwprofile` Command

The `collector hwprofile` command specifies options for collecting hardware-counter overflow profiling data.

Syntax

`collector hwprofile on|off`	By default, hardware-counter overflow profile data is not collected. To collect this data, specify `on`.
`collector hwprofile list`	Print out the list of available counters.
`collector hwprofile counter` *name interval* [*name2 interval2*]	Specify hardware counter name(s) and interval(s).

where:

name is the name of a hardware counter.

interval is the collection interval in milliseconds.

name2 is the name of a second hardware counter.

interval2 is the collection interval in milliseconds.

Hardware counters are system-specific, so the choice of counters available to you depends on the system you are using. Many systems do not support hardware-counter overflow profiling. On these machines, the feature is disabled.

`collector limit` Command

The `collector limit` command specifies the experiment file size limit.

Syntax

`collector limit` *value*

where:

value, in megabytes, limits the amount of profiling data recorded. When the limit is reached, no more profiling data is recorded but the experiment remains open and sample points continue to be recorded. The default limit on the amount of data recorded is 2000 Mbytes.

`collector mpitrace` Command

The `collector heaptrace` command specifies options for collecting MPI tracing data.

Syntax

`collector mpitrace on|off`	By default, MPI tracing data is not collected. To collect this data, specify `on`.

`collector pause` Command

The `collector pause` command causes the data collection to stop but leaves the current experiment open. Data collection can be resumed with the `collector resume` command (see "`collector resume` Command" on page 313).

Syntax

`collector pause`

`collector profile` Command

The collector profile command specifies options for collecting profile data.

Syntax

`collector profile on|off`	Specify profile data collection mode
`collector profile timer` *interval*	Specify profile timer period, fixed or floating point, with an optional trailing `m` for milliseconds or `u` for microseconds.

`collector resume` Command

The `collector resume` command causes the data collection to resume after a pause created by the `collector pause` command (see "`collector pause` Command" on page 312).

Syntax

```
collector resume
```

`collector sample` Command

The `collector sample` command specifies the sampling mode and the sampling interval.

Syntax

`collector sample periodic|manual`	Specify sampling mode.
`collector sample period` *seconds*	Specify sampling interval in *seconds*.
`collector sample record [` *name* `]`	Record a sample with an optional *name*.

where:

seconds is the length of the sampling interval.

name is the name of the sample.

`collector show` Command

The `collector show` command shows the settings of one or more categories of options.

Syntax

`collector show all`	Show all settings.
`collector show archive`	Show archive setting.
`collector show profile`	Show callstack profiling settings.
`collector show synctrace`	Show thread synchronization wait tracing settings.
`collector show hwprofile`	Show hardware counter data settings.
`collector show heaptrace`	Show heap tracing data settings.
`collector show limit`	Show experiment size limits.
`collector show mpitrace`	Show MPI trace data settings.
`collector show sample`	Show sample settings.
`collector show store`	Show store settings.

`collector status` Command

The `collector status` command inquires about the status of the current experiment.

Syntax

```
collector status
```

`collector store` Command

The `collector store` command specifies the directory and file name where an experiment is stored.

Syntax

`collector store directory` *pathname*	Specify directory where experiment is stored.
`collector store filename` *filename*	Specify experiment file name.
`collector store group` *string*	Specify experiment group name

where:

pathname is the pathname of the directory where an experiment is to be stored.

filename is the name of the experiment file

string is the name of an experiment group.

`collector synctrace` Command

The `collector synctrace` command specifies options for collecting synchronization wait tracing data.

Syntax

`collector synctrace on|off`	By default, thread synchronization wait tracing data is not collected. To collect this data, specify on.
`collector threshold` *microseconds*	Specify threshold in microseconds. The default value is 1000.
`collector threshold calibrate`	Threshold value will be automatically calculated

where:

microseconds is the threshold below which synchronization wait events are discarded.

`collector version` Command

The `collector version` command reports the version of `libcollector.so` that would be used to collect data.

Syntax

```
collector version on|off
```

`cont` Command

The `cont` command causes the process to continue execution. It has identical syntax and identical functionality in native mode and Java mode.

Syntax

`cont`	Continue execution. In an MT process all threads are resumed. Use Control-C to stop executing the program.
`cont ... -sig` *signal*	Continue execution with signal *signal*.
`cont ...` *id*	The *id* specifies which thread or LWP to continue.
`cont at` *line* `[` *id* `]`	Continue execution at line *line*. *id* is required if the application is multi-threaded.
`cont ... -follow parent|child|both`	If the `dbx follow_fork_mode` environment variable is set to ask, and you have chosen `stop`, use this option to choose which process to follow. both is only applicable under the dbx Debugger.

`dalias` Command

The `dalias` command defines a dbx-style (csh-style) alias. It is valid only in native mode.

Syntax

`dalias`	(`dbx` alias) List all currently defined aliases
`dalias` *name*	List the definition, if any, of alias *name*.
`dalias` *name definition*	Define *name* to be an alias for *definition*. *definition* may contain white space. A semicolon or newline terminates the definition.

where:

name is the name of an alias

definition is the definition of an alias.

`dbx` accepts the following csh history substitution meta-syntax, which is commonly used in aliases:

!:<n>

!-<n>

!^

!$

!*

The ! usually needs to be preceded by a backslash. For example:

```
dalias goto "stop at \!:1; cont; clear"
```

For more information, see the `csh`(1) man page.

`dbx` Command

The `dbx` command starts `dbx`.

Native Mode Syntax

`dbx` *options program_name*	Debug *program_name*.
`dbx` *options program_name core*	Debug *program_name* with corefile *core*.
`dbx` *options program_name process_id*	Debug *program_name* with process ID *process_id*.
`dbx` *options* `-` *process_id*	Debug process ID *process_id*; dbx finds the program via `/proc`.
`dbx` *options* `-` *core*	Debug using corefile *core*; see also "`debug` Command" on page 320.
`dbx` *options* `-r` *program_name arguments*	Run *program_name* with arguments *arguments*; if abnormal termination, start debugging *program_name*, else just exit.

where:

program_name is the name of the program to be debugged.

process_id is the process ID of a running process.

arguments are the arguments to be passed to the program.

options are the options listed in "Options" on page 319.

Java Mode Syntax

`dbx` *options program_name*`{.class | .jar}`	Debug *program_name*.
`dbx` *options program_name*`{.class | .jar}` *process_id*	Debug *program_name* with process ID *process_id*.
`dbx` *options* `-` *process_id*	Debug process ID *process_id*; `dbx` finds the program using `/proc`.
`dbx` *options* `-r` *program_name*`{.class | .jar}` *arguments*	Run *program_name* with arguments *arguments*; if abnormal termination, start debugging *program_name*, else just exit.

where:

program_name is the name of the program to be debugged.

process_id is the process ID of a running process.

arguments are the arguments to be passed to the program (not to the JVM software).

options are the options listed in "Options" on page 319.

Options

For both native mode and Java mode, *options* are the following options:

`-c` *commands*	Execute *commands* before prompting for input.
`-C`	Preload the Runtime Checking library (see "`check` Command" on page 304).
`-d`	Used with `-s`, removes *file* after reading.
`-e`	Echo input commands.
`-f`	Force loading of core file, even if it doesn't match.
`-h`	Print the usage help on `dbx`.
`-I` *dir*	Add *dir* to `pathmap` set (see "`pathmap` Command" on page 356).
`-k`	Save and restore keyboard translation state.
`-q`	Suppress messages about reading stabs.
`-r`	Run program; if program exits normally, exit.
`-R`	Print the readme file on `dbx`.
`-s` *file*	Use *file* instead of `/`*`current_directory`*`/.dbxrc` or `$HOME/.dbxrc` as the startup file
`-S`	Suppress reading of initialization file `/`*`installation_directory`*`/lib/dbxrc`.
`-V`	Print the version of `dbx`.
`-w` *n*	Skip *n* frames on `where` command.
`-x exec32`	Suppress using the 64-bit `dbx` binary that runs on systems that support SPARC-V9 binaries. Use the SPARC-V8 32-bit binary instead.
`--`	Marks the end of the option list; use this if the program name starts with a dash.

dbxenv Command

The dbxenv command is used to list or set dbx environment variables. It has identical syntax and identical functionality in native mode and Java mode.

Syntax

`dbxenv`	Display the current settings of the dbx environment variables.
`dbxenv` *environment_variable setting*	Set *environment_variable* to *setting.*

where:

environment_variable is a dbx environment variable.

setting is a valid setting for that variable.

debug Command

The debug command lists or changes the program being debugged. In native mode, it loads the specified application and begins debugging the application. In Java mode, it loads the specified Java application, checks for the existence of the class file, and begins debugging the application.

Native Mode Syntax

`debug`	Print the name and arguments of the program being debugged.
`debug` *program_name*	Begin debugging *program_name* with no process or core.
`debug -c` *core program_name*	Begin debugging *program_name* with core file *core*.
`debug -p` *process_id program_name*	Begin debugging *program_name* with process ID *process_id*.

`debug` *program_name core*	Begin debugging *program* with core file *core*. *program_name* may be `-`. dbx will attempt to extract the name of the executable from the core file. For details, see "Debugging a Core File" on page 50.
`debug` *program_name process_id*	Begin debugging *program_name* with process ID *process_id*. *program_name* may be `-`; `dbx` finds it using `/proc`
`debug -f ...`	Force loading of a core file, even if it doesn't match.
`debug -r ...`	The `-r` option causes `dbx` to retain all `display`, `trace`, `when`, and `stop` commands. With no `-r` option, an implicit `delete all` and `undisplay 0` are performed.
`debug -clone ...`	The `-clone` option causes another `dbx` process to begin execution, permitting debugging of more than one process at a time. Valid only if running under the dbx Debugger.
`debug -clone`	Starts another `dbx` process debugging nothing. Valid only if running under the dbx Debugger.
`debug [`*options*`] --` *program_name*	Start debugging *program*, even if *program_name* begins with a dash.

where:

core is the name of a core file.

options are the options listed in"Options" on page 322.

pid is the process ID of a running process.

program_name is the path name of the program.

Leaks checking and access checking are turned off when a program is loaded with the `debug` command. You can enable them with the `check` command (see "`check` Command" on page 304).

Java Mode Syntax

`debug`	Print the name and arguments of the program being debugged.
`debug` *program_name*`{.class | .jar}`	Begin debugging *program_name* with no process.
`debug -p` *process_id program_name*`{.class | .jar}`	Begin debugging *program_name* with process ID *process_id*.

`debug` *program_name*`{.class | .jar}` *process_id*	Begin debugging *program_name* with process ID *process_id*. *program_name* may be `-`; `dbx` finds it using `/proc`
`debug -r ...`	The `-r` option causes `dbx` to retain all `display`, `trace`, `when`, and `stop` commands. With no `-r` option, an implicit `delete all` and `undisplay 0` are performed.
`debug -clone ...`	The `-clone` option causes another `dbx` process to begin execution, permitting debugging of more than one process at a time. Valid only if running in the dbx Debugger window.
`debug -clone`	Starts another `dbx` process debugging nothing. Valid only if if running in the dbx Debugger window.
`debug [`*options*`] --` *program_name*`{.class | .jar}`	Start debugging *program_name*, even if *program_name* begins with a dash.

where:

file_name is the name of a file.

options are the options listed in"Options" on page 322.

process_id is the process ID of a running process.

program_name is the path name of the program.

Options

`-c` *commands*	Execute *commands* before prompting for input.
`-d`	Used with `-s`, removes *file_name* after reading.
`-e`	Echo input commands.
`-h`	Print the usage help on `dbx`.
`-I` *directory_name*	Add *directory_name* to `pathmap` set (see "`pathmap` Command" on page 356).
`-k`	Save and restore keyboard translation state.
`-q`	Suppress messages about reading stabs.
`-r`	Run program; if program exits normally, exit.
`-R`	Print the readme file on `dbx`.
`-s` *file*	Use *file* instead of `/`*current_directory*`/.dbxrc` or `$HOME/.dbxrc` as the startup file

`-S`	Suppress reading of initialization file `/installation_directory/lib/dbxrc`.
`-V`	Print the version of `dbx`.
`-w` *n*	Skip *n* frames on `where` command.
`--`	Marks the end of the option list; use this if the program name starts with a dash.

`delete` Command

The `delete` command deletes breakpoints and other events. It has identical syntax and identical functionality in native mode and Java mode.

Syntax

`delete [-h]` *handler_id* `...`	Remove `trace` commands, `when` commands, or `stop` commands of given *handler_id*(s). To remove hidden handlers, you must give the `-h` option.
`delete [-h] 0 | all | -all`	Remove all `trace` commands, `when` commands, and `stop` commands excluding permanent and hidden handlers. Specifying `-h` removes hidden handlers as well.
`delete -temp`	Remove all temporary handlers
`delete $firedhandlers`	Delete all the handlers that caused the latest stoppage.

where:

handler_id is the identifier of a handler.

`detach` Command

The `detach` command releases the target process from `dbx`'s control.

Native Mode Syntax

`detach`	Detach `dbx` from the target, and cancel any pending signals.
`detach -sig` *signal*	Detach while forwarding the given *signal*.
`detach -stop`	Detach `dbx` from the target and leave the process in a stopped state. This option allows temporary application of other `/proc`-based debugging tools that might be blocked due to exclusive access. For an example, see "Detaching `dbx` From a Process" on page 95.

where:

signal is the name of a signal.

Java Mode Syntax

`detach`	Detach dbx from the target, and cancel any pending signals.

`dis` Command

The `dis` command disassembles machine instructions. It is valid only in native mode.

Syntax

`dis` *address* `[/` *count*`]`	Disassemble *count* instructions (default is 10), starting at address *address*.
`dis` *address1*`,` *address2*	Disassemble instructions from *address1* through *address2*.
`dis`	Disassemble 10 instructions, starting at the value of + (see "`examine` Command" on page 328).
`dis /`*count*	Disassemble *count* instructions, starting at +.

where:

address is the address at which to start disassembling. The default value of *address* is the address after the last address previously assembled. This value is shared by the `examine` command (see "`examine` Command" on page 328).

address1 is the address at which to start disassembling.

address2 is the address at which to stop disassembling.

count is the number of instructions to disassemble. The default value of *count* is 10.

`display` Command

In native mode, the `display` command evaluates and prints expressions at every stopping point. In Java mode, the `display` command evaluates and prints expressions, local variables, or parameters at every stopping point. Object references are expanded to one level and arrays are printed itemwise.

Native Mode Syntax

`display`	Print the list of expressions being displayed.
`display` *expression*`, ...`	Display the value of expressions *expression*, ... at every stopping point.
`display [-r|+r|-d|+d|-p|+p|-L|-f`*format*`|-F`*format*`|--] `*expression*`, ...$`*newline*	See the "`print` Command" on page 358 for the meaning of these flags.

where:

expression is a valid expression.

format is the output format you want used to print the expression. For information on valid formats, see "`print` Command" on page 358.

Java Mode Syntax

`display`	Print the list of variables and parameters being displayed.
`display` *expression* | *identifier*`,` `...`	Display the value of variables and parameters *identifier, ...* at every stopping point.
`display` `[-r|+r|-d|+d|-p|+p|` `-f`*format*`|-F`*format*`|` `--]` *expression* | *identifier*`,` `...$`*newline*	See the "`print` Command" on page 358 for the meaning of these flags.

where:

class_name is the name of a Java class, using either the package path (using period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`) or the full path name (preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers; for example, `#test1/extra/T1$Inner`). Enclose *class_name* in quotation marks if you use the `$` qualifier.

expression is a valid Java expression.

field_name is the name of a field in the class.

format is the output format you want used to print the expression. For information on valid formats, see "`print` Command" on page 358.

identifier is a local variable or parameter, including `this`, the current class instance variable (*object_name.field_name*) or a class (static) variable (*class_name.field_name*).

object_name is the name of a Java object.

`down` Command

The `down` command moves down the call stack (away from `main`). It has identical syntax and identical functionality in native mode and Java mode.

Syntax

`down`	Move down the call stack one level.
`down` *number*	Move down the call stack *number* levels.
`down -h [`*number*`]`	Move down the call stack, but don't skip hidden frames.

where:

number is a number of call stack levels.

`dump` Command

The `dump` command prints all variables local to a procedure. It has identical syntax and identical functionality in native mode and Java mode.

Syntax

`dump`	Print all variables local to the current procedure.
`dump` *procedure*	Print all variables local to *procedure*.

where:

procedure is the name of a procedure.

`edit` Command

The `edit` command invokes `$EDITOR` on a source file. It is valid only in native mode.

The `edit` command uses `$EDITOR` if dbx is not running under the dbx Debugger Otherwise, it sends a message to the dbx Debugger to display the appropriate file.

Syntax

`edit`	Edit the current file
`edit` *filename*	Edit the specified file *filename*.
`edit` *procedure*	Edit the file containing function or procedure *procedure*.

where:

filename if the name of a file.

procedure is the name of a function or procedure.

`examine` Command

The `examine` command shows memory contents. It is valid only in native mode.

Syntax

`examine [` *address* `] [ / [` *count* `] [` *format* `] ]`	Display the contents of memory starting at *address* for *count* items in format *format*.
`examine` *address1* `,` *address2* `[ / [` *format* `] ]`	Display the contents of memory from *address1* through *address2* inclusive, in format *format*.
`examine` *address* `= [` *format* `]`	Display the address (instead of the contents of the address) in the given format. The *address* may be +, which indicates the address just after the last one previously displayed (the same as omitting it). x is a predefined alias for `examine`.

where:

address is the address at which to start displaying memory contents. The default value of *address* is the address after the address whose contents were last displayed. This value is shared by the `dis` command (see "`dis` Command" on page 324).

address1 is the address at which to start displaying memory contents.

address2 is the address at which to stop displaying memory contents.

count is the number of addresses from which to display memory contents. The default value of *count* is 1.

format is the format in which to display the contents of memory addresses. The default format is X (hexadecimal) for the first `examine` command, and the format specified in the previous `examine` command for subsequent `examine` commands. The following values are valid for *format*:

`d,D`	decimal (2 or 4 bytes)
`o,O`	octal (2 or 4 bytes)
`x,X`	hexadecimal (2 or 4 bytes)
`b`	octal (1 byte)
`c`	character
`w`	wide character
`s`	string
`W`	wide character string
`f`	hex and float (4 bytes, 6 digit prec.)
`F`	hex and float (8 bytes, 14 digit prec.)
`g`	same as `F'`
`E`	hex and float (16 bytes, 14 digit prec.)
`ld,lD`	decimal (4 bytes, same as `D`)
`lo,lO`	octal (4 bytes, same as `O`)
`lx,lX`	hexadecimal (4 bytes, same as `X`)
`Ld,LD`	decimal (8 bytes)
`Lo,LO`	octal (8 bytes)
`Lx,LX`	hexadecimal (8 bytes)

`exception` Command

The exception command prints the value of the current C++ exception. It is valid only in native mode.

Syntax

`exception [-d | +d]`	Prints the value of the current C++ exception, if any.

See the "`print` Command" on page 358 for the meaning of the `-d` flag.

`exists` Command

The `exists` command checks for the existence of a symbol name. It is valid only in native mode.

Syntax

exists *name*	Returns 0 if *name* is found in the current program, 1 if *name* is not found.

where:

name is the name of a symbol.

`file` Command

The `file` command lists or changes the current file. It has identical syntax and identical functionality in native mode and in Java mode.

Syntax

file	Print the name of the current file.
file *filename*	Change the current file.

where:

filename is the name of a file.

`files` Command

In native mode, the `files` command lists file names that match a regular expression. In Java mode, the `files` command lists all of the Java source files known to `dbx`. If your Java source files are not in the same directory as the `.class` or `.jar` files, `dbx` might not find them unless you have set the `CLASSPATH` environment variable (see "Specifying the Location of Your Java Source Files" on page 234).

Native Mode Syntax

files	List the names of all files that contributed debugging information to the current program (those that were compiled with `-g`).
files *regular_expression*	List the names of all files compiled with `-g` that match the given regular expression.

where:

regular_expression is a regular expression.

For example:

```
(dbx) files ^r
myprog:
retregs.cc
reg_sorts.cc
reg_errmsgs.cc
rhosts.cc
```

Java Mode Syntax

`files`	List the names of all of the Java source files known to `dbx`.

`fix` Command

The `fix` command recompiles modified source files and dynamically links the modified functions into the application. It is valid only in native mode. It is not valid on Linux platforms.

Syntax

`fix`	Fix the current file.
`fix` *filename filename* `...`	Fix *filename*.
`fix -f`	Force fixing the file, even if source hasn't been modified.
`fix -a`	Fix all modified files.
`fix -g`	Strip `-O` flags and add `-g` flag.
`fix -c`	Print compilation line (may include some options added internally for use by `dbx`).
`fix -n`	Do not execute compile/link commands (use with `-v`).
`fix -v`	Verbose mode (overrides dbx `fix_verbose` environment variable setting).
`fix +v`	Non-verbose mode (overrides dbx `fix_verbose` environment variable setting).

`fixed` Command

The `fixed` command lists the names of all fixed files. It is valid only in native mode.

Syntax

```
fixed
```

`frame` Command

The `frame` command lists or changes the current stack frame number. It has identical syntax and identical functionality in native mode and in Java mode.

Syntax

`frame`	Display the frame number of the current frame.
`frame [-h]` *number*	Set the current frame to frame *number*.
`frame [-h] +[`*number*`]`	Go *number* frames up the stack; default is 1.
`frame [-h] -[`*number*`]`	Go *number* frames down the stack; default is 1.
`-h`	Go to frame, even if frame is hidden.

where:

number is the number of a frame in the call stack.

`func` Command

In native mode, the `func` command lists or changes the current function. In Java mode, the `func` command lists or changes the current method.

Native Mode Syntax

func	Print the name of the current function.
func *procedure*	Change the current function to the function or procedure *procedure*.

where:

procedure is the name of a function or procedure.

Java Mode Syntax

func	Print the name of the current method.
func [*class_name*.]*method_name* *[(parameters)]*	Change the current function to the method *method_name*.

where:

class_name is the name of a Java class, using either the package path (using period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`) or the full path name (preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers; for example, `#test1/extra/T1$Inner`). Enclose *class_name* in quotation marks if you use the `$` qualifier.

method_name is the name of a Java method.

parameters are the method's parameters.

`funcs` Command

The `funcs` command lists all function names that match a regular expression. It is valid only in native mode.

Syntax

`funcs`	List all functions in the current program.
`funcs [-f` *filename*`] [-g]` [*regular_expression*]	If `-f` *filename* is specified, list all functions in the file. If `-g` is specified, list all functions with debugging information. If *regular_expression* is specified, list all functions that match the regular expression.

where:

filename is the name of the file for which you wish to list all the functions.

regular_expression is the regular expression for which you wish to list all the matching functions.

For example:

```
(dbx) funcs [vs]print
'libc.so.1'isprint
'libc.so.1'wsprintf
'libc.so.1'sprintf
'libc.so.1'vprintf
'libc.so.1'vsprintf
```

gdb Command

The gdb command supports the gdb command set. It is valid only in native mode.

Syntax

`gdb on | off`	Use `gdb on` to enter the gdb command mode under which `dbx` understands and accepts gdb commands. To exit the gdb command mode and return to the dbx command mode, enter `gdb off`. `dbx` commands are not accepted while in gdb command mode and vice versa. All debugging settings such as breakpoints are preserved across different command modes.

The following gdb commands are not supported in this release:

- commands
- define
- handle
- hbreak
- interrupt
- maintenance
- printf
- rbreak
- return
- signal

- `tcatch`
- `until`

`handler` Command

The `handler` command modifies event handlers (enable, disable, etc.). It has identical syntax and identical functionality in native mode and in Java mode.

A handler is created for each event that needs to be managed in a debugging session. The commands `trace`, `stop`, and `when` create handlers. Each of these commands returns a number known as the handler ID (*handler_id*). The `handler`, `status`, and `delete` commands manipulate or provide information about handlers in a generic fashion.

Syntax

`handler -enable` *handler_id* `...`	Enable given handlers, specify *handler_id* as all for all handlers.
`handler -disable` *handler_id* `...`	Disable given handlers, specify *handler_id* as all for all handlers. Use `$firedhandlers` instead of *handler_id* to disable the handlers that caused the most recent stoppage.
`handler -count` *handler_id*	Print value of trip counter for given handler.
`handler -count` *handler_id newlimit*	Set new count limit for given event.
`handler -reset` *handler_id*	Reset trip counter for given handler.

where:

handler_id is the identifier of a handler.

`hide` Command

The `hide` command hides stack frames that match a regular expression. It is valid only in native mode.

Syntax

`hide`	List the stack frame filters currently in effect.
`hide` *regular_expression*	Hide stack frames matching *regular_expression*. The regular expression matches either the function name, or the name of the loadobject, and is a sh or ksh file matching style regular expression.

where:

regular_expression is a regular expression.

`ignore` Command

The `ignore` command tells the dbx process not to catch the given signal(s). It is valid only in native mode.

Ignoring a signal causes dbx not to stop when the process receives that kind of signal.

Syntax

`ignore`	Print a list of the ignored signals.
`ignore` *number*`...`	Ignore signal(s) numbered *number*.
`ignore` *signal*`...`	Ignore signal(s) named by *signal*. SIGKILL cannot be caught or ignored.
`ignore $(catch)`	Ignore all signals.

where:

number is the number of a signal.

signal is the name of a signal.

`import` Command

The `import` command imports commands from a `dbx` command library. It has identical syntax and identical functionality in native mode and in Java mode.

Syntax

`import` *pathname*	Import commands from the `dbx` command library *pathname*.

where:

pathname is the pathname of a `dbx` command library.

`intercept` Command

The `intercept` command throws (C++ exceptions) of the given type (C++ only). It is valid only in native mode.

A thrown exception for which there is no matching catch is called an "unhandled" throw. A thrown exception that doesn't match the exception specification of the function it is thrown from is called an "unexpected" throw.

Unhandled and unexpected throws are intercepted by default.

Syntax

`intercept` *typename*	Intercept throws of type *typename*.
`intercept -a`	Intercept all throws.
`intercept -x` *typename*	Do not intercept *typename*.
`intercept -a -x` *typename*	Intercept all types except *typename*.
`intercept`	List intercepted types.

where:

typename may be either -unhandled or -unexpected.

`java` Command

The `java` command is used when `dbx` is in JNI mode to indicate that the Java version of a specified command is to be executed. It causes the specified command to use the Java expression evaluator, and when relevant, to display Java threads and stack frames.

Syntax

java *command*

where:

command is the command name and arguments of the command to be executed.

`jclasses` Command

The `jclasses` command prints the names of all Java classes known to `dbx` when you give the command. It is valid only in Java mode.

Classes in your program that have not yet been loaded are not printed.

Syntax

`jjclasses`	Print the names of all Java classes known to `dbx`
`jclasses -a`	Print system classes as well as other known Java classes.

joff Command

The joff command switches dbx from Java mode or JNI mode to native mode.

Syntax

```
joff
```

jon Command

The jon command switches dbx from native mode to Java mode.

Syntax

```
jon
```

jpkgs Command

The jpkgs command prints the names of all Java packages known to dbx when you give the command. It is valid only in Java mode.

Packages in your program that have not yet been loaded are not printed.

Syntax

```
jpkgs
```

`kill` Command

The `kill` command sends a signal to a process and kills the target process. It is valid only in native mode.

Syntax

`kill -l`	List all known signal numbers, names, and descriptions.
`kill`	Kill the controlled process.
`kill` *job*`...`	Send the SIGTERM signal to the listed jobs.
`kill` *–signal job*`...`	Send the given signal to the listed jobs.

where:

job may be a process ID or may be specified in any of the following ways:

%+	Kill the current job.
%-	Kill the previous job.
%*number*	Kill job number *number*.
%*string*	Kill the job which begins with *string*.
%?*string*	Kill the job which contains *string*.

signal is the name of a signal.

`language` Command

The `language` command lists or changes the current source language. It is valid only in native mode.

Syntax

`language`	Print the current language mode set by the `dbx` `language_mode` environment variable (see "Setting dbx Environment Variables" on page 66). If the language mode is set to `autodetect` or `main`, the command also prints the name of the current language used for parsing and evaluating expressions.

where:

language is c, c++, fortran, or fortran90.

Note – c is an alias for ansic.

`line` Command

The `line` command lists or change the current line number. It has identical syntax and identical functionality in native mode and in Java mode.

Syntax

`line`	Display the current line number.
`line` *number*	Set the current line number to `number`.
`line` `"`*filename*`"`	Set current line number to line 1 in *filename*.
`line` `"`*filename*`"`:*number*	Set current line number to line *number* in *filename*.

where:

filename is the name of the file in which to change the line number. The "" around the filename is optional.

number is the number of a line in the file.

Examples

```
line 100
line "/root/test/test.cc":100
```

`list` Command

The `list` command displays lines of a source file. It has identical syntax and identical functionality in native mode and in Java mode.

The default number of lines listed, N, is controlled by the `dbx output_list_size` environment variable.

Syntax

`list`	List N lines.
`list` *number*	List line number *number*.
`list +`	List next N lines.
`list +`*n*	List next *n* lines.
`list -`	List previous N lines.
`list -`*n*	List previous *n* lines.
`list` *n1*`,`*n2*	List lines from *n1* to *n2*.
`list` *n1*`,+`	List from *n1* to *n1* + N.
`list` *n1*`,+`*n2*	List from *n1* to *n1* + *n2*.
`list` *n1*`,-`	List from *n1*-N to *n1*.
`list` *n1*`,-`*n2*	List from *n1*-*n2* to *n1*
`list` *function*	List the start of the source for *function*. `list` *function* changes the current scope. See "Program Scope" on page 76 for more information.
`list` *filename*	List the start of the file *filename*.
`list` *filename*`:`*n*	List file *filename* from line *n*. Where appropriate, the line number may be '$' which denotes the last line of the file. Comma is optional.

where:

filename is the file name of a source code file.

function is the name of a function to display.

number is the number of a line in the source file.

n is a number of lines to display.

n1 is the number of the first line to display.

n2 is the number of the last line to display.

Options

`-i` or `-instr`	Intermix source lines and assembly code
`-w` or `-w`*n*	List N (or *n*) lines (window) around line or function. This option is not allowed in combination with the '+' or '-' syntax or when two line numbers are specified.

Examples

```
list                        // list N lines starting at current line
list +5                    // list next 5 lines starting at current line
list -                      // list previous N lines
list -20                    // list previous 20 lines
list 1000                   // list line 1000
list 1000,$                 // list from line 1000 to last line
list 2737 +24               // list line 2737 and next 24 lines
list 1000 -20               // list line 980 to 1000
list test.cc:33             // list source line 33 in file test.cc
list -w                     // list N lines around current line
list -w8 `test.cc`func1     // list 8 lines around function func1
list -i 500 +10            // list source and assembly code for line
                               500 to line 510
```

`listi` Command

The `listi` command displays source and disassembled instructions. It is valid only in native mode.

See "`list` Command" on page 343 for details.

`loadobject` Command

The `loadobject` command lists and manages symbolic information from loadobjects. It is valid only in native mode.

Syntax

`loadobject` *command_list*

`-list [` *regexp* `] [ -a ]`	Show currently loaded loadobjects (see "`loadobject -list` Command" on page 347)
`-load` *loadobject*	Load symbols for specified loadobject (see "`loadobject -load` Command" on page 348)
`-unload [` *regexp* `]`	Unload specified loadobjects (see "`loadobject -unload` Command" on page 349)
`-hide [` *regexp* `]`	Remove loadobject from dbx's search algorithm (see "`loadobject -hide` Command" on page 347)
`-use [` *regexp* `]`	Add loadobject to dbx's search algorithm (see "`loadobject -use` Command" on page 349)
`-dumpelf [` *regexp* `]`	Show various ELF details of the loadobject (see "`loadobject -dumpelf` Command" on page 346)
`-exclude` *ex-regexp*	Don't automatically load loadobjects matching *ex-regexp* (see "`loadobject -exclude` Command" on page 346)
`-exclude`	Show list of exclude patterns (see "`loadobject -exclude` Command" on page 346)
`-exclude -clear`	Clear the 'exclude' list of patterns (see "`loadobject -exclude` Command" on page 346)

where:

regexp is a regular expression. If it is not specified the command applies to all loadobjects.

ex-regexp is not optional, it must be specified.

This command has a default alias `lo`.

`loadobject -dumpelf` Command

The `loadobject -dumpelf` command unloads specified loadobjects. It is valid only in native mode.

Syntax

```
loadobject -dumpelf [ regexp ]
```

where:

regexp is a regular expression. If it is not specified the command applies to all loadobjects.

This command dumps out information related to the ELF structure of the loadobject file on disk. The details of this output are highly subject to change. If you want to parse this output, use the Solaris operating environment commands `dump` or `elfdump`.

`loadobject -exclude` Command

The `loadobject -exclude` command tells `dbx` not to automatically load loadobjects matching the specified regular expression.

Syntax

```
loadobject -exclude ex-regexp [ -clear ]
```

where:

ex-regexp is a regular expression.

This command prevents dbx from automatically loading symbols for loadobjects that match the specified regular expression. Unlike *regexp* in other loadobject subcommands, if *ex- regexp* is not specified, it does not default to all. If you do not specify *ex- regexp*, the command lists the excluded patterns that have been specified by previous loadobject -exclude commands.

If you specify -clear, the list of excluded patterns is deleted.

Currently this functionality cannot be used to prevent loading of the main program, or the runtime linker. Also, using it to prevent loading of C++ runtime libraries could cause C++ functionality.

This option should not be used with runtime checking (RTC).

loadobject -hide Command

The loadobject -hide command removes loadobjects from dbx's search algorithm.

Syntax

```
loadobject -hide [ regexp ]
```

where:

regexp is a regular expression. If it is not specified the command applies to all loadobjects.

This command removes a loadobject from the program scope, and hides its functions and symbols from dbx. This command also resets the 'preload' bit

loadobject -list Command

The loadobject -list command shows currently loaded loadobjects. It is valid only in native mode.

Syntax

```
loadobject -list [ regexp ] [ -a]
```

where:

regexp is a regular expression. If it is not specified the command applies to all loadobjects.

The full path name for each loadobject is shown along with letters in the margin to show status. Loadobjects that are hidden are listed only if you specify the `-a` option.

h	This means “hidden” (the symbols are not found by symbolic queries like `whatis` or `stop in`).
u	If there is an active process, u means “unmapped”.
p	This letter indicates an LO that is preloaded, that is, the result of a ‘`loadobject -load`’ command or a `dlopen` event in the program. (See ‘help loadobject preloading’)

For example:

```
(dbx) lo -list libm
/usr/lib/64/libm.so.1
/usr/lib/64/libmp.so.2
(dbx) lo -list ld.so
h  /usr/lib/sparcv9/ld.so.1 (rtld)
```

This last example shows that the symbols for the runtime linker are hidden by default. To use those symbols in dbx commands, see the ‘lo -use’ command below.

`loadobject -load` Command

The `loadobject -load` command loads symbols for specified loadobjects. It is valid only in native mode.

Syntax

```
loadobject -load loadobject ...
```

where:

loadobject can be a full path name or a library in `/usr/lib` or `/usr/lib/sparcv9`. If there is a program being debugged, then only the proper ABI library directory will be searched.

`loadobject -unload` Command

The `loadobject -unload` command unloads specified loadobjects. It is valid only in native mode.

Syntax

```
loadobject -unload [ regexp ]
```

where:

regexp is a regular expression. If it is not specified the command applies to all loadobjects.

This command unloads the symbols for any loadobjects matching the *regexp* supplied on the command line. The main program loaded with the `debug` command cannot be unloaded. `dbx` may also refuse to unload other loadobjects that might be currently in use, or critical to the proper functioning of `dbx`.

`loadobject -use` Command

The `loadobject -use` command adds loadobjects from dbx's search algorithm. It is valid only in native mode.

Syntax

```
loadobject -use [ regexp ]
```

where:

regexp is a regular expression. If it is not specified the command applies to all loadobjects.

`lwp` Command

The `lwp` command lists or changes the current LWP (lightweight process). It is valid only in native mode.

Note – The `lwp` command is available only on Solaris platforms.

Syntax

`lwp`	Display current LWP.
`lwp` *lwp_id*	Switch to LWP *lwp_id*.
`lwp -info`	Displays the name, home, and masked signals of the current lwp.

where:

lwp_id is the identifier of a lightweight process.

`lwps` Command

The `lwps` command lists all LWPs (lightweight processes) in the process. It is valid only in native mode.

Note – The `lwps` command is available only on Solaris platforms.

Syntax

`lwps`	List all LWPs in the current process

`mmapfile` Command

The `mmapfile` command views the contents of memory mapped files that are missing from a core dump. It is valid only in native mode.

Solaris core files do not contain any memory segments that are read-only. Executable read-only segments (that is, text) are dealt with automatically and `dbx` resolves memory accesses against these by looking into the executable and the relevant shared objects.

Syntax

`mmapfile` *mmapped_file address offset length*	View contents of memory mapped files missing from core dump.

where:

mmapped_file is the file name of a file that was memory mapped during a core dump.

address is the starting address of the address space of the process.

length is length in bytes of the address space to be viewed.

offset is the offset in bytes to the starting address in *mmapped_file*.

`module` Command

The `module` command reads debugging information for one or more modules. It is valid only in native mode.

Syntax

`module [-v]`	Print the name of the current module.
`module [-f] [-v] [-q]` *name*	Read in debugging information for the module called *name*.
`module [-f] [-v] [-q] -a`	Read in debugging information for all modules.

where:

name is the name of a module for which to read debugging information.

`-a` specifies all modules.

`-f` forces reading of debugging information, even if the file is newer than the executable (use with caution!).

`-v` specifies verbose mode, which prints language, file names, etc.

`-q` specifies quiet mode.

Example

Read-only data segments typically occur when an application memory maps a database. For example:

```
caddr_t vaddr = NULL;
off_t offset = 0;
size_t = 10 * 1024;
int fd;
fd = open("../DATABASE", ...)
vaddr = mmap(vaddr, size, PROT_READ, MAP_SHARED, fd, offset);
index = (DBIndex *) vaddr;
```

To be able to access the database through the debugger as memory you would type:

```
mmapfile ../DATABASE $[vaddr] $[offset] $[size]
```

Then you could look at your database contents in a structured way by typing:

```
print *index
```

`modules` Command

The `modules` command lists module names. It is valid only in native mode.

Syntax

`modules [-v]`	List all modules
`modules [-v] -debug`	List all modules containing debugging information.
`modules [-v] -read`	List names of modules containing debugging information that have been read in already.

where:

`-v` specifies verbose mode, which prints language, file names, etc.

`native` Command

The `native` command is used when `dbx` is in Java mode to indicate that the native version of a specified command is to be executed. Preceding a command with "native" results in `dbx` executing the command in native mode. This means that expressions are interpreted and displayed as C expressions or C++ expressions, and certain other commands produce different output than they do in Java mode.

This command is useful when you are debugging Java code but you want to examine the native environment.

Syntax

`native` *command*

where:

command is the command name and arguments of the command to be executed.

`next` Command

The `next` command steps one source line (stepping over calls).

The `dbx` `step_events` environment variable (see "Setting `dbx` Environment Variables" on page 66) controls whether breakpoints are enabled during a step.

Native Mode Syntax

`next`	Step one line (step over calls). With multithreaded programs when a function call is stepped over, all LWPs (lightweight processes) are implicitly resumed for the duration of that function call in order to avoid deadlock. Non-active threads cannot be stepped.
`next` *n*	Step *n* lines (step over calls).
`next ... -sig` *signal*	Deliver the given signal while stepping.
`next ...` *thread_id*	Step the given thread.
`next ...` *lwp_id*	Step the given LWP. Will not implicitly resume all LWPs when stepping over a function.

where:

n is the number of lines to step.

signal is the name of a signal.

thread_id is a thread ID.

lwp_id is an LWP ID.

When an explicit *thread_id* or *lwp_id* is given, the deadlock avoidance measure of the generic `next` command is defeated.

See also "`nexti` Command" on page 355 for machine-level stepping over calls.

Note – For information on lightweight processes (LWPs), see the Solaris *Multithreaded Programming Guide.*

Java Mode Syntax

`next`	Step one line (step over calls). With multithreaded programs when a function call is stepped over, all LWPs (lightweight processes) are implicitly resumed for the duration of that function call in order to avoid deadlock. Non-active threads cannot be stepped.
`next` *n*	Step *n* lines (step over calls).
`next ...` *thread_id*	Step the given thread.
`next ...` *lwpid*	Step the given LWP. Will not implicitly resume all LWPs when stepping over a function.

where:

n is the number of lines to step.

thread_id is a thread identifier.

lwpid is an LWP identifier.

When an explicit *thread_id* or *lwpid* is given, the deadlock avoidance measure of the generic `next` command is defeated.

Note – For information on lightweight processes (LWPs), see the Solaris *Multithreaded Programming Guide.threaded Programming Guide*.

`nexti` Command

The `nexti` command steps one machine instruction (stepping over calls). It is valid only in native mode.

Syntax

`nexti`	Step one machine instruction (step over calls).
`nexti` *n*	Step *n* machine instructions (step over calls).
`nexti -sig` *signal*	Deliver the given signal while stepping.
`nexti ...` *lwp_id*	Step the given LWP.
`nexti ...` *thread_id*	Step the LWP on which the given thread is active. Will not implicitly resume all LWPs when stepping over a function.

where:

n is the number of instructions to step.

signal is the name of a signal.

thread_id is a thread ID.

lwp_id is an LWP ID.

`pathmap` Command

The `pathmap` command maps one path name to another for finding source files, etc. The mapping is applied to source paths, object file paths and the current working directory (if you specify -c). The `pathmap` command has identical syntax and identical functionality in native mode and in Java mode.

The `pathmap` command is useful for dealing with automounted and explicit NFS mounted filesystems with different paths on differing hosts. Specify `-c` when you are trying to correct problems arising due to the automounter since CWD's are inaccurate on automounted filesystems as well. The `pathmap` command is also useful if source or build trees are moved.

`pathmap /tmp_mnt /` exists by default.

The `pathmap` command is used to find load objects for core files when the `dbx` environment variable `core_lo_pathmap` is set to on. Other than this case, the `pathmap` command has no effect on finding load objects (shared libraries). See "Debugging a Mismatched Core File" on page 51.

Syntax

`pathmap [ -c ] [-`*index*`] `*from*` `*to*	Establish a new mapping from *from* to *to*.
`pathmap [ -c ] [-`*index*`] to`	Map all paths to *to*.
`pathmap`	List all existing path mappings (by index)
`pathmap -s`	The same, but the output can be read by `dbx`.
`pathmap -d` *from1* *from2*`...`	Delete the given mapping(s) by path.
`pathmap -d` *index1* *index2* `...`	Delete the given mapping(s) by index.

where:

from and *to* are filepath prefixes. *from* refers to the filepath compiled into the executable or objectfile and *to* refers to the filepath at debug time.

from1 is filepath of the first mapping to be deleted.

from2 is filepath of the last mapping to be deleted.

index specifies the index with which the mapping is to be inserted in the list. If you do not specify an index, the mapping is added to the end of the list.

index1 is the index of the first mapping to be deleted.

index2 is the index of the last mapping to be deleted.

If you specify `-c`, the mapping is applied to the current working directory as well.

If you specify `-s`, the existing mappings are listed in an output format that `dbx` can read.

If you specify `-d`, the specified mappings are deleted.

Examples

```
(dbx) pathmap /export/home/work1 /net/mmm/export/home/work2
  # maps /export/home/work1/abc/test.c to
/net/mmm/export/home/work2/abc/test.c
(dbx) pathmap /export/home/newproject
  # maps /export/home/work1/abc/test.c to
/export/home/newproject/test.c
(dbx) pathmap
(1) -c /tmp_mnt /
(2) /export/home/work1 /net/mmm/export/home/work2
(3) /export/home/newproject
```

`pop` Command

The `pop` command removes one or more frames from the call stack. It is valid only in native mode.

You can pop only to a frame for a function that was compiled with `-g`. The program counter is reset to the beginning of the source line at the callsite. You cannot pop past a function call made by the debugger; use `pop -c`.

Normally a `pop` command calls all the C++ destructors associated with the popped frames; you can override this behavior by setting the `dbx pop_auto_destruct` environment variable to off (see "Setting dbx Environment Variables" on page 66).

Syntax

`pop`	Pop current top frame from stack
`pop` *number*	Pop *number* frames from stack
`pop -f` *number*	Pop frames from stack until specified frame *number*
`pop -c`	Pop the last call made from the debugger.

where:

number is the number of frames to pop from the stack.

`print` Command

In native mode, the `print` command prints the value of an expression. In Java mode, the `print` command prints the value of an expression, local variable, or parameter.

Native Mode Syntax

`print` *expression*`, ...`	Print the value of the expression(s) *expression*, ...
`print -r` *expression*	Print the value of the expression *expression* including its inherited members (C++ only).
`print +r` *expression*	Don't print inherited members when the `dbx output_inherited_members` environment variable is on (C++ only).
`print -d [-r]` *expression*	Show dynamic type of expression *expression* instead of static type (C++ only).
`print +d [-r]` *expression*	Don't use dynamic type of expression *expression* when the `dbx output_dynamic_type` environment variable is on (C++ only).
`print -p` *expression*	Call the `prettyprint` Function.

`print +p` *expression*	Do not call the `prettyprint` Function when the `dbx output_pretty_print` environment variable is on.
`print -L` *expression*	If the printing object *expression* is larger than 4K, enforce the printing.
`print -l` *expression*	('Literal') Do not print the left side. If the expression is a string (`char *`), do not print the address, just print the raw characters of the string, without quotes.
`print -f`*format expression*	Use *format* as the format for integers, strings, or floating-point expressions.
`print -F`*format expression*	Use the given format but do not print the left hand side (the variable name or expression)).
`print -o` *expression*	Print the value of *expression*, which must be an enumeration as an ordinal value. You may also use a format string here (`-f`*format*). This option is ignored for non-enumeration expressions.
`print --` *expression*	'--' signals the end of flag arguments. This is useful if *expression* may start with a plus or minus (see "Program Scope" on page 76 for scope resolution rules.

where:

expression is the expression whose value you want to print.

format is the output format you want used to print the expression. If the format does not apply to the given type, the format string is silently ignored and `dbx` uses its built-in printing mechanism.

The allowed formats are a subset of those used by the `printf`(3S) command. The following restrictions apply:

- No `n` conversion.
- No `*` for field width or precision.
- No `%<digits>$` argument selection.

- Only one conversion specification per format string.

The allowed forms are defined by the following simple grammar:

```
FORMAT::=  CHARS % FLAGS WIDTH PREC MOD SPEC CHARS
CHARS::=  <any character sequence not containing a %>
|     %%
|     <empty>
|     CHARS CHARS
FLAGS::=  + | - | <space> | # | 0 | <empty>
WIDTH::=  <decimal_number> | <empty>
PREC::=  . | . <decimal_number> | <empty>
MOD::=  h | l | L | ll | <empty>
SPEC::=  d | i | o | u | x | X | f | e | E | g | G |
                 c | wc | s | ws | p
```

If the given format string does not contain a `%`, `dbx` automatically prepends one. If the format string contains spaces, semicolons, or tabs, the entire format string must be surrounded by double quotes.

Java Mode Syntax

`print` *expression*`, ... |` *identifier*`, ...`	Print the value(s) of the expression(s) *expression*, `...` or identifier(s) *identifier*, `....`
`print -r` *expression* `|` *identifier*	Print the value of *expression* or *identifier* including its inherited members.
`print +r` *expression* `|` *identifier*	Don't print inherited members when the `dbx` `output_inherited_members` environment variable is on.
`print -d [-r]` *expression* `|` *identifier*	Show dynamic type of *expression* or *identifier* instead of static type.
`print +d [-r]` *expression* `|` *identifier*	Don't use dynamic type of *expression* or *identifier* when the `dbx` `output_dynamic_type` environment variable is on.
`print --` *expression* `|` *identifier*	'--' signals the end of flag arguments. This is useful if *expression* may start with a plus or minus (see "Scope" in *Debugging a Program With dbx* for scope resolution rules.

where:

class_name is the name of a Java class, using either the package path (using period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`) or the full path name (preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers; for example, `#test1/extra/T1$Inner`). Enclose *class_name* in quotation marks if you use the `$` qualifier.

expression is the Java expression whose value you want to print.

field_name is the name of a field in the class.

identifier is a local variable or parameter, including `this`, the current class instance variable (*object_name.field_name*) or a class (static) variable (*class_name.field_name*).

object_name is the name of a Java object.

`proc` Command

The `proc` command displays the status of the current process. It has identical syntax and identical functionality in native mode and in Java mode.

Syntax

`proc -map`	Show the list of loadobjects with addresses
`proc -pid`	Show current process ID (pid)

`prog` Command

The `prog` command manages programs being debugged and their attributes. It has identical syntax and identical functionality in native mode and Java mode.

Syntax

`prog -readsyms`	Read symbolic information which was postponed by having set the `dbx run_quick` environment variable to on.
`prog -executable`	Prints the full path of the executable, `-` if the program was attached to using `-`.
`prog -argv`	Prints the whole `argv`, including `argv[0]`.
`prog -args`	Prints the `argv`, excluding `argv[0]`.
`prog -stdin`	Prints < *filename* or empty if `stdin` is used.
`prog -stdout`	Prints > *filename* or >> *filename* or empty if `stdout` is used. The outputs of `-args`, `-stdin`, `-stdout` are designed so that the strings can be combined and reused with the `run` command (see "`run` Command" on page 366).

`quit` Command

The `quit` command exits `dbx`. It has identical syntax and identical functionality in native mode and Java mode.

If `dbx` is attached to a process, the process is detached from before exiting. If there are pending signals, they are cancelled. Use the `detach` command (see "`detach` Command" on page 323) for fine control.

Syntax

`quit`	Exit `dbx` with return code 0. Same as *exit*.
`quit` *n*	Exit with return code *n*. Same as `exit` *n*.

where:

n is a return code.

`regs` Command

The `regs` command prints the current value of registers. It is valid only in native mode.

Syntax

```
regs [-f] [-F]
```

where:

- `-f` includes floating-point registers (single precision) (SPARC platform only)
- `-F` includes floating-point registers (double precision) (SPARC platform only)

Example (SPARC platform)

```
dbx[13] regs -F
current thread: t@1
current frame:  [1]
g0-g3             0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7             0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3             0x00000003 0x00000014 0xef7562b4 0xefffff420
o4-o7             0xef752f80 0x00000003 0xefffff3d8 0x000109b8
l0-l3             0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7             0xefffff438 0x00000001 0x00000007 0xef74df54
i0-i3             0x00000001 0xefffff4a4 0xefffff4ac 0x00020c00
i4-i7             0x00000001 0x00000000 0xefffff440 0x000108c4
y                 0x00000000
psr               0x40400086
pc                0x000109c0:main+0x4mov     0x5, %l0
npc               0x000109c4:main+0x8st      %l0, [%fp - 0x8]
f0f1              +0.00000000000000e+00
f2f3              +0.00000000000000e+00
f4f5              +0.00000000000000e+00
f6f7              +0.00000000000000e+00
```

`replay` Command

The `replay` command replays debugging commands since the last `run`, `rerun`, or `debug` command. It is valid only in native mode.

Syntax

`replay [-`*number*`]`	Replay all or all minus *number* commands since last `run` command, `rerun` command, or `debug` command.

where:

number is the number of commands not to replay.

`rerun` Command

The `rerun` command runs the program with no arguments. It has identical syntax and identical functionality in native mode and Java mode.

Syntax

rerun	Begin executing the program with no arguments
rerun *arguments*	Begin executing the program with new arguments by the `save` command (see "`save` Command" on page 368).

`restore` Command

The `restore` command restores dbx to a previously saved state. It is valid only in native mode.

Syntax

restore [*filename*]	Restore `dbx` to the state it was in when it was saved.

where:

filename is the name of the file to which the `dbx` commands executed since the last `run`, `rerun`, or `debug` command were saved.

`rprint` Command

The `rprint` command prints an expression using shell quoting rules. It is valid only in native mode.

Syntax

`rprint [-r|+r|-d|+d|-p|+p|-L|-l|-f`*format*`|-F`*format*`|--]` *expression*	Print the value of the expression. No special quoting rules apply, so `rprint a > b` puts the value of `a` (if it exists) into file `b` (see "`print` Command" on page 358 for the meanings of the flags).

where:

expression is the expression whose value you want to print.

format is the output format you want used to print the expression. For information on valid formats, see "`print` Command" on page 358.

`rtc -showmap` Command

The `rtc -showmap` command reports the address range of program text categorized by instrumentation type. It is valid only in native mode.

Note – The `rtc -showmap` command is available only on Solaris platforms.

Syntax

`showmap`	Show address range of branches and traps (RTC)

This command is intended for expert users, and internal debugging of `dbx`. Runtime checking instruments program text for access checking. The instrumentation type can be a branch or a trap instruction based on available resources. The `rtc -showmap` command reports the address range of program text categorized by instrumentation type. This map can be used to find an optimal location for adding patch area object files, and to avoid the automatic use of traps. See "Runtime Checking's 8 Megabyte Limit" on page 164 for details.

`run` Command

The `run` command runs the program with arguments.

Use Control-C to stop executing the program.

Native Mode Syntax

`run`	Begin executing the program with the current arguments
`run` *arguments*	Begin executing the program with new arguments.
run ... >|>> *input_file*	Set the output redirection.
run ... < *output_file*	Set the input redirection.

where:

arguments are the arguments to be used in running the target process.

input_file is the file name of the file from which input is to be redirected.

output_file is the file name of the file to which output is to be redirected.

Note – There is currently no way to redirect `stderr` using the `run` or `runargs` command.

Java Mode Syntax

`run`	Begin executing the program with the current arguments
`run` *arguments*	Begin executing the program with new arguments.

where:

arguments are the arguments to be used in running the target process. They are passed to the Java application, not to the JVM software. Do not include the main class name as an argument.

You cannot redirect the input or output of a Java application with the `run` command.

Breakpoints you set in one run persist in subsequent runs.

`runargs` Command

The `runargs` command changes the arguments of the target process. It has identical syntax and identical functionality in native mode and Java mode.

Use the `debug` command (see "`debug` Command" on page 320) with no arguments to inspect the current arguments of the target process.

Syntax

`runargs` *arguments*	Set the current arguments, to be used by the run command (see "`run` Command" on page 366).
`runargs ... >|>>` *file*	Set the output redirection to be used by the `run` command.
`runargs ... <` *file*	Set the input redirection to be used by the `run` command.
`runargs`	Clear the current arguments.

where:

arguments are the arguments to be used in running the target process.

file is the file to which output from the target process or input to the target process is to be redirected.

`save` Command

The `save` command saves commands to a file. It is valid only in native mode.

Syntax

`save [-`*number*`] [`*filename*`]`	Save all or all minus *number* commands since last `run` command, `rerun` command, or `debug` command to the default file or *filename*.

where:

number is the number of commands not to save.

filename is the name of the file to save the `dbx` commands executed since the last `run`, `rerun`, or `debug` command.

`scopes` Command

The `scopes` command prints a list of active scopes. It is valid only in native mode.

Syntax

```
scopes
```

`search` Command

The `search` command searches forward in the current source file. It is valid only in native mode

Syntax

`search` *string*	Search forward for *string* in the current file.
search	Repeat search, using last search string

where:

string is the character string for which you wish to search.

`showblock` Command

The `showblock` command shows where the particular heap block was allocated from runtime checking. It is valid only in native mode.

Note – The `showblock` command is available only on Solaris platforms.

When memory use checking or memory leak checking is turned on, the `showblock` command shows the details about the heap block at the specified address. The details include the location of the blocks' allocation and its size. See "`check` Command" on page 304.

Syntax

```
showblock -a address
```

where:

address is the address of a heap block.

`showleaks` Command

The `showleaks` command reports new memory leaks since last `showleaks` command. It is valid only in native mode.

Note – The `showleaks` command is available only on Solaris platforms.

In the default non-verbose case, a one line report per leak record is printed. Actual leaks are reported followed by the possible leaks. Reports are sorted according to the combined size of the leaks.

Syntax

```
showleaks [-a] [-m m] [-n number] [-v]
```

where:

`-a` shows all the leaks generated so far (not just the leaks since the last `showleaks` command).

`-m` *m* combines leaks; if the call stack at the time of allocation for two or more leaks matches *m* frames, then these leaks are reported in a single combined leak report. If the `-m` option is given, it overrides the global value of *m* specified with the `check` command (see "`check` Command" on page 304)..

`-n` *number* shows up to *number* records in the report. The default is to show all records.

-v Generate verbose output. The default is to show non-verbose output.

`showmemuse` Command

The `showmemuse` command shows memory used since last `showmemuse` command. It is valid only in native mode.

Note – The `showmemuse` command is available only on Solaris platforms.

A one line report per "block in use" record is printed. The commands sorts the reports according to the combined size of the blocks. Any leaked blocks since the last `showleaks` command (see "`showleaks` Command" on page 369) are also included in the report.

Syntax

```
showmemuse [-a] [-m <m>] [-n number] [-v]
```

where:

`-a` shows all the blocks in use (not just the blocks since the last showmemuse command).

`-m` *m* combines the blocks in use reports. The default value of *m* is 2 or the global value last given with the `check` command (see "`check` Command" on page 304). If the call stack at the time of allocation for two or more blocks matches *m* frames then these blocks are reported in a single combined report. If the `-m` option is given, it overrides the global value of *m*.

`-n` *number* shows up to *number* records in the report. The default is 20.`-v` generates verbose output. The default is to show non-verbose output.

`source` Command

The source command executes commands from a given file. It is valid only in native mode.

Syntax

`source` *filename*	Execute commands from file *filename*. $PATH is not searched.

`status` Command

The `status` command lists event handlers (breakpoints, etc.). It has identical syntax and identical functionality in native mode and Java mode.

Syntax

status	Print trace, when, and stop breakpoints in effect.
`status` *handler_id*	Print status for handler *handler_id*.
`status -h`	Print trace, when, and stop breakpoints in effect including # the hidden ones.
`status -s`	The same, but the output can be read by `dbx`.

where:

handler_id is the identifier of an event handler.

Example

```
(dbx) status -s > bpts
...
(dbx) source bpts
```

`step` Command

The `step` command steps one source line or statement (stepping into calls).

The `dbx` `step_events` environment variable controls whether breakpoints are enabled during a step.

Native Mode Syntax

`step`	Single step one line (step into calls). With multithreaded programs when a function call is stepped over, all LWPs (lightweight processes) are implicitly resumed for the duration of that function call in order to avoid deadlock. Non-active threads cannot be stepped.
`step` *n*	Single step *n* lines (step into calls).
`step up`	Step up and out of the current function.
`step ... -sig` *signal*	Deliver the given signal while stepping.

`step ... thread_id`	Step the given thread. Does not apply to `step up`.
`step ... lwp_id`	Step the given LWP. Does not implicitly resume all LWPs when stepping over a function.
step to [*function*]	Attempts to step into *func* in the current source code line. If *func* is not given, attempts to step into the last function called as determined by the assembly code for the current source code line.

where:

n is the number of lines to step.

signal is the name of a signal.

thread_id is a thread ID.

lwp_id is an LWP ID.

function is a function name.

When an explicit *thread_id* or *lwp_id* is given, the deadlock avoidance measure of the generic `step` command is defeated.

When executing the `step to` command, while an attempt is made to step into the last assemble call instruction or step into a function (if specified) in the current source code line, the call may not be taken due to a conditional branch. In a case where the call is not taken or there is no function call in the current source code line, the `step to` command steps over the current source code line. Take special consideration on user-defined operators when using the `step to` command.

Java Mode Syntax

`step`	Single step one line (step into calls). With multithreaded programs when a method call is stepped over, all LWPs (lightweight processes) are implicitly resumed for the duration of that method call in order to avoid deadlock. Non-active threads cannot be stepped.
`step n`	Single step *n* lines (step into calls).
`step up`	Step up and out of the current method.

`step ... tid`	Step the given thread. Does not apply to `step up`.
`step ... lwpid`	Step the given LWP. Does not implicitly resume all LWPs when stepping over a method.
`step to [method]`	Attempts to step into *method* in the current source code line. If *method* is not given, attempts to step into the last method called as determined by the assembly code for the current source code line.

When executing the `step to` command, while an attempt is made to step into the last assembler call instruction or step into a method (if specified) in the current source code line, the call may not be taken due to a conditional branch. In a case where the call is not taken or there is no method call in the current source code line, the `step to` command steps over the current source code line. Take special consideration on user-defined operators when using the `step to` command.

See also "`stepi` Command" on page 374 for machine-level stepping.

`stepi` Command

The `stepi` command steps one machine instruction (stepping into calls). It is valid only in native mode.

Syntax

`stepi`	Single step one machine instruction (step into calls).
`stepi n`	Single step *n* machine instructions (step into calls).
`stepi -sig signal`	Step and deliver the given signal.
`stepi ... lwp_id`	Step the given LWP.
`stepi ... thread_id`	Step the LWP on which the given thread is active.

where:

n is the number of instructions to step.

signal is the name of a signal.

lwp_id is an LWP ID.

thread_id is a thread ID.

`stop` Command

The `stop` command sets a source-level breakpoint.

Syntax

The `stop` command has the following general syntax:

`stop` *event-specification* `[` *modifier* `]`

When the specified event occurs, the process is stopped.

Native Mode Syntax

The following specific syntaxes are valid in native mode.

`stop [ -update ]`	Stop execution now. Only valid within the body of a when command.
`stop -noupdate`	Same as `-update`, but does not update dbx Debugger views.
`stop access` *mode address_expression* [*,byte_size_expression*]	Stop execution when the memory specified by *address_expression* has been accessed. See also "Stopping Execution When an Address Is Accessed" on page 107.
`stop at` *line_number*	Stop execution at *line_number*. See also "Setting a `stop` Breakpoint at a Line of Source Code" on page 102.
`stop change` *variable*	Stop execution when the value of *variable* has changed.
`stop cond` *condition_expression*	Stop execution when the condition denoted by *condition_expression* evaluates to true.
`stop in` *function*	Stop execution when *function* is called. See also "Setting a `stop` Breakpoint in a Function" on page 103.
`stop inclass` *class_name* `[ -recurse | -norecurse]`	C++ only: Set breakpoints on all member functions of a class, struct, union, or template class. `-norecurese` is the default. If `-recurse` is specified, the base classes are included. See also "Setting Breakpoints in Member Functions of the Same Class" on page 105.

`stop infunction` *name*	C++ only: Set breakpoints on all non-member functions *name*.
`stop inmember` *name*	C++ only: set breakpoints on all member functions *name*. See "Setting Breakpoints in Member Functions of Different Classes" on page 104.
stop inobject *object_expression* `[ -recurse | -norecurse]`	C++ only: set breakpoint on entry into any non-static method of the class and all its base classes when called from the object *object_expression*. `-recurse` is the default. If `-norecurse` is specified, the base classes are not included. See also "Setting Breakpoints in Objects" on page 106.

where:

line is the number of a source code line.

function is the name of a function.

class_name is the name of a C++ class, struct, union, or template class.

mode specifies how the memory was accessed. It can be composed of one or all of the letters:

`r`	The memory at the specified address has been read.
`w`	The memory has been written to.
`x`	The memory has been executed.

mode can also contain the following:

`a`	Stops the process after the access (default).
`b`	Stops the process before the access.

name is the name of a C++ function.

object_expression identifies a C++ object.

variable is the name of a variable.

The following modifiers are valid in native mode.

`-if` *condition_expression*	The specified event occurs only when *condition_expression* evaluates to true.
`-in` *function*	Execution stops only if the specified event occurs in *function*.
`-count` *number*	Starting at 0, each time the event occurs, the counter is incremented. When *number* is reached, execution stops and the counter is reset to 0.
`-count infinity`	Starting at 0, each time the event occurs, the counter is incremented. Execution is not stopped.
`-temp`	Create a temporary breakpoint that is deleted when the event occurs.
`-disable`	Create the breakpoint in a disabled state.
`-instr`	Do instruction level variation. For example, `step` becomes instruction level stepping, and `at` takes a text address for an argument instead of a line number.
`-perm`	Make this event permanent across `debug`. Certain events (like breakpoints) are not appropriate to be made permanent. `delete all` will not delete permanent handlers, use `delete` *hid*.
`-hidden`	Hide the event from the `status` command. Some import modules may choose to use this. Use `status -h` to see them.
`-lwp` *lwpid*	Execution stops only if the specified event occurs in the given LWP.
`-thread` *tid*	Execution stops only if the specified event occurs in the given thread.

Java Mode Syntax

The following specific syntaxes are valid in Java mode.

`stop access` *mode class_name.field_name*	Stop execution when the memory specified by *class_name.field_name* has been accessed.
`stop at` *line_number*	Stop execution at *line_number*.
`stop at` *file_name*`:`*line_number*	Stop execution at *line_number* in *file_name*.
`stop change` *class_name.field_name*	Stop execution when the value of *field_name* in *class_name* has changed.
`stop classload`	Stop execution when any class is loaded.

`stop classload` *class_name*	Stop execution when *class_name* is loaded.
`stop classunload`	Stop execution when any class is unloaded.
`stop classunload` *class_name*	Stop execution when *class_name* is unloaded.
`stop cond` *condition_expression*	Stop execution when the condition denoted by *condition_expression* evaluates to true.
`stop in` *class_name*`.`*method_name*	Stop execution when *class_name.method_name* has been entered, and the first line is about to be executed. If no parameters are specified and the method is overloaded, a list of methods is displayed.
`stop in` *class_name*`.`*method_name* `(`[*parameters*]`)`	Stop execution when *class_name.method_name* has been entered, and the first line is about to be executed.
`stop inmethod` *class_name*`.`*method_name*	Set breakpoints on all non-member methods *class_name*`.`*method_name*.
`stop inmethod` *class_name*`.`*method_name* `(`[*parameters*]`)`	Set breakpoints on all non-member methods *class_name*`.`*method_name*.
`stop throw`	Stop execution when a Java exception has been thrown.
`stop throw` *type*	Stop execution when a Java exception of *type* has been thrown.

where:

class_name is the name of a Java class, using either the package path (using period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`) or the full path name (preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers; for example, `#test1/extra/T1$Inner`). Enclose *class_name* in quotation marks if you use the `$` qualifier.

condition_expression can be any expression, but it must evaluate to an integral type.

field_name is the name of a field in the class.

file_name is the name of a file.

line_number is the number of a source code line.

method_name is the name of a Java method.

mode specifies how the memory was accessed. It can be composed of one or all of the letters:

`r`	The memory at the specified address has been read.
`w`	The memory has been written to.

mode can also contain the following:

`b`	Stops the process before the access.

The program counter will point at the offending instruction.

parameters are the method's parameters.

type is a type of Java exception. `-unhandled` or `-unexpected` can be used for type.

The following modifiers are valid in Java mode:

`-if` *condition_expression*	The specified event occurs only when *condition_expression* evaluates to true.
`-count` *number*	Starting at 0, each time the event occurs, the counter is incremented. When *number* is reached, execution stops and the counter is reset to 0.
`-count infinity`	Starting at 0, each time the event occurs, the counter is incremented. Execution is not stopped.
`-temp`	Create a temporary breakpoint that is deleted when the event occurs.
`-disable`	Create the breakpoint in a disabled state.

See also "stopi Command" on page 380 for setting a machine-level breakpoint.

For a list and the syntax of all events see "Setting Event Specifications" on page 277.

stopi Command

The stopi command sets a machine-level breakpoint. It is valid only in native mode.

Syntax

The stopi command has the following general syntax:

stopi *event-specification* [*modifier*]

When the specified event occurs, the process is stopped.

The following specific syntaxes are valid:

stopi at *address*	Stop execution at location *address*.
stopi in *function*	Stop execution when *function* is called.

where:

address is any expression resulting in or usable as an address.

function is the name of a function.

For a list and the syntax of all events see "Setting Event Specifications" on page 277.

suppress Command

The suppress command suppresses reporting of memory errors during runtime checking. It is valid only in native mode.

Note – The suppress command is available only on Solaris platforms.

If the `dbx rtc_auto_suppress` environment variable is on, the memory error at a given location is reported only once.

Syntax

`suppress`	History of `suppress` and `unsuppress` commands (not including those specifying the `-d` and `-reset` options).
`suppress -d`	List of errors being suppressed in functions not compiled for debugging (default suppression). This list is per loadobject. These errors can be unsuppressed only by using the unsuppress with the `-d` option.
`suppress -d` *errors*	Modify the default suppressions for all loadobjects by further suppressing *errors*.
`suppress -d` *errors* `in` *loadobjects*	Modify the default suppressions in the *loadobjects* by further suppressing *errors*.
`suppress -last`	At error location suppress present error.
`suppress -reset`	Set the default suppression to the original value (startup time)
`suppress -r <id> ...`	Remove the unsuppress events as given by the id(s) (id(s) can be obtained with `unsuppress` command (see "`unsuppress` Command" on page 394).
`suppress -r 0 | all | -all`	Remove all the unsuppress events as given by the `unsuppress` command (see "`unsuppress` Command" on page 394)
`suppress` *errors*	Suppress *errors* everywhere
`suppress` *errors* `in` [*functions*] [*files*] [*loadobjects*]	Suppress *errors* in list of *functions*, list of *files*, and list of *loadobjects*.
`suppress` *errors* `at` *line*	Suppress *errors* at *line*.
`suppress` *errors* `at` "*file*":*line*	Suppress *errors* at *line* in *file*.
suppress *errors* addr *address*	Suppress *errors* at location *address*.

where:

address is a memory address.

errors are blank separated and can be any combination of:

all	All errors
`aib`	Possible memory leak - address in block
`air`	Possible memory leak - address in register
`baf`	Bad free
`duf`	Duplicate free
`mel`	Memory leak
`maf`	Misaligned free
`mar`	Misaligned read
`maw`	Misaligned write
`oom`	Out of memory
`rua`	Read from unallocated memory
`rui`	Read from uninitialized memory
`wro`	Write to read-only memory
`wua`	Write to unallocated memory
`biu`	Block in use (allocated memory). Though not an error, you can use `biu` just like *errors* in the suppress commands.

file is the name of a file.

files is the names of one or more files.

functions is one or more function names.

line is the number of a source code line.

loadobjects is one or more loadobject names.

See the "`unsuppress` Command" on page 394 for information on unsuppressing errors.

`sync` Command

The `sync` command shows information about a given synchronization object. It is valid only in native mode.

Note – The `sync` command is available only on Solaris platforms.

Syntax

`sync -info` *address*	Show information about the synchronization object at address.

where:

address is the address of the synchronization object.

`syncs` Command

The `syncs` command lists all synchronization objects (locks). It is valid only in native mode.

Note – The `syncs` command is available only on Solaris platforms.

Syntax

```
syncs
```

`thread` Command

The `thread` command lists or changes the current thread.

Native Mode Syntax

`thread`	Display current thread
`thread` *thread_id*	Switch to thread *thread_id*.

In the following variations, a missing *thread_id* implies the current thread.

`thread -info [` *thread_id* `]`	Print everything known about the given thread.
`thread -hide [` *thread_id* `]`	Hide the given (or current) thread. It will not show up in the generic threads listing.
`thread -unhide [` *tid* `]`	Unhide the given (or current) thread.
`thread -unhide all`	Unhide all threads.
`thread -suspend` *thread_id*	Keep the given thread from ever running. A suspended thread shows up with an "S" in the threads list.
`thread -resume` *thread_id*	Undo the effect of `-suspend`.
`thread -blocks [` *thread_id* `]`	List all locks held by the given thread blocking other threads.
`thread -blocked by [` *thread_id* `]`	Show which synchronization object the given thread is blocked by, if any.

where:

thread_id is a thread ID.

Java Mode Syntax

`thread`	Display current thread
`thread` *thread_id*	Switch to thread *thread_id*.

In the following variations, a missing *thread_id* implies the current thread.

`thread -info [` *thread_id* `]`	Print everything known about the given thread.
`thread -hide [` *thread_id* `]`	Hide the given (or current) thread. It will not show up in the generic threads listing.
`thread -unhide [` *thread_id* `]`	Unhide the given (or current) thread.

`thread -unhide all`	Unhide all threads.
`thread -suspend` *thread_id*	Keep the given thread from ever running. A suspended thread shows up with an “S” in the threads list.
`thread -resume` *thread_id*	Undo the effect of `-suspend`.
`thread -blocks [` *thread_id* `]`	Lists the Java monitor owned by *thread_id*.
`thread -blockedby [` *thread_id* `]`	Lists the Java monitor on which *thread_id* is blocked.

where:

thread_id is a `dbx`-style thread ID of the form `t@`*number* or the Java thread name specified for the thread.

`threads` Command

The `threads` command lists all threads.

Native Mode Syntax

`threads`	Print the list of all known threads.
`threads -all`	Print threads normally not printed (zombies).
`threads -mode all|filter`	Controls whether all threads are printed or threads are filtered. The default is to filter threads. When filtering is on, threads that have been hidden by the `thread -hide` command are not listed.
`threads -mode auto|manual`	Under the dbx Debugger, enables automatic updating of the thread listing.
`threads -mode`	Echo the current modes

Each line of information is composed of the following:

- An * (asterisk) indicating that an event requiring user attention has occurred in this thread. Usually this is a breakpoint.

  An ‘o’ instead of an asterisk indicates that a `dbx` internal event has occurred.

- An > (arrow) denoting the current thread.

- `t@num`, the thread id, referring to a particular thread. The *number* is the `thread_t` value passed back by `thr_create`.
- `b l@num` meaning the thread is bound (currently assigned to the designated LWP), or `a l@num` meaning the thread is active (currently scheduled to run).
- The "Start function" of the thread as passed to `thr_create`. A `?()` means that the start function is not known.
- The thread state, which is one of the following:
  - monitor
  - running
  - sleeping
  - wait
  - unknown
  - zombie
- The function that the thread is currently executing.

Java Mode Syntax

`threads`	Print the list of all known threads.
`threads -all`	Print threads normally not printed (zombies).
`threads -mode all|filter`	Controls whether all threads are printed or threads are filtered. The default is to filter threads.
`threads -mode auto|manual`	Under the dbx Debugger, enables automatic updating of the thread listing.
`threads -mode`	Echo the current modes

Each line of information in the listing is composed of the following:

- An > (arrow) denoting the current thread
- `t@number`, a `dbx`-style thread ID
- The thread state, which is one of the following:
  - monitor
  - running
  - sleeping
  - wait
  - unknown
  - zombie
- The thread name in single quotation marks
- A number indicating the thread priority

`trace` Command

The trace command shows executed source lines, function calls, or variable changes.

The speed of a trace is set using the `dbx` `trace_speed` environment variable.

If `dbx` is in Java mode and you want to set a trace breakpoint in native code, switch to Native mode using the `joff` command (see "`joff` Command" on page 340) or prefix the `trace` command with `native` (see "`native` Command" on page 353).

If `dbx` is in JNI mode and you want to set a trace breakpoint in Java code, prefix the `trace` command with `java` (see "`java` Command" on page 339).

Syntax

The trace command has the following general syntax:

`trace` *event-specification* `[` *modifier*`]`

When the specified event occurs, a trace is printed.

Native Mode Syntax

The following specific syntaxes are valid in native mode:

`trace -file` *file_name*	Direct all trace output to the given *file_name*. To revert trace output to standard output use - for *file_name*. trace output is always appended to *file_name*. It is flushed whenever `dbx` prompts and when the application has exited. The *filename* is always re-opened on a new run or resumption after an attach.
`trace step`	Trace each source line, function call, and return.
`trace next -in` *function*	Trace each source line while in the given *function*
`trace at` *line_number*	Trace given source *line*.
`trace in` *function*	Trace calls to and returns from the given *function*.
`trace inmember` *function*	Trace calls to any member function named *function*.

`trace infunction` *function*	Trace when any function named *function* is called.
`trace inclass` *class*	Trace calls to any member function of *class.*
`trace change` *variable*	Trace changes to the *variable.*

where:

file_name is the name of the file to which you want trace output sent.

function is the name of a function.

line_number is the number of a source code line.

class is the name of a class.

variable is the name of a variable.

The following modifiers are valid in native mode.

`-if` *condition_expression*	The specified event occurs only when *condition_expression* evaluates to true.
`-in` *function*	Execution stops only if the specified event occurs in *function*.
`-count` *number*	Starting at 0, each time the event occurs, the counter is incremented. When *number* is reached, execution stops and the counter is reset to 0.
`-count infinity`	Starting at 0, each time the event occurs, the counter is incremented. Execution is not stopped.
`-temp`	Create a temporary breakpoint that is deleted when the event occurs.
`-disable`	Create the breakpoint in a disabled state.
`-instr`	Do instruction level variation. For example, `step` becomes instruction level stepping, and `at` takes a text address for an argument instead of a line number.
`-perm`	Make this event permanent across `debug`. Certain events (like breakpoints) are not appropriate to be made permanent. `delete all` will not delete permanent handlers, use `delete` *hid*.
`-hidden`	Hide the event from the `status` command. Some import modules may choose to use this. Use `status -h` to see them.
`-lwp` *lwpid*	Execution stops only if the specified event occurs in the given LWP.
`-thread` *thread_id*	Execution stops only if the specified event occurs in the given thread.

Java Mode Syntax

The following specific syntaxes are valid in Java mode.

`trace -file` *file_name*	Direct all trace output to the given *file_name*. To revert trace output to standard output use `-` for *file_name*. trace output is always appended to *file_name*. It is flushed whenever `dbx` prompts and when the application has exited. The *file_name* is always re-opened on a new run or resumption after an attach.
`trace at` *line_number*	Trace *line_number*.
`trace at` *file_name*`.`*line_number*	Trace given source *file_name*`.`*line_number*.
`trace in` *class_name*`.`*method_name*	Trace calls to and returns from *class_name*`.`*method_name*.
`trace in` *class_name*`.`*method_name* `([`*parameters*`])`	Trace calls to and returns from *class_name*`.`*method_name*`([`*parameters*`])`.
`trace inmethod` *class_name*`.`*method_name*	Trace calls to and returns from any method named *class_name*`.`*method_name* is called.
`trace inmethod` *class_name*`.`*method_name* `([`*parameters*`])`	Trace calls to and returns from any method named *class_name*`.`*method_name*`([`*parameters*`])` is called.

where:

class_name is the name of a Java class, using either the package path (using period (`.`) as a qualifier, for example, `test1.extra.T1.Inner`) or the full path name (preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers, for example, `#test1/extra/T1$Inner`). Enclose *class_name* in quotation marks if you use the `$` qualifier.

file_name is the name of a file.

line_number is the number of a source code line.

method_name is the name of a Java method.

parameters are the method's parameters

The following modifiers are valid in Java mode.

`-if` *condition_expression*	The specified event occurs and the trace is printed only when *condition_expression* evaluates to true.
`-count` *number*	Starting at 0, each time the event occurs, the counter is incremented. When *number* is reached, the trace is printed and the counter is reset to 0.
`-temp`	Create a temporary breakpoint that is deleted when the event occurs and the trace is printed. If `-temp` is used with `-count`, the breakpoint is deleted only when the counter is reset to 0.
`-disable`	Create the breakpoint in a disabled state.

For a list and the syntax of all events see "Setting Event Specifications" on page 277.

`tracei` Command

The `tracei command` shows machine instructions, function calls, or variable changes. It is valid only in native mode.

`tracei` is really a shorthand for `trace` *event-specification* `-instr` where the `-instr` modifier causes tracing to happen at instruction granularity instead of source line granularity. When the event occurs, the printed information is in disassembly format instead of source line format.

Syntax

`tracei step`	Trace each machine instruction.
`tracei next -in` *function*	Trace each instruction while in the given *function.*
`tracei at` *address*	Trace the instruction at *address*.
`tracei in` *function*	Trace calls to and returns from the given *function.*
`tracei inmember` *function*	`Trace calls to any member function named` *function.*

`tracei infunction` *function*	Trace when any function named *function* is called.
`tracei inclass` *class*	Trace calls to any member function of *class.*
`tracei change` *variable*	Trace changes to the *variable.*

where:

filename is the name of the file to which you want trace output sent.

function is the name of a function.

line is the number of a source code line.

class is the name of a class.

variable is the name of a variable.

See "`trace` Command" on page 387 for more information.

`uncheck` Command

The `uncheck` command disables checking of memory access, leaks, or usage. It is valid only in native mode.

Note – The `uncheck` command is available only on Solaris platforms.

Syntax

`uncheck`	Print current status of checking.
`uncheck -access`	Turn off access checking.
`uncheck -leaks`	Turn off leak checking.
`uncheck -memuse`	Turn off memuse checking (leak checking is turned off as well).
`uncheck -all`	Equivalent to `uncheck -access`; `uncheck -memuse`.
`uncheck [`*functions*`] [`*files*`] [`*loadobjects*`]`	Equivalent to `suppress all` in *functions files loadobjects.*

where:

functions is one or more function names.

files is one or more file names.

loadobjects is one or more loadobject names

See "`check` Command" on page 304 for information to turn on checking.

See "`suppress` Command" on page 380 for information on suppressing of errors.

See "Capabilities of Runtime Checking" on page 136 for an introduction to runtime checking.

`undisplay` Command

The `undisplay` command undoes `display` commands.

Native Mode Syntax

`undisplay` *expression*`, ...`	Undo a `display` *expression* command.
`undisplay` *n*`, ...`	Undo the `display` commands numbered *n*...
`undisplay 0`	Undo all `display` commands.

where:

expression is a valid expression.

Java Mode Syntax

`undisplay` *expression*`, ...` `|` *identifier*`, ...`	Undo a `display` *expression*`, ...` `or display` *identifier*`, ...` command.
`undisplay` *n*`, ...`	Undo the `display` commands numbered *n*...
`undisplay 0`	Undo all `display` commands.

where:

expression is a valid Java expression.

field_name is the name of a field in the class.

identifier is a local variable or parameter, including `this`, the current class instance variable (*object_name.field_name*) or a class (static) variable (*class_name.field_name*).

`unhide` Command

The `unhide` command undoes `hide` commands. It is valid only in native mode.

Syntax

`unhide 0`	Delete all stack frame filters
`unhide` *regular_expression*	Delete stack frame filter *regular_expression*.
`unhide` *number*	Delete stack frame filter number *number*.

where:

regular_expression is a regular expression.

number is the number of a stack frame filter.

The hide command (see "`hide` Command" on page 336) lists the filters with numbers.

`unintercept` Command

The `unintercept` command undoes `intercept` commands (C++ only). It is valid only in native mode.

Syntax

`unintercept` *typename*	Delete *typename* from `intercept` list.
`unintercept -a`	Delete all types from `intercept` list.
`unintercept -x` *typename*	Delete *typename* from `intercept -x` list.
`unintercept -x -a`	Delete all types from `intercept -x` list.
`unintercept`	List intercepted types

where:

typename may be either `-unhandled` or `-unexpected`.

`unsuppress` Command

The `unsuppress` command undoes `suppress` commands. It is valid only in native mode.

Note – The `unsuppress` command is available only on Solaris platforms.

Syntax

`unsuppress`	History of suppress and unsuppress commands (not those specifying the `-d` and `-reset` options)
`unsuppress -d`	List of errors being unsuppressed in functions that are not compiled for debugging. This list is per loadobject. Any other errors can be suppressed only by using the suppress command (see "`suppress` Command" on page 380) with the `-d` option.
`unsuppress -d` *errors*	Modify the default suppressions for all loadobjects by further unsuppressing *errors*.
`unsuppress -d` *errors* `in` *loadobjects*	Modify the default suppressions in the *loadobjects* by further unsuppressing *errors*.
`unsuppress -last`	At error location unsuppress present error.

`unsuppress -reset`	Set the default suppression mask to the original value (startup time).
`unsuppress` *errors*	Unsuppress *errors* everywhere.
`unsuppress` *errors* `in` [*functions*] [*files*] [*loadobjects*]	Suppress *errors* in list of *functions*, list of *files*, and list of *loadobjects*.
`unsuppress` *errors* `at` *line*	Unsuppress *errors* at *line*.
`unsuppress` *errors* `at` *"file"*`:`*line*	Unsuppress *errors* at *line* in *file*.
`unsuppress` *errors* `addr` *address*	Unsuppress *errors* at location *address*.

`up` Command

The `up` command moves up the call stack (toward `main`). It has identical syntax and identical functionality in native mode and in Java mode.

Syntax

up	Move up the call stack one level.
up *number*	Move up the call stack *number* levels.
up -h [*number*]	Move up the call stack, but don't skip hidden frames.

where:

number is a number of call stack levels.

`use` Command

The `use` command lists or changes the directory search path. It is valid only in native mode.

This command is an anachronism and usage of this command is mapped to the following `pathmap` commands:

`use` is equivalent to `pathmap -s`

`use` *directory* is equivalent to `pathmap` *directory*.

where:

array-expression is an expression that can be depicted graphically.

seconds is a number of seconds.

`whatis` Command

In native mode, the `whatis` command prints the type of expression or declaration of type. In Java mode, the `whatis` command prints the declaration of an identifier. If the identifier is a class, it prints method information for the class, including all inherited methods.

Native Mode Syntax

`whatis [-n] [-r]` *name*	Print the declaration of the non-type *name*.
`whatis -t [-r]` *type*	Print the declaration of the type *type*
`whatis -e [-r] [-d]` *expression*	Print the type of the expression *expression*.

where:

name is the name of a non-type.

type is the name of a type.

expression is a valid expression.

`-d` shows dynamic type instead of static type (C++ only).

`-e` displays the type of an expression.

`-n` displays the declaration of a non-type. It is not necessary to specify `-n`; this is the default if you type the `whatis` command with no options.

`-r` prints information about base classes (C++ only).

`-t` displays the declaration of a type.

The `whatis` command, when run on a C++ class or structure, provides you with a list of all the defined member functions (undefined member functions are not listed), the static data members, the class friends, and the data members that are defined explicitly within that class.

Specifying the `-r` (recursive) option adds information from the inherited classes.

The `-d` flag, when used with the `-e` flag, uses the dynamic type of the expression.

For C++, template-related identifiers are displayed as follows:

- All template definitions are listed with `whatis -t`.
- Function template instantiations are listed with `whatis`.
- Class template instantiations are listed with `whatis -t`.

Java Mode Syntax

`whatis` *identifier*	Print the declaration of *identifier.*

where:

identifier is a class, a method in the current class, a local variable in the current frame, or a field in the current class.

`when` Command

The `when` command executes commands when a specified event occurs.

If `dbx` is in Java mode and you want to set a when breakpoint in native code, switch to Native mode using the `joff` command (see "`joff` Command" on page 340) or prefix the `when` command with `native` (see "`native` Command" on page 353).

If `dbx` is in JNI mode and you want to set a when breakpoint in Java code, prefix the `when` command with `java` (see "`java` Command" on page 339).

Syntax

The `when` command has the following general syntax:

`when` *event-specification* `[` *modifier* `]{` *command*`;` `...` `}`

When the specified event occurs, the commands are executed.

Native Mode Syntax

The following specific syntaxes are valid in native mode:

`when at` *line_number* `{` *command*`;` `}`	Execute *command*(s) when *line_number* is reached.
`when in` *procedure* `{` *command*`;` `}`	Execute *command*(s) when *procedure* is called.

where:

line_number is the number of a source code line.

command is the name of a command.

procedure is the name of a procedure.

Java Mode Syntax

The following specific syntaxes are valid in Java mode.

`when at` *line_number*	Execute command(s) when source *line_number* is reached.
`when at` *file_name*`.`*line_number*	Execute command(s) when *file_name.line_number* is reached.
`when in` *class_name*`.`*method_name*	Execute command(s) when *class_name.method_name* is called.
`when in` *class_name*`.`*method_name* `(`[*parameters*]`)`	Execute command(s) when *class_name.method_name*([*parameters*]) is called.

class_name is the name of a Java class, using either the package path (using period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`) or the full path name (preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers; for example, `#test1/extra/T1$Inner`). Enclose *class_name* in quotation marks if you use the `$` qualifier.

file_name is the name of a file.

line_number is the number of a source code line.

method_name is the name of a Java method.

parameters are the method's parameters.

For a list and the syntax of all events, see "Setting Event Specifications" on page 277.

See "`wheni` Command" on page 399 for executing commands on given low-level event.

`wheni` Command

The `wheni` command executes commands when a given low-level event occurs. It is valid only in native mode.

The wheni command has the following general syntax:

Syntax

`wheni` *event-specification* `[` *modifier* `]{` *command* `...` `;` `}`

When the specified event occurs, the commands are executed.

The following specific syntax is valid:

`wheni at` *address* `{` *command*`;` `}`	Execute *command*(s) when *address* is reached.

where:

address is any expression resulting in or usable as an address.

command is the name of a command.

For a list and the syntax of all events see "Setting Event Specifications" on page 277.

`where` Command

The `where` command prints the call stack.

Native Mode Syntax

`where`	Print a procedure traceback.
`where` *number*	Print the *number* top frames in the traceback.
`where -f` *number*	Start traceback from frame *number*.
`where -h`	Include hidden frames.

`where -l`	Include library name with function name
`where -q`	Quick traceback (only function names).
`where -v`	Verbose traceback (include function args and line info).

where:

number is a number of call stack frames.

Any of the above forms may be followed by a thread or LWP ID to obtain the traceback for the specified entity.

Java Mode Syntax

`where [`*thread_id*`]`	Print a method traceback.
`where [`*thread_id*`]` *number*	Print the *number* top frames in the traceback.
`where -f [`*thread_id*`]` *number*	Start traceback from frame *number*.
`where -q [`*thread_id*`]`	Quick trace back (only method names).
`where -v [`*thread_id*`]`	Verbose traceback (include method arguments and line information).

where:

number is a number of call stack frames.

thread_id is a `dbx`-style thread ID or the Java thread name specified for the thread.

`whereami` Command

The `whereami` command displays the current source line. It is valid only in native mode.

Syntax

`whereami`	Display the source line corresponding to the current location (top of the stack), and the source line corresponding to the current frame, if different.
`whereami -instr`	Same as above, except that the current disassembled instruction is printed instead of the source line.

`whereis` Command

The `whereis` command prints all uses of a given name, or symbolic name of an address. It is valid only in native mode.

Syntax

`whereis` *name*	Print all declarations of *name*.
`whereis -a` *address*	Print location of an *address* expression.

where:

name is the name of a loadable object that is in scope; for example, a variable, function, class template, or function template.

address is any expression resulting in or usable as an address.

`which` Command

The `which` command prints the full qualification of a given name. It is valid only in native mode.

Syntax

`which [-n]` *name*	Print full qualification of *name*.
`which -t` *type*	Print full qualification of *type*.

where:

name is the name of something that is in scope; for example, a variable, function, class template, or function template.

type is the name of a type.

`-n` displays the full qualification of a non-type. It is not necessary to specify `-n`; this is the default if you type the `which` command with no options.

`-t` displays the full qualification of a type.

`whocatches` Command

The `whocatches` command tells where a C++ exception would be caught. It is valid only in native mode.

Syntax

`whocatches` *type*	Tell where (if at all) an exception of type *type* would be caught if thrown at the current point of execution. Assume the next statement to be executed is a `throw` *x* where *x* is of type *type*, and display the line number, function name, and frame number of the `catch` clause that would catch it. Will return "*type* `is unhandled`" if the catch point is in the same function that is doing the throw.

where:

type is the type of an exception.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X