WorkShop: Command-Line Utilities

THE NETWORK IS THE COMPUTER™

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 802-5763-10
Revision A, December 1996

Contents

Preface

WorkShop: Command-Line Utilities provides the reference information needed to perform Sun WorkShop™ functions from the command line.

Who Should Use This Book

This book is designed to assist users who choose to perform WorkShop functions from the command line instead of using WorkShop's graphical user interface (GUI).

The audience for this book includes programmers, developers, and engineers who need, or prefer, to work directly from a command line.

Before You Read This Book

For an overview and details about using the WorkShop suite of applications, see *WorkShop: Getting Started.*

If you are building, analyzing, debugging, or optimizing distributed, multithreaded, and multiprocessor applications and programs, you will need the information in *WorkShop: Beyond the Basics*.

DMake users will find basic information about the Sun WorkShop™ TeamWare suite of applications in *Sun WorkShop TeamWare: User's Guide* and *Sun WorkShop TeamWare: Solutions Guide*.

How This Book Is Organized

Part 1, "Using `dbx`**,"** explains the commands and topics you need to debug programs from the command line.

Part 2, "Using Multithreaded Tools," explains how to request and use a loop report and explains the LockLint commands and utilities.

Part 3, "Using Source Browsing," explains how to use `sbquery`, `sbinit`, and `sbtags` to browse sources from the command line.

Part 4, "Using Merging," explains how to use Merging to compare text files.

How to Get Help

This release of WorkShop includes a new documentation delivery system as well as online manuals and video demonstrations. To find out more, you can start in any of the following places:

Apart from this manual, you can also access help online in two modes: from the WorkShop help menu, and in the `dbx` Commands window.

- From the main Workshop window, choose Help ➤ Help Contents.
- In the Dbx Commands window, at the `dbx` prompt:
  - *All commands*—a list of commands, grouped by action, type `help`.
  - *Details of one command*—a command explanation, type `help` *cmdname*.
  - *Changes*—a list of the new and changed features, type `help changes`
  - *FAQ*—answers to frequently asked questions, type `help FAQ`
- **Online Help** – A new help system containing extensive task-oriented, context-sensitive help. To access the help, choose Help ➤ Help Contents. Help menus are available from all WorkShop windows.
- **WorkShop Documentation** – A complete set of online manuals, which make up the complete documentation set for WorkShop and are available using AnswerBook™ or any HTML browser. To access the online manuals, choose Help ➤ WorkShop Manuals from any WorkShop window, or type **`answerbook &`**.
- **Video Demonstrations** – These demos provide a general overview of WorkShop and describe how to use WorkShop to build targets or debug programs. To access them, choose Help ➤ Demos from the WorkShop main window.

- **Release Notes** – The Release Notes contain last-minute information about WorkShop and specific information about software limitations and bugs. To access the Release Notes, choose Help ➤ Release Notes.
- **Manual Pages** – The man pages provide information about the command-line utilities of the SunOS operating system. Each tool has at least one man page. To access the man pages, type **man** *utility_name*.

How to Access the AnswerBook Documentation

To access the AnswerBook online documentation for WorkShop, you must run a script to set up your environment.

- At a command prompt, type:

```
% workshop-answerbooks
```

The script sets the AB_CARDCATALOG environment variable and runs /usr/openwin/bin/answerbook. The AnswerBook Navigator opens and displays the available AnswerBook documents.

Related Books

Sun WorkShop provides comprehensive documentation. Depending on which WorkShop you have, the following books are available in online and printed forms.

Sun WorkShop Documentation

Available with all WorkShop products.

WorkShop Roadmap	Provides a map to installation and use of your version of WorkShop. Includes a complete list of the documentation included with your WorkShop.
WorkShop Installation and Licensing Guide	Provides instructions about product licensing and installation of WorkShop products. Provides instructions for local or remote installation for single independent license servers, multiple independent license servers, and redundant license servers.
WorkShop: Getting Started	Provides the information you need to use the basic WorkShop features.

WorkShop: Beyond the Basics	Contains information about the advanced programming, debugging, browsing, and visualization applications in the WorkShop product suite, including: DMake, LoopTool, Thread Analyzer, and WorkShop Visual.
WorkShop: Command-Line Utilities	Contains instructions for command-line use of dbx, multithreaded tools, the source browser, and merging.
WorkShop Online Help	Contains extensive task-oriented information for the tools included with WorkShop.
WorkShop Video Demonstrations	Consists of three video demonstrations providing information about WorkShop building and debugging as well as general product information.
Release Notes	Contains last-minute information regarding WorkShop, including any software incompatibilities and limitations. To access the Release Notes, choose Help ➤ Release Notes.
Manual Pages	Provide information about the WorkShop command-line utilities.

Sun WorkShop TeamWare Documentation

Available only with Sun Performance WorkShop Fortran and Sun Visual WorkShop C++.

Sun WorkShop TeamWare: User's Guide	Describes how to use all the tools in the TeamWare toolset, for both the command-line interface and the graphical user interface.
Sun WorkShop TeamWare: Solutions Guide	Provides an in-depth case study and eight scenario-based topics to help users take full advantage of TeamWare's features.
Sun WorkShop TeamWare Online Help	Provides succinct task-oriented information to help you become familiar with the application. Help volume includes video demonstrations.
Manual Pages	Provide information about the TeamWare commands and utilities.

Sun Visual WorkShop C++ Documentation

Available only with Sun Visual WorkShop C++.

WorkShop documentation	Visual WorkShop C++ contains the entire WorkShop and TeamWare documentation sets.

C++ User's Guide	Describes how to use the Sun Compiler C++ to write programs in C++. It covers the C++ compiler options, programs, templates, exception handling, and more. It is intended for the experienced C++ programmer.
C++ Library Reference	Describes how to use the complex, coroutine, and iostream libraries, and lists the manual pages for these libraries.
Tools.h++ Class Library Reference	Describes how to use the Tools.h++ class library, and a set of C++ classes that can simplify programming while maintaining efficiency.
C++ Quick Reference Card	Provides concise descriptions of the C++ compiler flags.
C User's Guide	Describes how to use the Sun Compiler ANSI C to write programs in C. It covers the C compiler options, the pragmas, the lint tool, the cscope tool, and more. Intended for the experienced C programmer.
WorkShop: Visual User's Guide	Explains how to use Visual, an interactive tool for building graphical user interfaces (GUIs) using the widgets of the standard OSF/Motif toolkit or Microsoft Foundation Class. It includes a tutorial as well as reference information for the more advanced user.
Visual Quick Reference Card	Contains menu shortcuts and icon explanations for Visual.
Numerical Computation Guide	Describes the floating-point software and hardware for the SPARC™, Intel, and PowerPC architectures. It also contains a tutorial on floating-point arithmetic.
Incremental Link Editor (ild)	Describes how to use `ild` as an incremental linker to replace `ld` for linking programs. `ild` allows you to complete the development sequence more quickly than a standard linker.
Performance Profiling Tools	Describes the `prof` (1), `gprof` (1), and `tcov`(1) utilities.
Manual Pages	Provide information about the command-line commands and utilities included with the Visual WorkShop C++.

Sun Performance WorkShop Fortran Documentation

Available only with Sun Performance WorkShop Fortran.

WorkShop documentation	Contains the entire WorkShop and TeamWare documentation set.

Fortran User's Guide	Describes how to use the Sun Compiler Fortran 77 4.0 and the Sun Compiler Fortran 90 1.2, including the compiler command options, debugging and development tools, program profiling and performance tuning, mixing C and Fortran, and making and using libraries. Intended for programmers with knowledge of Fortran.
FORTRAN 77 Language Reference	Describes and defines the Fortran 77 language accepted by the Sun Compiler `f77`. Intended for use by programmers with knowledge of and experience with Fortran.
Fortran Programmer's Guide	Provides the essential information programmers need to develop efficient applications using the Sun Compiler Fortran 77 4.0 and the Sun Compiler Fortran 90 1.2. Includes information on input/output, program development, use and creation of software libraries, program analysis and debugging, numerical accuracy, porting, performance, optimization, parallelization, and the C/Fortran interface.
Fortran Library Reference	Describes the language and routines of the Fortran compilers.
Fortran 90 Browser	Describes how to use the Sun Fortran 90 Browser, one of the development tools in the `f90` package, to view Fortran 90 source code. Intended for programmers with knowledge of Fortran 90.
Fortran Quick Reference Card	Lists the Sun Compiler `f77` 4.0's command-line options with brief descriptions.
Numerical Computation Guide	Describes the floating-point software and hardware for the SPARC, Intel, and PowerPC architectures. It also contains a tutorial on floating-point arithmetic.
Incremental Link Editor (ild)	Describes how to use `ild` as an incremental linker to replace `ld` for linking programs. `ild` allows you to complete the development sequence more quickly than with a standard linker.
Performance Profiling Tools	Describes the `prof` (1), `gprof` (1), and `tcov`(1) utilities.
Manual Pages	Provide information about the Fortran command-line commands and utilities.

Sun WorkShop Professional Pascal Documentation

Available only with Sun WorkShop Professional Pascal.

WorkShop documentation	Contains the entire WorkShop documentation set.

Pascal User's Guide	Describes how to begin writing and compiling Pascal programs on Solaris. Pascal is a derivative of the Berkeley Pascal system distributed with UNIX® 4.2 BSD. It complies with FIPS PUB 109 ANSI/IEEE 770 X3.97-1983 and BS6192/ISO7185 at both level 0 and level 1, and it includes many extensions to the standard.
Pascal Language Reference	Provides reference material for the Sun Compiler Pascal 4.0, an implementation of the Pascal language that includes all the standard language elements and many extensions. Pascal 4.0 contains a compiler switch, `-xl`, to provide compatibility with Apollo DOMAIN Pascal to ease the task of porting your Apollo Pascal applications to workstations.
Pascal Quick Reference Card	Lists all of the Sun Compiler Pascal 4.0 options with a brief, one-line description of each option.
Numerical Computation Guide	Describes the floating-point software and hardware for the SPARC, Intel, and PowerPC architectures. It also contains a tutorial on floating-point arithmetic.
Incremental Link Editor (ild)	Describes how to use `ild` as an incremental linker to replace `ld` for linking programs. `ild` allows you to complete the development sequence more quickly than with a standard linker.
Performance Profiling Tools	Describes the `prof` (1), `gprof` (1), and `tcov`(1) utilities.
Manual Pages	Provide information about the command-line commands and utilities.

Sun WorkShop Professional C Documentation

Available only with Sun WorkShop Professional C.

WorkShop documentation	Contains the entire WorkShop documentation set.
C User's Guide	Describes how to use the Sun Compiler ANSI C to write programs in C. It covers the C compiler options, the pragmas, the lint tool, the cscope tool, and more. Intended for the experienced C programmer.
C Quick Reference Card	Describes the C compiler options in a concise and easy-to-read format.
Numerical Computation Guide	Describes the floating-point software and hardware for the SPARC, Intel, and PowerPC architectures. It also contains a tutorial on floating-point arithmetic.

Incremental Link Editor (ild)	Describes how to use `ild` as an incremental linker to replace `ld` for linking programs. `ild` allows you to complete the development sequence more quickly than a standard linker.
Performance Profiling Tools	Describes the `prof` (1), `gprof` (1), and `tcov`(1) utilities.
Manual Pages	Provide information about the command-line commands and utilities.

Ordering Additional Hardcopy Documentation

You can order additional copies of the hard-copy documentation by calling SunExpress at 1-800-USE-SUNX or visiting the SunExpress Web page at:

`http://sunexpress.usec.sun.com.`

Sun on the World Wide Web

World Wide Web users can view Sun's Developer Products site at the following URL:

`http://sun-www.EBay.Sun.COM:80/sunsoft/Developer-products/`

This area is updated regularly and contains helpful information, including current release and configuration tables, special programs, and success stories.

Sun Education Classes

Sun Educational Services offers many classes for programmers who are developing applications. For more information about classes offered, contact Sun Education by telephone or email:

Sun Education Registrar 1-800-422-8020 or (408) 263-9367

`training_seats@sun.com` (schedule and availability)

`edbrochure@sun.com` (class description)

Or go to the Sun Educational Services Web site:

`http://www.sun.com/sunservice/suned`

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

Table P-1 Typographic Conventions

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

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell, as well as the default prompt when `dbx` is running in a Korn shell.

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	#
Korn shell running `dbx`	(dbx)

Part 1 — Using dbx

A dbx *Overview* 1

dbx is an interactive debugging tool which provides facilities to run a program in a controlled fashion, and to inspect the state of a stopped program. dbx gives you complete control of the dynamic execution of a program, including the collection of performance data. This section contains the following:

Starting dbx describes how to start and stop a debugging session, discusses compiling options, and describes how to save all or part of a debugging run and replay it later.

Viewing and Visiting Code covers visiting code, visiting functions, locating symbols and looking up variables, members, types, and classes.

Controlling Program Execution describes how to run, attach to, continue, stop, and rerun a program in dbx, and how to single-step through program code.

Examining the Call Stack describes how to examine the *call stack,* and how to debug a core file with the where command.

Evaluating and Displaying Data shows you how to evaluate data, display the value of expressions, variables, and other data structures, and how to assign a value to an expression.

Setting Breakpoints and Traces describes common operations, such as how to set, clear, and list breakpoints and traces, and how to use watchpoints.

Event Management describes how to manage events, and describes the general capability of dbx to perform certain actions when certain events take place in the program being debugged.

1

Using Runtime Checking is a feature that allows you to automatically detect runtime errors in an application during the development phase.

Using Fix and Continue describes how to modify a source file without leaving dbx, recompile the file, and continue execution of the program.

Collecting Data describes the dbx collector commands you can use to collect performance data.

Debugging Multithreaded Applications describes how to find information about threads by using the dbx thread commands.

Customizing dbx describes how to adjust dbxenv variables to customize certain attributes of your debugging environment, and how to use the initialization file, .dbxrc, to preserve changes and adjustments from session to session.

Debugging at the Machine Instruction Level 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.

Debugging Child Processes describes several dbx facilities to help you debug processes that create children.

Working with Signals describes how to use dbx to work with signals.

Debugging C++ describes dbx's support of C++ templates, and discusses the commands that are available for handling C++ exceptions, and how dbx handles these exceptions.

Debugging Fortran using dbx introduces some dbx features to be used with Fortran.

dbx *and the Dynamic Linker* describes dbx's debugging support for programs that use dynamically-linked, shared libraries.

Using the KornShell explains differences between ksh-88 and dbx command language.

Modifying a Program State discusses the dbx operations that modify the state of a program.

User Tips recommends timesavers, warns of common pitfalls, and suggests some debugging techniques for dbx users.

Starting dbx 2

This chapter describes how to start, execute, save, restore, and quit a dbx debugging session.

Topics covered in this chapter are:

Basic Concepts

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.

Starting a Debugging Session

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

To start dbx from a shell, type:

```
$ dbx
```

dbx *Start-up Sequence*

Upon invocation, dbx looks for and reads the installation startup file, dbxrc in the directory *install-directory*/lib.

Next, dbx searches for the startup file .dbxrc in the current directory, then in $HOME. If the file is not found, it searches for the startup file .dbxinit in the current directory, then in $HOME. Generally, the contents of .dbxrc and .dbxinit files are the same with one major exception. In the .dbxinit file, the alias command is defined to be dalias and not the normal default of kalias. A different startup file may be given explicitly with the -s command-line option.

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 in before the program or process is loaded. The startup file may also *source* other files using the source or .(period) command. The startup file is also used to set other dbx options.

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

Once the program is finished loading, dbx is in a ready state, visiting the "main" block of the program (For C, C++, or Fortran 90: main(); for FORTRAN 77: MAIN(); and for Pascal: program()). Typically, you want to set a breakpoint and then issue a run command, such as stop in main and run for a C program.

If a Core File Exists

If a file named core exists in the directory where you start dbx, it is not read in by default; you must explicitly request the core file. Use the where command to see where the program was executing when it dumped core.

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

Process ID

You can attach a running process to `dbx` using the *process_id* (*pid*) as an argument to the `dbx` command.

```
$ dbx your_program_name pid
```

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

```
$ dbx - pid
```

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

Setting Startup Properties

`pathmap`

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 / object files are not there or the machine you are using doesn't use the same pathname, you must inform `dbx` of their location.

If you move the source / 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.

Common pathmaps should be added to your `.dbxrc` file.

To establish a new mapping from directory *from* to 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 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.

dbxenv

The `dbxenv` command can be used to either list or set `dbx` customization variables. Customize your `dbxenv` setting by placing them in your `.dbxrc` file. To list variables, type:

```
dbxenv
```

You can also set `dbx` environment variables. See Chapter , “Customizing dbx” on page 161 for more information about setting these variables.

alias

You can create your own `dbx` commands using the `kalias` or `dalias` commands.

Debugging Optimized Code

`dbx` 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

However, with optimized code, dbx cannot:

- Single-step from one line to another (next or step command)
- Evaluate, display, or modify local variables

Compiling with the -g Option

To use dbx effectively, a program must have been compiled with the -g or -g0 option. The -g option instructs the compiler to generate debugging information during compilation.

The -g0 (zero) option is for C++ support.

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

For example, to compile using C++:

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

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 also applies to code that has had strip -x applied to it.

Shared Libraries Need -g for Full dbx Support

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

C++ Support and the `-g` *Option*

In C++, `-g` 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 this option. The `-g0` option can significantly decrease link time and `dbx` start-up time (depending on the use of inlined functions by the program).

Completely Stripped Programs

`dbx` 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. Runtime Checking cannot work on stripped programs or load objects.

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 Execution

You can stop execution of a process at any time using 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 using the `detach` command.

To detach a process from dbx without killing the process:

```
(dbx) detach
```

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:

```
(dbx) kill
```

Saving and Restoring a Debugging Run

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

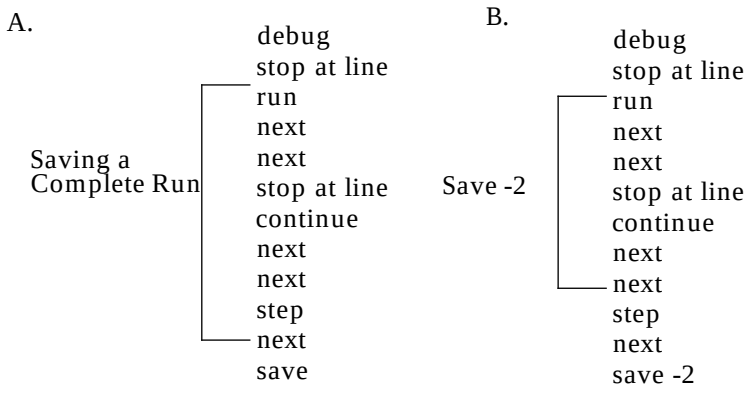
- `save [-`*number*`] [`*filename*`]`
- `restore [`*filename*`]`
- `replay [-`*number*`]`

save

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.

To save all of a debugging run up to the `save` command:

```
(dbx) save
```

To save part of a debugging run

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

where *number* is the number of commands back from the save command that you do *not* want saved.

Saving a Series of Debugging Runs as Checkpoints

If you save a debugging run without specifying a *filename*, `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 back 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 how 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 merely 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* of 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 may not get an exact restoration.

To restore a saved debugging run:

```
(dbx) restore
```

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

```
(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_of_ commands* argument, which it passes to the s`ave` 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:

```
(dbx) replay
```

To replay the current debugging run and stop the run before the second to last command, use the `dbx replay` command where *number* is the number of commands back from the last debugging command:

```
(dbx) replay -number
```

Command Reference

Syntax

```
$ dbx [- options] [program name [corefile | process id]]
```

Start-up Options

`-c` *cmds*	Execute *cmds* before prompting for input and after loading the program
`-C`	Preload the runtime checking library
`-d`	Used with `-s`, removes *file* after reading
`-f`	Force loading of corefile, even if it doesn't appear to match
`-F`	Enable Cfront demangling
`-h`	Print the usage help on `dbx`
`-I` *dir*	Add *dir* to *pathmap* set
`-k`	Save and restore keyboard translation state
`-q`	Suppress messages about reading stabs
`-r`	Run program; if program exits normally, quit `dbx`
`-R`	Print the `README` file on `dbx`
`-s` *file*	Use *file* instead of `.dbxrc` or `.dbxinit` as the startup file
`-S`	Suppress reading of the installation startup file
-V	Print the version of `dbx`
-w *n*	Skip *n* frames on `where` command.
--	Marks the end of the option list; use if the program name starts with a dash

2

Viewing and Visiting Code 3

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

This chapter is organized into the following sections

Basic Concepts

Each time the program 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 visit, functions and files elsewhere in the program.

3

Mapping to the Location of the Code

`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 one they 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 or change to the new location before debugging.

If you move the source/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.

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

```
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.

Visiting Code

You can visit code elsewhere in the program any time the program is not running. You can visit any function or file that is part of the program.

Visiting a File

You can visit any file `dbx` recognizes as part of the program (even if a module or file was not compiled with the `-g` option.) Visiting a file does *not* change the current function. To visit a file:

```
file filename
```

Using the file command by itself echoes the file you are currently visiting:

```
file
```

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

```
file filename ; list line_number
```

Visiting Functions

You can use the `func` command to visit a function. To visit a function, type the command `func` followed by the function name. The `func` command by itself echoes the currently visited function. For example:

```
(dbx) func adjust_speed
```

Selecting from a List of C++ Ambiguous Function Names

If you try to visit a C++ member function with an ambiguous name or an overloaded function name, a list is displayed, showing all functions with the overloaded name.

If the specified function is ambiguous, type the number of the function you want to visit. If you know which specific class a function belong to, you can type:

```
(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 block::block
Class block has more than one function member named block.
```

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` merely echoes the name.

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

Printing a Source Listing

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

```
list [-i | -instr] [+] [-] number [ function | filename ]
```

Walking the Call Stack to Visit Code

Another way to visit code when a live process exists is to “walk the call stack,” using the stack commands to view functions currently on the stack.

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.

Qualifying Symbols with Scope Resolution Operators

When using `func` or `file`, you may 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 visiting code, symbol name qualifying is also necessary for printing and displaying out-of-scope variables and expressions, and for displaying type and class declarations (`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

The backquote character(`‘`) can be used to find a variable of global scope:

```
(dbx) print ‘item
```

A program may 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 knows which function you mean to visit. To qualify a function name with respect to its filename, 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 or top level function with:

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

You may want to qualify an overloaded function name. If you do not qualify it, dbx pops up an overload display list for you to choose which function you mean to visit. 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:

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

Block Local Operator

The block local operator (:*lineno*) is used in conjunction with the backquote operator. It identifies the line number of an expression that references the instance you're interested in.

In the following example, :230 is the block local operator:

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

Linker Names

dbx provides a special syntax for looking up symbols by their linker names (mangled names in C++). You 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
```

Scope Resolution Search Path

When you issue a debugging command with a *symbol* target name, the search order is as follows:

1. dbx first *searches within the scope of the current function*. If the program is stopped in a nested block, dbx searches within that block, then in the scope of all enclosing blocks.
2. For Pascal only: the next immediately enclosing function.
3. For C++ only: class members of the current function‘s class and its base class.
4. The immediately enclosing “compilation unit”: generally, the file containing the current function.
5. The LoadObject scope.
6. The global scope.
7. If none of the above searches are successful, dbx assumes you are referencing a private, or file static, variable or function. dbx optionally searches for a file static symbol in every compilation unit depending on the value of the dbxenv setting scope_lookaside.

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

Locating Symbols

In a program, the same name may 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 uses if you give that name as the target of a debugging command.

Printing a List of Occurrences of a Symbol

To print a list of all the occurrences of a specified symbol, use the `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 knows only about occurrences of a symbol that are already loaded. As a debugging session gets longer, the list of occurrences may grow.

Determining Which Symbol `dbx` *Uses*

The `which` command tells you which symbol with a given name it uses if you specify that name (without fully qualifying it) as the target of a debugging command. 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
`example‘file1.c‘fid
‘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 merely echoes the name.

Remember that 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 know which occurrence of two or more names it uses. dbx lists the possibilities and waits for you to choose one.

Viewing Variables, Members, Types, and Classes

dbx’s 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.

To print out the declaration of an identifier:

```
(dbx) whatis identifier
```

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

To print out the member function

```
(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();
Notice that table::pos is inherited from block:
(dbx) whatis table::pos
class point *block::pos();
```

To print out the data member:

```
(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` tells you 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 lookup the `this` pointer. In this example, the output from the `whatis` shows that the compiler automatically allocated this variable to a register.

```
(dbx) stop in brick::draw
(dbx) cont
// expose the blocks window (if exposed, hide then expose) to
force program to hit the breakpoint.
(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

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

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

To see inherited members the `whatis` command takes an option `-r` (for recursive) that displays the declaration of a specified class together with the members it inherits from parent 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. Note the inserted comment lines separating 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 exampleprotected:
    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);
};
```

Using the Auto-Read Facility

In general, you should compile the entire program you want to debug using the `-g` option. Depending on how the program was compiled, the debugging information generated for each program and shared library module is stored in either the object code file (`.o` file) for each program and shared library module, and/or the program executable file.

When you compile with the `-g -c` compiler option, debugging information for each module remains stored in its `.o` file. `dbx` then reads in debugging information for each module automatically, as it is needed, during a session. This read-on-demand facility is called *Auto-Read*. Auto-Read is the Default for `dbx`.

Auto-Read saves considerable time when loading a large program into `dbx`. Auto-Read depends on the continued presence of the program `.o` files in a location known to `dbx`.

Note – If you archive `.o` files into `.a` files, and then link using the archive libraries, you can then remove the associated `.o` files, but you must keep the `.a` files.

By default, `dbx` looks for files in the directory where they were when the program was compiled and the `.o` files in the location from which they were linked. If the files are not there, use the `pathmap` command to set the search path.

If no object file is produced, debugging information is stored in the executable. That is, for a compilation that does not produce `.o` files, the compiler stores all the debugging information in the executable and `dbx` does not use the Auto-Read facility.

Disabling Auto-Read with the -xs *Compiler Option*

Programs compiled with `-g -c` store debugging information for each module in the module's `.o` file. Auto-Read requires the continued presence of the program and shared library `.o` files.

In circumstances where it is not feasible to keep program `.o` files or shared library `.o` files for modules that you want to debug, compile the program using the compiler `-xs` option (use in addition to `-g`). (You can have some modules compiled with `-xs` and some without.) The `-xs` option instructs the compiler to have the linker place all of the debugging information in the program executable. Then when you load the program executable into `dbx`, all of the information loads at once. For a large program compiled with `-xs`, reading in the debugging information requires additional time; however, debugging is no longer dependent on the availability of the `.o` files.

Listing Modules

The `dbx modules` 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*:

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

To read in debugging information for all modules:

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

`-f`	Forces reading of debugging information, even if the file is newer than the executable
`-q`	Quiet mode

Command Reference

`modules`

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

```
modules -read
```

To list names of all program modules (with or without debugging info):

```
modules
```

To list all program modules with debugging info:

```
modules -debug
```

To print the name of the current module:

```
module
```

`whatis`

To print out non-type identifiers:

```
whatis [-n] [-r]
```

To print out type identifiers:

```
whatis -t [-r]
```

To print out expressions:

```
whatis -e [-r]
```

`list`

The default number of lines listed when no *number* is specified is controlled by the `dbxenv` variable `output_list_size`. Where appropriate, the line number may be $ (dollar sign) which denotes the last line of the file. A comma is optional.

To print a specific line number:

```
list number
```

To print the next *number* lines, or the previous number lines, use the plus or inus sign:

```
list [ + | - ] number
```

To list lines from one number to another:

```
list number1 number2
```

To list the start of the file *filename*:

```
list filename
```

To list the file *filename* from line *number*:

```
list filename:number
```

To list the start of the source for *function*:

```
list function
```

This command changes the current scope.

To intermix source lines and assembly code:

```
list -i
```

To list *number* lines, a window, around a line or function:

```
list -w number
```

This option is not allowed in combination with the + or - syntax or when two line numbers are specified.

3

Controlling Program Execution 4

This chapter describes how to run, attach to, continue, and rerun a program in `dbx`, and how to single-step through lines of program code.

This chapter is organized into the following sections:

Basic Concepts

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 later in this manual, you can control the run-time behavior of a program as it executes under `dbx`.

4

Running a Program in dbx

When you first load a program into dbx, dbx visits the program's "main" block (main for C, C++, and Fortran 90, MAIN for FORTRAN 77 and program for Pascal). dbx waits for you to issue further commands; you can visit code or use event management commands.

You may choose to set breakpoints in the program before running it. When ready, use the run command to start program execution.

To run a program in dbx without arguments, type:

```
(dbx) run
```

To run a program with command line arguments, type:

```
(dbx) run -o <my_input_file
```

To redirect a program, type:

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

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 by itself restarts the program using the previous arguments and redirection. The rerun command restarts the program and clears the original arguments and redirection.

Attaching dbx *to a Running Process*

You may need to debug a program that is already running. For example, you may want to debug a running server and you do not want to stop or kill it; or a program may be looping indefinitely and you would like to examine it under the control of dbx without killing it. You can *attach* dbx to a running program by using the program's *pid* number as an argument to the dbx start-up command.

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

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

To attach `dbx` to a program that is running independently of `dbx`:

1. **If** `dbx` **is already running, type:**

```
(dbx) debug program_name process_ID
```

   You can substitute a -(dash) for the *program_name*; `dbx` automatically finds the program associated with the pid and loads it.

2. **If** `dbx` **is not running, start** `dbx` **with the** *process_ID* **(**`pid`**) number as an argument:**

```
% dbx program_name process_ID
```

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

Detaching a Process from `dbx`

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`.

To detach a process from running under the control of `dbx`, use the `dbx detach` command:

```
(dbx) detach
```

4

Executing a Program

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

If the line executed contains a function call, next executes that call and returns from it ("steps over" the call). step stops at the first line in a called function.

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

You can specify the number of single steps. The dbx commands step and next accept a number argument that lets you specify an exact number of source lines to execute before stopping. The default is 1.

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) step n
```

pop pops the top frame off the stack and adjusts the frame pointer and the stack pointer accordingly. The pop command also changes the program counter to the beginning of the source line at the call site.

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:

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

Notice that while the parameters are optional, you must type in the parentheses after the *function_name*, for example:

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

If the source file in which the function is defined was compiled with the `-g` flag, 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.

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

For C++, `dbx` handles default arguments and function overloading. Automatic resolution of the C++ overloaded functions is done if possible. If any ambiguity remains (for example, functions not compiled with `-g`), `dbx` shows a list of the overloaded names.

When you use `call`, `dbx` behaves "`next`-like," returning from the called function. However, if the program hits a breakpoint in the called function, `dbx` stops the program at the breakpoint and emits the message. If you now issue 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 nested interpreters. You can then re-issue the command.

Continuing a Program

To continue a program, just use the `cont` command:

```
(dbx) cont
```

The `cont` command has a variant, `cont at` *line_number*, which allows you to 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, enter the `cont at` *line_number* command in the command pane:

```
(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 function.

A useful application of `cont at` is in conjunction with an `assign` command. Using `cont at` *line_number* with `assign`, you can avoid executing a line of code that contains a call to a function that may 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;
```

If you use this 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;}
```

Using Ctrl+C to Stop a Process

You can stop a process running in `dbx` using Ctrl+C (`^C`). When you stop a process using `^C`, `dbx` ignores the `^C`, but the child process sees 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 `cont`. 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.

Command Summary

`run`

Use the `run` command by itself to execute the program with the current arguments:

```
run
```

To begin executing the program with new arguments:

```
run args
```

`rerun`

To re-execute the program with no arguments:

```
rerun
```

`next`

The `next` command with no arguments steps one line stepping over calls:

```
next
```

To step *n* lines skipping over calls:

```
next n
```

To deliver the given signal while executing the `next` command:

```
next ... -sig sig
```

The `dbxenv` variable `step_events` controls whether breakpoints are enabled during a step.

To step the given thread:

```
next tid
```

To step the given LWP:

```
next lwpid
```

This will not implicitly resume all LWPs when skipping a function. When an explicit *tid* or *lwpid* is given, the deadlock avoidance measure of the generic `next` is defeated.

With multithreaded programs, when a function call is skipped over, all LWPs are implicitly resumed for the duration of that function call in order to avoid deadlock. Non-active threads cannot be stepped.

cont

Use the `cont` command to continue execution. In a multithreaded process, all threads are resumed.

```
cont
```

To continue execution at line *line*. *id* is optional.

```
cont line id
```

To continue execution with the signal *signo*:

```
cont ... -sig signo
```

To continue execution from a specific thread or LWP:

```
cont ... id
```

To continue execution and follow a forked process:

```
cont ... -follow parent|child
```

step

The `step` command with no arguments steps one line stepping *into* calls:

```
step
```

To step *n* lines stepping into calls:

```
step n
```

To step *n* lines stepping into calls and out of the current function:

```
step up n
```

To deliver the given signal while executing the `step` command:

```
step ... -sig sig
```

The `dbxenv` variable `step_events` controls whether breakpoints are enabled during a step.

To step the given thread (`step up` does not apply):

```
step ... tid
```

To step the given LWP:

```
step ... lwpid
```

This will not implicitly resume all LWPs when skipping a function. When an explicit *tid* or *lwpid* is given, the deadlock avoidance measure of the generic `step` is defeated.

With multithreaded programs, when a function call is skipped over, all LWPs are implicitly resumed for the duration of that function call in order to avoid deadlock. Non-active threads cannot be stepped.

debug

The `debug` command prints the name and arguments of the program being debugged.

```
debug
```

To begin debugging a *program* with no process or core:

```
debug program
```

To begin debugging a *program* with corefile *core*:

```
debug -c core program
-or-
debug program core
```

To begin debugging a *program* with process ID *pid*:

```
debug -p pid program
-or-
debug program pid
```

To force the loading of a corefile; even if it doesn't match:

```
debug -f ...
```

To retain all `display`, `trace`, `when`, and `stop` commands. If no -r option is given, an implicit delete all and undisplay 0 is performed.

```
debug -r
```

To start debugging a *program* even if the program name begins with a dash:

```
debug [options] -- prog
```

detach

The detach command detaches `dbx` from the target, and cancels any pending signals:

```
detach
```

To detach while forwarding the given signal:

```
detach -sig sig
```

4

Examining the Call Stack 5

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

This chapter is organized into the following sections:

Basic Concepts

The call stack represents all currently active routines—routines that have been called but have not yet returned to their respective caller.

Because the call stack grows from higher memory to lower memory, *up* means going up the memory to the caller's frame and *down* means going down the memory. The current program location—the routine executing when the program stopped at a breakpoint, after a single-step, or when it faults producing a core file—is in higher memory, while a caller routine, such as `main()`, is located lower in memory.

Finding Your Place on the Stack

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

```
where [-f] [-h] [-q] [-v] 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 a program crashes and produces a core file, you can load the core file into `dbx`.

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 you start from, *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 some *number* of frames up or down the stack from the current frame. If *number* is not specified, the default is one. 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. To move up the call stack (toward `main`) *number* levels:

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

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

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

Moving to a Specific Frame

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

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

The `frame` command without an argument prints the current frame number. With *number*, the command allows you to go directly to the frame indicated by the number. By including a + (plus sign) or - (minus sign), the command allows you to 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.

Command Reference

`where`

The where command shows the call stack for your current process. To print a procedure traceback:

```
where
```

To print the *number* top frames in the traceback:

```
where number
```

To start the traceback from frame *number*:

```
where -f number
```

To include hidden frames:

```
where -h
```

To print only function names:

```
where -q
```

To include function args and line info:

```
where -v
```

Any of the previous commands may be followed by a thread or LWP ID to view the call stack.

hide/unhide

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

To hide or delete all stack frames matching a regular expression *regexp:*

```
[hide | unhide] regexp
```

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.

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
```

Evaluating and Displaying Data 6

This chapter describes how to evaluate data, display the value of expressions, variables, and other data structures, and assign a value to an expression.

The chapter is organized into the following sections:

Basic Concepts

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

Evaluating Variables and Expressions

This section shows how 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.

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.
- Visit the function by changing the current function.
- Print the value of a variable or expression

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

To evaluate a variable or expression:

```
print expression
```

Evaluating Pascal Character Strings

To evaluate a Pascal character string in dbx, use the double quote ("") syntax to identify the string as a character string, as opposed to a character constant. This dbx convention also applies to passing parameters when calling a Pascal function.

To evaluate the Pascal character string, abc:

```
print "abc"
```

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 figure out what an object's type is.

You can use the commands `print` or `display` 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 which toggles the default behavior of the `dbxenv output_derived_type`.

Using the -d flag or setting the `dbxenv 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 baseclass to a derived class is not legal in C++).

Evaluating Unnamed Arguments in C++ Programs

C++ allows you to 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, dbx encodes unnamed arguments in a form that allows you to evaluate them. The form is:

```
_ARG_%n_
```

where dbx assigns an integer to %n.

To obtain an assigned argument name from dbx, issue the whatis command with the function name as its target:

```
(dbx) whatis tester
void tester(int _ARG_0_);
(dbx) whatis main
int main(int _ARG_1_, char **_ARG_2_);
```

To evaluate (or display) an unnamed function argument,

```
(dbx) print _ARG_1_
_ARG_1_ = 4
```

Dereferencing Pointers

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

To dereference a pointer, dbx prints 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:

```
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:

```
display
```

Turning Off Display (Undisplay)

`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:

```
undisplay expression
```

To turn off the display of all currently monitored variables:

```
undisplay 0
```

6

Assigning a Value to a Variable

To assign a value to a variable:

```
assign variable = expression
```

Evaluating Arrays

You evaluate arrays the same way you evaluate other types of variables. For more information on working with arrays in Fortran, see the chapter, *Debugging with Fortran* later in this manual.

Here is an example, using an array:

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

To evaluate the array:

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

Array Slicing for Arrays

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

- *Array Slicing*

  Prints any rectangular, *n*-dimensional box of a multi-dimensional 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).

Syntax for Array Slicing and Striding

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

Array-slicing syntax for C and C++, where:

```
print arr-exp [first-exp .. last-exp : stride-exp}
```

arr-exp	Expression that should evaluate to an array or pointer type
first-exp	First element to be printed. Defaults to 0.
last-exp	Last element to be printed. Defaults to its upper bound.
stride-exp	Stride. Defaults to 1.

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

```
(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..8:2] =
[2] = 2
[4] = 4
[6] = 6
```

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

```
(dbx) print arr(exp1:exp2:exp3)
```

exp1	start_of_slice
exp2	end_of_slice
exp3	length_of_stride (the number of elements skipped is *exp3* - 1)

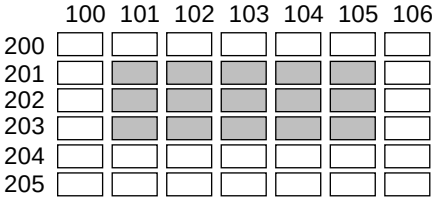
For an *n*-dimensional slice, separate the definition of each slice with a comma:

```
(dbx) print arr(exp1:exp2:exp3, exp1:exp2:exp3,...)
```

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 commands omit exp3, using the default stride value of 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 at the row `201` and ends with `203`. The second set of expressions, separated by a comma from the first, defines the slice for the 2nd dimension. The slice begins at column `101` and ends after column 105.

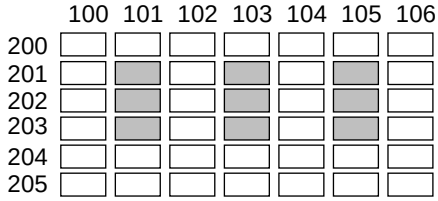
Strides

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, (*exp3*), specifies the length of the stride. The value of *exp3* specifies the elements to print; the number of elements skipped is equal to *exp3* -1. 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)
```



A stride of 2 prints every 2nd element, skipping every other element.

Shorthand Syntax

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:

`print arr`	Prints entire array, default boundaries.
`print arr(:)`	Prints entire array, default boundaries and default stride of 1.
`print arr(::`*exp3*`)`	Prints the whole array with a stride of *exp3*.

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:

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

Command Reference

`print`

To print the value of the expression(s) *expression:*

```
print expression, ...
```

In C++, to print the value of the expression *expression* including its inherited members*:*

```
print -r expression
```

In C++, to not print the expression *expression*'s inherited members when the `dbxenv output_inherited_members` is on*:*

```
print -r expression
```

In C++, to print the derived type of expression *expression* instead of the static type*:*

```
print -d [-r] expression
```

In C++, to print the static type of expression *expression* when the `dbxenv output_dynamic_type` is on*:*

```
print -d [-r] expression
```

To call the *prettyprint* function*:*

```
print -p expression
```

To not call the *prettyprint* function when the `dbxenv output_pretty_print` is on*:*

```
print -p expression
```

To not print the left hand side (the variable name or expression). If the expression is a string (char *), do not print the address, just print the raw characters of the string, without quotes*:*

```
print -l expression
```

To use *format* as the format for integers, strings, or floating point expressions*:*

```
print -f format expression
```

To use the given *format* without printing the left hand side (the variable name or expression)*:*

```
print -F format expression
```

To signal the end of flag arguments. Useful if *expression* starts with a plus or minus*:*

```
print -- expression
```

6

Setting Breakpoints and Traces 7

This chapter describes how to set, clear, and list breakpoints and traces, and how to use watchpoints.

The chapter is organized into the following sections:

Basic Concepts

The `stop`, `when`, and `trace` commands are called *event management* commands. Event management refers to the general capability of `dbx` to perform certain actions when certain events take place in the program being debugged.

Setting Breakpoints

There are three types of breakpoint action commands:

- `stop` **type 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`, `stop`, or `next`.
- `when` **type breakpoints —** the program halts and `dbx` executes one or more debugging commands, then the program continues (unless one of the commands is `stop`).
- `trace` **type breakpoints —** the program halts and an event-specific `trace` information line is emitted, then the program continues.

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

You can set a breakpoint at a line number, using the `dbx stop at` command:

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

where `n` is a source code line number and *filename* is an optional program filename qualifier. For example

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

If the line specified in a `stop` or `when` command is not an executable line of source code, `dbx` sets the breakpoint at the next executable line.

Setting a `when` *Breakpoint at a Line*

A `when` breakpoint command accepts other `dbx` commands like `list`, allowing you to write your own version of `trace`.

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

`when` operates with an implied `cont` command. In the example above, after listing the source code at the current line, the program continues executing.

Setting a Breakpoint in a Dynamically Linked Library

`dbx` provides full debugging support for code that makes use of 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()/dlclose()` allows you to 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.

There are three exceptions:

- You cannot set a breakpoint in a `dlopen`'ed library before that library is loaded by `dlopen()`.
- You cannot set a breakpoint in a `dlopen`'ed filter library until the first function in it is called.
- When a library is loaded by `dlopen()`, an initialization routine named `_init()` is called. This routine may call other routines in the library. `dbx` cannot place breakpoints in the loaded library until after this initialization is completed. You cannot have `dbx` stop at `_init()` in a library loaded by `dlopen`.

Setting Multiple Breaks in C++ Programs

You may want to 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 `when inmember`.

In the Blocks demo program, a member function `draw()`, is defined in each of five different classes: hand, brick, ball, wedge, table.

Setting Breakpoints in Member Functions of Same Class

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

To set a `when` breakpoint, use `when inclass`.

In the Blocks demo program, to set a breakpoint in all member functions of the class draw:

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

Breakpoints are inserted in only the class member functions defined in the class. It does not include those that it may inherit from base classes.

Due to the large number of breakpoints that may be inserted by `stop inclass` and other breakpoint selections, you should be sure to set your `dbxenv step_events` to `on` to speed up `step` and `next`.

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 in function` command.

To set a `when` breakpoint, use `when infunction`.

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

```
(dbx) when infunction sort {cmd;}
```

Tracing Code

Tracing displays information about the line of code about to be executed or a function about to be called.

Setting `trace` *Commands*

Set `trace` commands from the command line. The following table shows the command syntax for the types of traces that you can set. The information a `trace` provides depends on the type of *event* associated with it.

Command	`trace` **prints ...**
`trace step`	every line in the program as it is about to be executed.
`trace next -in` *function*	every line while the program is in the function
`trace at` *line_number*	the line number and the line itself, as that line becomes the next line to be executed.
`trace in` *function*	the name of the function that called *function*; line number, parameters passed in, and return value
`trace inmember` *member_function*	the name of the function that called *member_function* of any class; its line number, parameters passed in, and its return value
`trace inclass` *class*	the name of the function that called any member_function in *class*; its line number, parameters passed in, and return value
`trace infunction` *function*	the name of the function that called any member_function in *class*; its line number, parameters passed in, and return value
`trace change` *variable* `[-in` *function*`]`	the new value of *variable*, if it changes, and the line at which it changed

Controlling the Speed of a Trace

In many programs, code execution is too fast to view the code. The `dbxenv trace_speed` allows you to control the delay after each trace is printed. The default delay is `0.5` seconds.

To set the interval between execution of each line of code during a `trace`:

```
dbxenv trace_speed number
```

7

Listing and Clearing Event Handlers

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 print ID numbers in parenthesis, which can then be used by other commands.

As noted, 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 prints 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:

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

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

```
(dbx) delete all
```

Watchpoints

Watchpointing is the capability of dbx to note when the value of a variable or expression has changed.

Stopping Execution When Modified

To stop program execution when the contents of an address gets written to:

```
(dbx) stop modify &variable
```

Keep these points in mind when using `stop modify`:

- The event occurs when a variable gets written to even if it is the same value.
- The event occurs *before* the instruction that wrote to the variable is executed, although the new contents of the memory are preset by `dbx` by emulating the instruction.
- You cannot use addresses of stack variables, for example, auto function local variables.

Stopping Execution When Variables Change

To stop program execution if the value of a specified variable changes:

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

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

- `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.

  `dbx` implements `stop change` by causing automatic single stepping together with a check on the value at each step. Stepping skips over library calls. So, if control flows in the following manner:

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

dbx does not trace the nested *user_ routine2* because tracing skips the library call and the nested call to user_routine2, so 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 a block local “nested” variable, dbx issues an error informing you that it cannot perform this operation.

Stopping Execution on a Condition

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

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

The Faster modify *Event*

A faster way of setting watchpoints is to use the modify command. Instead of automatically single-stepping the program, it uses a page protection scheme which is much faster. The speed depends on how many times the page on which the variable you are watching is modified, as well as the overall system call rate of the program being debugged.

Setting Breakpoint Filters

In dbx, most of the event management commands also support an optional *event filter* modifier statement. 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. If the condition evaluates to false (0), dbx continues program execution as if the event never happened.

To set a breakpoint at a line or in a function that includes a conditional filter, 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.

Note – With location-based breakpoints 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 may have to use syntax to specify the scope precisely.

These two filters are *not* the same:

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

The former will breakpoint at `foo` and test the condition. The latter automatically single-steps and tests for the condition.

This point is emphasized because 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 modify &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 to see if the new value of *speed* is equal to `fast_enough`. If it is not, the program continues on, “ignoring” the `stop`.

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

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

7

Efficiency Considerations

Various events have varying 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, like 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`env `step_events` 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 watchpoints `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 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 in the cases where you suspect that the `lookup()` function is clobbering your variable.

Event Management 8

Event management refers to the capability of dbx to perform actions when events take place in the program being debugged.

This chapter is organized into the following sections:

Basic Concepts

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 need to be specified.

An example of the association of a program event with a dbx action is setting a breakpoint on a particular line. A change in the value of a variable triggers a stop.

The most generic form of creating a handler is through the when command:

```
when event-specification {action; ... }
```

Although all event management can be performed through when, dbx has historically had many other commands, which are still retained, either for backward compatibility, or because they are simpler and easier to use.

In many places examples are given on how a command (like stop, step, or ignore) can be written 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

The commands when, stop, and trace are used to create event handlers. An *event-spec* is a specification of an event as documented later in this chapter.

Every command returns a number known as a handler id (*hid*). This number can be accessed via the predefined variable $newhandlerid.

An attempt has been made to make the stop and when commands conform to the handler model. However, backward compatibility with previous dbx releases forces some deviations.

For example, the following samples from an earlier dbx release are equivalent to:

when *cond body*	when step -if *cond body*
when *cond* in *func body*	when next -if *cond* -in *func body*

These examples illustrate that cond is not a pure event; there is no internal handler for conditions.

when

The when command has a general syntax. When the specified event occurs, the *cmds* are executed. Once the commands have all executed, the process is automatically continued.

```
when event-specification [ modifier ] { cmds ... ; }
```

stop

The stop command has a general syntax. When the specified event occurs, the process is stopped.

```
 stop event-specification [ modifier ]
```

stop is shorthand for a common when idiom:

```
when event-specification { stop -update; whereami; }
```

trace

When the specified event occurs, a trace message is printed:

```
trace event-specification
```

Most of the trace commands can be hand-crafted by using the when command, ksh functionality, and event variables. This is especially useful if you want stylized tracing output.

Manipulating Event Handlers

The following list contains commands to manipulate event handlers. For more information on any of the commands, see the section *Command Reference* at the end of this chapter.

- status - lists handlers

- `delete` - deletes all handlers including temporary handlers
- `clear` - deletes handlers based on breakpoint position.
- `handler -enable` - enables handlers
- `handler -disable` - disables handlers

Using Event Counters

Event handlers have trip counters. There is a count limit and the actual counter. Whenever the event occurs, the counter is incremented. The action associated with the handler executes 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.

The count limit can be set using the `-count` modifier. Otherwise, use the handler to individually manipulate event handlers.

```
handler [ -count | -reset ] hid new-count new-count-limit
```

Setting Event Specifications

Event specifiers are used by the `stop, when` and `trace` commands to denote event types and parameters. The format is that of a keyword to represent the event type and optional parameters.

Event Specifications

The following are event specifications, syntax and descriptions.

`in` *func*

The function has been entered and the first line is about to be executed. If the `-instr` modifier is used, it is the first instruction of the function about to be executed. (Do not confuse `in` *`func`* with the `-in` *func* modifier.) The *func* specification can take a formal parameter signature to help with overloaded function names, or template instance specification. For example, you can say:

```
stop in mumble(int, float, struct Node *)
```

`at` *lineno*

The designated line is about to be executed. If *filename* is provided, then the designated line in the specified file is about to be executed. The filename can either be the `.o`, `.c`, or `.cc` name of the file. Although quote marks are not required, they may be necessary if the filename contains odd characters.

```
at filename:lineno
```

`infunction` *func*

Equivalent to `in` *func* for all overloaded functions named *func*, or all template instantiations thereof.

`inmember` *func*

`inmethod` *func*

Equivalent to `in` *func* for the member function named *func* for every class.

`inclass classname`

Equivalent to `in` *func* for all member functions that are members of *classname*.

`inobject` *obj-expr*

A member function called on the object denoted by *`obj-expr`* has been called.

`change` *variable*

Fires when the value of *variable* has changed.

`cond` *cond-expr*

The condition denoted by *cond-expr* evaluates to true. Any expression can be used for *cond-expr*, but it has to evaluate to an integral type.

`returns`

This event is just a breakpoint at the return point of the current *visited* function. The visited function is used so that you can use the `returns` event spec after doing a number of `up`'s. The plain return event is always `-temp` and can only be created in the presence of a live process.

`returns` *func*

This event fires 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 it through:

Sparc	`$o0`
Intel	`$eax`
Power PC	`$r3`

It is another way of saying:

```
when in func { stop returns; }
```

step

The step event fires when execution reaches the first instruction of a source line. For example, you can get simple tracing with:

```
when step { echo $lineno: $line; }
```

When enabling a step event you instruct dbx to single-step automatically next time cont is used. The step *command* can be implemented as follows:

```
alias step="when step -temp { whereami; stop; }; cont"
```

next

Similar to step except that functions are not stepped into.

timer *second*

Fires when the debugee has been running for *seconds*. The timer used with this is shared with the collector command.

sig *sig*

When the signal is first delivered to the debugee, this event fires. sig can either be a decimal number or the signal name in upper or lower case; the prefix is optional. This is completely independent of the catch/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'nt seen it yet. Only if you `cont` the process with the given signal is the signal forwarded to it.

`sig` *sig sub-code*

When the specified signal with the specified sub-code is first delivered to the child, this event fires. Just as with signals, the sub-code can be entered as a decimal number, in capital or lower case; the prefix is optional.

`fault` *fault*

This event fires when the specified fault occurs. The faults are architecture dependent, but a set of them is known to `dbx` as defined by `proc` (4):

`FLTILL`	Illegal instruction
`FLTPRIV`	Privileged instruction
`FLTBPT`	Breakpoint instruction
`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

These faults are taken from `/sys/fault.h`. *fault* can be any of those listed above, in upper or lower case, with or without the `FLT-` prefix, or the actual numerical code. Be aware that `BPT`, `TRACE`, and `BOUNDS` are used by `dbx` to implement breakpoints, single-stepping, and watchpoints. Handling them may interfere with the inner workings of `dbx`.

modify addr-exp [, byte-size]

The specified address range has been modified. This is the general watchpoint facility. The syntax of the event-specification for watchpoints is:

```
modify addr-exp [, byte-size-exp ]
```

addr-exp is any expression that can be evaluated to produce an address. If a symbolic expression is used, the size of the region to be watched is automatically deduced, or you can override that with the `,' syntax. You can also use nonsymbolic, typeless address expressions; in which case, the size is mandatory. For example:

```
stop modify 0x5678, sizeof(Complex)
```

Limitations of modify *event-spec*

Addresses on the stack cannot be watched.

The event is not fired if the address being watched is modified by a system call.

Shared memory (MAP_SHARED) cannot be watched, because dbx cannot catch the other processes stores into shared memory. Also, dbx cannot properly deal with SPARC swap and ldstub instructions.

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 procfs(4). These are 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.

sysout code|name

The specified system call is finished and the process is about to return to user mode.

sysin | sysout

Without arguments, all system calls are traced. Note that certain `dbx` features, for example `modify` event and RTC, cause the child to execute system calls for their own purposes and show up if traced.

prog_new

Fired 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. Whenever the process stops such that the user gets a prompt, particularly in response to a `stop` handler, this event is fired. For example, the following are equivalent:

```
display x
when stop {print x;}
```

sync

The process being debugged has just been `exec`()'ed. All memory specified in `a.out` is valid and present but pre-loaded shared libraries have not been loaded yet. For example, `printf`, although known to `dbx`, has not been mapped into memory yet.

A `stop` on this event is ineffective; however, you can use this event with the `when` command.

syncrtld

This event is fired after a `sync` (or `attach` if the process being debugged has not yet processed shared libraries). It fires after the startup code has executed and the symbol tables of all preloaded shared libraries have been loaded.

A `stop` on this event is ineffective; however, you can use this event with the `when` command.

attach

`dbx` has successfully attached to a process.

detach

The debugee has been detached from.

lwp_exit

Fired when `lwp` has been exited. `$lwp` contains the id of the exited LWP.

proc_gone

Fired when `dbx` is no longer associated with a debugged process. The predefined variable `$reason` will be `signal`, `exit`, `kill`, or `detach`.

lastrites

The process being debugged is about to expire. There are only three reasons this can happen:

- 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.

`dlopen [ lib-path ] | dlclose [ lib-path ]`

These events are fired 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 `$dllist`. The first word in `$dllist` is actually a “+” or a “-”, indicating whether the list of libraries is being added or deleted.

lib-path is the name of a shared library you are interested in. If it is specified, the event only fires 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 fire whenever there is any `dl`-activity. `$dlobj` is empty but `$dllist` is valid.

Event Specification Modifiers

An event specification modifier sets additional attributes of a handler, the most common kind being event filters. Modifiers have to appear after the keyword portion of an event spec. They all begin with a hyphen (`-`), preceded by blanks. Modifiers consist of the following:

`-if` *cond*

The condition is evaluated when the event specified by the *event-spec* occurs. The event is fired only if the condition evaluates to nonzero. The condition is sometimes called a filter. A handler created with a filter is known as a filtered handler.

If `-if` is used with a location-based event, like `in` or `at,` *`cond`* is evaluated in the scope corresponding to that location, otherwise it should be properly qualified with the desired scope.

`-in` *func*

The handler is enabled only while within the given function, or any function called from func. The number of times the function is entered is reference counted so as to properly deal with recursion. The handler that has been modified by the `-in` modifier is said to be "bounded by *func*."

-disable

Create the handler in the disabled state.

--count *n* -count infinity

Have the handler count from 0. 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.

-temp

Create a temporary handler. Once the event is fired 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 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 *tid*

The event is fired only if the thread that caused it matches *tid*.

--lwp *lid*

The event is fired only if the thread that caused it matches *lid*.

-hidden

Makes the handler not show up in a regular `status` command. Use `status -h` to see hidden handlers.

-perm

Normally all handlers get thrown away when a new program is loaded. Using this modifier causes the handler to be retained across debuggings. A plain `delete` command will not delete a permanent handler. Use `delete -p` to delete a permanent handler.

Parsing and Ambiguity

Syntax for event-specs and modifiers is:

- Keyword driven
- Based on `ksh` conventions; everything is split into words delimited by spaces

Since expressions can have spaces embedded in them, this can cause 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-spec* in favor of `-temp` being a modifier. In the second example, `a-temp` is collectively passed to a language specific expression parser and 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 are always valid:

Variable	Definition
`$pc`	Current program counter address (hexadecimal)
`$ins`	Disassembly of the current instruction
`$lineno`	Current line number in decimal
`$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`
`$dllist`	After `dlopen` or `dlclose` event, contains the list of load objects just dlopened or dlclosed. The first word of `dllist` is actually a “+” or a “-” depending on whether a `dlopen` or a `dlclose` has occurred.
`$newhandlerid`	ID of the most recently created handler
`$proc`	Process id of the current process being debugged
`$lwp`	Lwp id of the current LWP
`$thread`	Thread id of the current thread

Variable	Definition
`$prog`	Full pathname of the program being debugged
`$oprog`	Old, or original value of `$prog`. This is very handy for getting back to what you were debugging following an `exec()`.
`$exitcode`	Exit status from the last run of the program. The value is an empty string if the process hasn't actually exited.

As an example, consider that `whereami` can be roughly implemented as:

```
function whereami {
  echo Stopped in $func at line $lineno in file $(basename $file)
  echo "$lineno\t$line"
}
```

Event-Specific Variables

The following event-specific variables are only valid within the body of a `when`.

`$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`

Is set to `true` if the event occurs during the *boot* process. Whenever a new program is debugged, it is first booted so that the list and location of shared libraries can be ascertained. The process is then killed.

While booting happens, 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`.

Valid Variables

For Event `sig`

`$sig`	Signal number that caused the event
`$sigstr`	Name of $sig
`$sigcode`	Subcode of $sig if applicable
`$sigcodestr`	Name of $sigcode
`$sigsender`	Process id of sender of the signal, if appropriate

For Event `exit`

`$exitcode`	Value of the argument passed to _exit(2) or exit(3) or the return value of main

For Events `dlopen` *and* `dlclose`

`$dlobj`	Pathname of the load object dlopened or dlclosed

For Events `sysin` ***and*** `sysout`

`$syscode`	System call number
`$sysname`	System call name

For Event `proc_gone`

`$reason`	One of signal, exit, kill, or detach

Examples

Use these examples for setting event handlers.

Set Watchpoint for Store to Array Member

To set a watchpoint on `array`[99]:

```
(dbx) stop modify &array[99]
(2) stop modify &array[99], 4
(dbx) run
Running: watch.x2
watchpoint array[99] (0x2ca88[4]) at line 22 in file "watch.c"
    22  array[i] = i;
```

Simple Trace

To implement a simple trace:

```
(dbx) when step { echo at line $lineno; }
```

Enable Handler While Within the Given Function (`in` func)

For example:

```
trace step -in foo
```

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"; };
    }
```

Determine the Number of Lines Executed in a Program

To see how many lines were executed in a small program:

```
(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. 133 is the number of lines executed. This process is very slow though. This technique is more useful with breakpoints on functions that are called many times.

Determine the Number of Instructions Executed by a Source Line

To count how many instructions a line of code executes:

```
(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, you end up counting those as well. You can use the `next` event instead of `step` to count instructions, excluding called functions.

Enable Breakpoint after Event Occurs

Enable a breakpoint only after another event has fired. Suppose things go bad in function `hash`, but only after the 1300'th symbol lookup:

```
(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.

Set Automatic Breakpoints for `dlopen` *Objects*

To have breakpoints in dlopened objects be managed automatically:

```
(dbx) when dlopen mylib.so {  # delete old one if it exists.
    if [ -n "$B1" ]
    then delete "$B1"
    fi                            # create new one
    stop in func
    B1="$newhandlerid"
}
```

Reset 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 for you 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
```

Check Program Status

To see quickly where the program is while it's running:

```
(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 of course). 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:

```
(dbx)ignore FPE               # turn off default handler
(dbx) help signals | grep FPE # can't remember the subcode name
...
(dbx) stop sig fpe FPE_FLTUND
...
```

Command Reference

when

To execute *command*(s) when *line* is reached

```
when at line { command(s); }
```

To execute *command(s)* when *proc* is called:

```
when in proc { command(s); }
```

stop

To stop execution now and update displays. Whereas normally the process is continued after the body has executed, the `stop` command prevents that. This form is only valid within the body of a `when`:

```
stop -update
```

Same as above, but does not update displays:

```
stop -noupdate
```

To stop execution at line *line*:

```
stop at line
```

To stop execution when function *func* is called

```
stop in func
```

To set breakpoints on all member functions of a class / struct / union or template class:

```
stop inclass classname
```

To set breakpoints on all member functions:

```
stop inmember name
```

To set breakpoints on all non-member functions:

```
stop infunction name
```

step

The `step` command is equivalent to:

```
when step -temp { stop; }; cont
```

The `step` command can take a `sig` argument. A `step` by itself cancels the current signal just like `cont` does. To forward the signal, you must explicitly give the signal. You can use the variable `$sig` to step the program sending it the current signal:

```
step -sig $sig
```

cancel

Only valid within the body of `when`. The `cancel` command cancels any signal that might have been delivered, and lets the process continue. For example:

```
when sig SIGINT { echo signal info;  cancel; }
```

status

The `status` command lists handlers, those created by `trace`, `when`, and `stop`. `status` *hid* lists the given handler. If the handler is disabled, its *hid* is printed inside square brackets [ ] instead of parentheses ( ).

```
status [-s] -h hid
```

The output of `status` can be redirected to a file. Nominally, the format is unchanged during redirection. You can use the `-s` flag to produce output that allows a handler to be reinstated using the `source` command. If the `-h` option is used, *hidden* handlers are also listed.

The original technique of redirecting handlers using `status` and `sourc`ing the file was a way to compensate for the lack of handler enabling and disabling functionality.

delete

`delete` *hid* deletes the specified handler; `delete all` or `delete` *0* (zero), and `delete -all` deletes all handlers including temporary handlers. `delete -temp` deletes only all temporary handlers.

```
delete all -all -temp hid [hid ... ]
```

`-h` *hid* is required if *hid* is a hidden handler. `-h all` deletes all hidden handlers as well.

clear

`clear` with no arguments deletes all handlers based on breakpoints at the location where the process stopped. `clear` *line* deletes all handlers based on breakpoints on the given line.

```
clear line
```

handler

`handler -disable` *hid* disables the specified event; `handler -disable all` disables all handlers. `handler -enable` *hid* enables the specified handler; `handler -enable all` enables all handlers.

```
handler [ -disable | -enable ] all hid [...]
```

`handler -count` *hid* returns the count of the event in the form *current-count*/*limit* (same as printed by `status`). *limit* might be the keyword `infinity`. Use the ksh modifiers `${#}` and `${##}` to split the printed value.

```
handler [ -count | -reset ] hid new-count-limit
```

`handler -count` *hid new-count* assigns a new count limit to the given handler. `handler -reset` *hid* resets the count of the handler to 0 (zero).

Using Runtime Checking 9

Note – Access checking is available only on SPARC systems.

Runtime checking (RTC) enables you to automatically detect runtime errors in an application during the development phase. RTC lets you detect runtime errors such as memory access errors and memory leak errors and monitor memory usage.

The following topics are covered in this chapter:

9

Basic Concepts

Because RTC is an integral debugging feature, all debugging functions such as setting breakpoints and examining variables can be used with RTC, except the Collector.

The following list briefly describes the capabilities of RTC:

- Detects memory access errors
- Detects memory leaks
- Collects data on memory use
- Works with all languages
- Works on code that you do not have the source for, such as system libraries
- Works with multithreaded code
- Requires no recompiling, relinking, or makefile changes

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

For more detailed information on any aspect of RTC, see the online help.

When to Use RTC

One way to avoid seeing a large number of errors at once is to use RTC 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 RTC 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 RTC again only when you make changes to a module.

Requirements

To use RTC, you must fulfill the following requirements.

- Programs compiled using a Sun vompiler
- Dynamic linking with `libc`

- Use of the standard `libc` `malloc/free/realloc` functions or allocators based on those functions

  RTC does provide an API to handle other allocators; see *Using Fix & Continue With RTC* on page 121.

- Programs that are not fully stripped; programs stripped with `strip -x` are acceptable.

Limitations

RTC does not handle program text areas and data areas larger than 8 megabytes.

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 RTC

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

To turn on the desired checking mode, start `dbx` with the `-C` option:

```
% dbx -C program_name
```

The `-C` flag forces early loading of the RTC library. When you start `dbx` without the `-C` option and then enable checking, the RTC library is loaded when you issue the next `run` command; that may cause reloading of the shared libraries needed by the program. Using the `-C` flag initially allows you to avoid reloading.

Note – You must turn on the type of checking you want before you run the program.

To turn on memory use and memory leak checking:

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

To turn on memory access checking only:

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

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

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

To turn off RTC entirely:

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

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 `dbxenv` variable `rtc_auto_continue` is set to `on`). You can then issue `dbx` commands, such as `where` to get the current stacktrace 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.

If `rtc_auto_continue` *is* set to `on`, RTC continues to find errors, and keeps running automatically. It redirects errors to the value of the `dbxenv` variable `rtc_error_log_file_name`.

You can limit the reporting of RTC errors using the `suppress` command. The program continues to the next error or breakpoint, whichever is detected first.

You can perform any of the usual debugging activities, such as setting breakpoints and examining variables. The `cont` command runs the program until another error or breakpoint is encountered or until the program terminates.

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 symbolic information for hello
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
(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. RTC reports this access error as a Read from uninitialized (`rui`).

Function `memory_leak`() does not `free`() the variable `local` before it returns. When `memory_leak`() returns this variable, it 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 (SPARC only)

RTC checks whether your program accesses memory correctly by monitoring each read, write, and memory free operation.

Programs may 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 because a function returned a pointer to a local variable. Access errors may 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.

RTC maintains a table that tracks the state of each block of memory being used by the program. RTC checks each memory operation against the state of the block of memory it involvesand 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.

9

Using RTC 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 sort of chain reaction. The first error in the chain is therefore the "first cause," and fixing that error may 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

RTC prints the following information for memory access errors:

type	Type of error.
access	Type of access attempted (read or write).
size	Address of attempted access.
addr	Size 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 is 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, RTC provides *function* and *address*.

The following example shows a typical access error:

```
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff67c
    which is 1268 bytes above the current stack pointer
Location of error: Basic.c, line 56,
     read_uninited_memory()
```

Memory Access Errors

RTC detects the following memory access errors:

- Read from uninitialized memory (`rui`)
- Read from unallocated memory (`rua`)
- Write to unallocated memory (`wua`)
- Write to read-only memory (`wro`)
- Misaligned read (`mar`)
- Misaligned write (`maw`)
- Duplicate free (`duf`)
- Bad free (`baf`)
- Misaligned free (`maf`)
- Out of memory (`oom`)

For a full explanation of each error and an example, see *RTC Errors* later in this chapter.

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 even reference them, much less free them. RTC finds and reports such blocks.

Memory leaks result in increased virtual memory consumption and generally result in memory fragmentation. This may 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.*/
}
```

Memory leaks can be avoided by following a good programming practice of 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 anyway. RTC does not report a block as a leak if the program still retains one or more pointers to it.

Detecting Memory Leak Errors

Note – RTC only finds leaks of `malloc` memory. If your program does not use `malloc`, RTC cannot find memory leaks.

RTC detects the following memory leak errors:

- Memory Leak (`mel`)
- Possible leak — Address in Register (`air`)
- Possible leak — Address in Block (`aib`)

For a full explanation of each error and an example, see *RTC Errors* later in this chapter.

Possible Leaks

There are two cases where RTC may report a “possible” leak. The first case is when no pointers were found pointing to the beginning of the block, but a pointer was 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 happened to point 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 RTC cannot distinguish these two cases, it reports them as possible leaks, allowing the user to make the determination.

The second type of possible leak occurs when no pointers to a block were found in the data space, but a pointer was found in a register. This case is reported as an “Address in Register (`air`)” error. If the register happens to point 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. In such cases, this 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.

Note – RTC leak checking requires use of the standard `libc` `malloc/free/realloc` functions or allocators based on those functions. For other allocators, see *Using Fix & Continue With RTC* on page 121.

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
```

Clicking on the call stack location hypertext link takes you to that line of the source code in the editor window.

UNIX programs have 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 simply 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 globals 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 RTC 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 RTC `showleaks` command to report all the true leaks, omitting the leaks that would result merely from variables in `main` going out of scope.

Understanding the Memory Leak Report

With leak checking turned on, you get an automatic leak report when the program exits. All possible leaks are reported—provided the program has not been killed using the `kill` command. By default, a non-verbose leak report is generated, which is controlled by the `dbxenv` variable `rtc_mel_at_exit`.

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:

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`

The non-verbose report capsulizes the error information into a table, while the verbose report gives you a separate error message for each error. They both contain a hypertext link to the location of the error in the source code.

Here is the corresponding non-verbose 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.

Combining Leaks

Because the number of individual leaks can be very large, RTC 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, RTC 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, there are some general guidelines for fixing the memory leaks. The most important thing is to 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. `showleaks` by default gives you only the new leaks created since the last `showleaks` command. You can run `showleaks` repeatedly to narrow the window where the block was leaked.

9

Using Memory Use 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 to the blocks, the program cannot even reference them, much less free them. RTC finds and reports such blocks.

Memory leaks result in increased virtual memory consumption and generally result in memory fragmentation. This may slow down the performance of your program and the whole system.

RTC lets you see all the heap memory in use. You can use this information to get a sense of where memory gets 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 may help in performance tuning

RTC 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 by a command that causes memory usage to be displayed.

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 non-verbose blocks in use report is generated at program exit, which is controlled by the `dbxenv` variable `rtc_biu_at_exit`.

The following is a typical non-verbose 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

RTC 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.

Error suppression is done using the `suppress` command. Suppression can be undone using the `unsuppress` command. Suppression is persistent across `run` commands within the same debug session, but not across `debug` commands.

The following kinds 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:

  Global — the default; applies to the whole program

  Load Object — applies to an entire loadobject, such as a shared library

  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, RTC suppresses the most recent error to prevent repeated reports of the same error. The is controlled by the `dbxenv` variable `rtc_auto_suppress`.

- Others

  You can use the `dbxenv` variable `rtc_error_limit` to limit the number of errors that will be reported.

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 `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
```

Default Suppressions

To detect all errors RTC 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`. 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 by using the `-d` option of the `suppress` and `unsuppress` commands.

The following command causes RTC to no longer suppress read from uninitialized memory (`rui`) in code that does not have symbolic information (compiled without `-g`):

```
unsuppress -d rui
```

Using Suppression to Manage Errors

For the initial run on a large program, the number of errors may be so large as to be overwhelming. In this case, it may be better to take a phased approach. This can be done 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 errors (although they can cause more serious errors to happen later), and often a program may 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 of some sort.

You could 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 there are no errors left.

If you want to suppress the last reported error, use `suppress -last`. You also can limit the number of errors reported without using the `suppress` command by using `dbxenv` variable `rtc_error_limit` *n* instead.

Using RTC on a Child Process

`dbx` supports Runtime Checking of a child process if RTC is enabled for the parent and the `dbxenv` variable `follow_fork_mode` is set to `child`. When a fork happens, `dbx` automatically performs RTC on the child. If the program does an exec(), the RTC settings of the program calling `exec()` are passed on to the program.

At any given time, just one process can be under RTC control. 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 }
%
```

RTC reports first error in the parent, program1

```
% 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
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;
```

Because follow_fork_mode is set to child, when the fork occurs error checking is switched from the parent to the child process	(dbx) **cont** dbx: warning: Fork occurred; error checking disabled in parent detaching from process 3885 Attached to process 3886 stopped in _fork at 0xef6b6040 0xef6b6040: _fork+0x0008:bgeu _fork+0x30 Current function is main 13 child_pid = fork(); parent: child's pid = 3886
RTC reports an error in the child	(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 Variable is 'child_j' Current function is main 22 child_i = child_j;
When the exec of program2 occurs, the RTC settings are inherited by program2 so access and memory use checking are enabled for that process	(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 Enabling Error Checking... done stopped in main at line 8 in file "program2.c" 8 printf ("program2: pid = %d\n", getpid());
RTC reports an access error in the exec'd program, program2	(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 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 RTC on an Attached Process

RTC works with attached processes as well. However, to use RTC on an attached process, that process must be started with `librtc.so` preloaded. `librtc.so` resides in the `lib` directory of the product (`../lib` from the path of `dbx`; if the product is installed in `/opt`, it is `/opt/SUNWspro/lib/librtc.so`).

To preload `librtc.so`:

```
% setenv LD_PRELOAD path-to-librtc/librtc.so
```

It is a good idea to set to preload `librtc` only when needed (as with `attach`); do not have it on all the time. For example:

```
% setenv LD_PRELOAD...
% start-your-application
% unsetenv LD_PRELOAD
```

Once you attach to the process, you can enable RTC.

In case the program you want to attach to gets forked or executed from some other program, you need to set `LD_PRELOAD` for the main program (which will fork). The setting of `LD_PRELOAD` is inherited across fork/exec.

Using Fix & Continue With RTC

You can use RTC along with Fix & Continue to rapidly isolate and fix programming errors. Fix & 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.
```

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. This is so that RTC can keep track of all the allocations and deallocations in the program.

Many applications write their own memory management routines either on top of `malloc`-free or from scratch. When you use your own allocators (referred to as *private allocators*), RTC cannot automatically track them, thus you do not learn of leak and memory access errors resulting from their improper use.

However, RTC provides an API for the use of private allocators. This API allows the private allocators to get the same treatment as the standard heap allocators. The API itself is provided in a header file `rtc_api.h` and is distributed as a part of WorkShop. The man page `rtc_api(3x)` details the RTC API entry points.

Some minor differences may exist with RTC access error reporting when private allocators do not use the program heap. The error report will not include the allocation item.

Using RTC in Batch Mode

`bcheck(1)` is a convenient batch interface to the RTC feature of `dbx`. It runs a program under `dbx` and by default places the RTC error output in the default file `program.errs`.

`bcheck` can perform memory leak checking, memory access checking, memory use checking, or all three. Its default action is to perform only leak checking. Refer to the `bcheck` (1) man page for more details on its use.

The syntax for `bcheck` is:

```
bcheck [-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 `bcheck`.

The `-q` option makes `bcheck` completely quiet, returning with the same status as the program. This is useful when you want to use `bcheck` in scripts or makefiles.

Perform only leak checking on `hello`:

```
bcheck hello
```

Perform only access checking on `mach` with the argument `5`:

```
bcheck -access mach 5
```

Perform memory use checking on `cc` quietly and exit with normal exit status:

```
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*. But 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 `dbxenv` 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.

You can also enable a batch-like mode directly from `dbx` by setting the following `dbxenv` variables:

```
(dbx) dbxenv rtc_auto_continue on
(dbx) dbxenv rtc_error_log_file_name logfile
```

With these settings, the program does not stop when runtime errors are detected, and all error output is redirected to your error log file.

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 may occur if you are using RTC on an attached process and have set LD_PRELOAD to a version of librtc.so other than the one shipped with your WorkShop dbx image. To fix this, change the setting of LD_PRELOAD.

```
patch area too far (8mb limitation); Access checking disabled
```

RTC was unable to find patch space close enough to a load object for access checking to be enabled. See *rtc_patch_area* later in this chapter.

RTC's 8 Megabyte Limit

When access checking, dbx replaces each load and store instruction with a branch instruction that branches to a patch area. This branch instruction has an 8 megabytes range. This means that if the debugged program has used up all the address space within 8 megabytes of the particular load/store instruction being replaced, there is no place to put the patch area.

If RTC can't 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.

Working Around the 8 Megabyte Limit

dbx provides some possible workarounds to users who have run into this limit. These workarounds require the use of a utility called rtc_patch_area which is included with WorkShop.

9

This utility creates object files or shared object files that can be linked into your program to create patch areas for RTC to use.

There are two situations that can prevent `dbx` from finding patch areas within 8 megabytes of all the loads and stores in an executable image:

Case 1: The statically linked `a.out` file is too large.

Case 2: One or more dynamically linked shared libraries is too large.

When `dbx` runs into this limitation, it prints a message telling how much patch space it needs and directs you to the appropriate case (Case 1 or Case 2).

Case 1

In Case 1, you can use `rtc_patch_area` to make one or more object files to serve as patch areas and link them into the a.out.

After you have seen a message like the following:

```
Enabling Error Checking... dbx: warning: rtc: cannot find patch
space within 8Mb (need 6490432 bytes for ./a.out)
dbx: patch area too far (8Mb limitation); Access checking
disabled
         (See `help rtc8M', case 1)
```

1. **Create an object file `patch.o` for a patch area with a size less than or equal to 8 megabytes:**

```
rtc_patch_area -o patch.o -size 6490432
```

The `-size` flag is optional; the default value is 8000000.

2. **If the size request from the error message is satisfied, continue to the next step. Otherwise, repeat step 1 and create more .o files as needed.**

3. **Relink the `a.out`, adding the `patch.o` files to the link line.**

4. **Try RTC again with the new binary.**
   If RTC still fails, you may try to reposition the `patch.o` files on the link line.

An alternate workaround is to divide the `a.out` into a smaller `a.out` and shared libraries.

Case 2

If you are running RTC on an attached process, see case 2a. Otherwise, the only workaround is to rebuild the shared library with extra patch space:

After you have seen a message like the following::

```
Enabling Error Checking... dbx: warning: rtc: cannot find patch
space within 8Mb (need 563332 bytes for ./sh1.so)
dbx: patch area too far (8Mb limitation); Access checking
disabled
         (See `help rtc8M', case 2)
```

1. **Create an object file** `patch.o` **for a patch area with a size less than or equal to 8 megabytes:**

```
rtc_patch_area -o patch.o -size 563332
```

The -size flag is optional; the default value is 8000000.

2. **If the size request from the error message is satisfied, continue to the next step. Otherwise, repeat step 1 and create more .o files as needed.**
3. **Relink** `sh1.so`**, adding the** `patch.o` **files to the link line.**
4. **Try RTC again with the new binary; if** `dbx` **requests patch space for another shared library, repeat steps 1-3 for that library.**

Case 2a

In Case 2a, if the shared library patch space requested by `dbx` is 8 megabytes or less, you can use `rtc_patch_area` to make a shared library to serve as a patch area and link it into the a.out.

After you have seen a message like the following:

```
Enabling Error Checking... dbx: warning: rtc: cannot find patch
space within 8Mb (need 563332 bytes for ./sh1.so)
dbx: patch area too far (8Mb limitation); Access checking
disabled
         (See `help rtc8M', case 2)
```

1. **Create a shared object** `patch1.so` **for a patch area:**

```
rtc_patch_area -so patch1.so -size 563332
```

The `-size` flag is optional; the default value is 8000000.

2. **2) Relink the program with** `patch1.so` **placed adjacent to** `sh1.so` **in the link line. If** `dbx` **still requests patch space for the same** `sh1.so`**, then try placing** `patch1.so` **immediately before** `sh1.so`**.**
   It may be necessary to use full pathnames instead of the `ld -l` option to get the desired shared library ordering.

3. **Try RTC again with the new binary; if** `dbx` **requests patch space for another shared library, repeat steps 1-2 for that library.**
   If the patch space requested by `dbx` is more than 8 megabytes for a given shared library, follow the steps in case 2 above.

rtc_patch_area

`rtc_patch_area` is a shell script that creates object files or shared library files that can be linked into the user's program to add patch area space to programs with large text, data, or bss images.

The object file (or shared library) created contains one RTC patch area of the specified size or 8000000 if size is not supplied.

The name of the resulting object file (or shared library) is written to the standard output. Either the `-o` or `-so` option must be used.

Specify the name of the shared library to be created. This name is then written to the standard output:

```
-so sharedlibname
```

Specify the name of the object file to be created. This name is then written to the standard output:

```
-o objectname
```

Create a patch area of *size* bytes (default and reasonable maximum is 8000000):

```
-size size
```

Use *compiler* instead of `cc` to build the object file:

```
-cc compiler
```

Examples

Generate a standard 8 megabytes patch area object file:

```
rtc_patch_area -o patch.o
```

Generate an object file containing a 100,000 byte patch:

```
rtc_patch_area -size 100000 -o patch.o
```

Generate a 1 megabyte patch area shared library:

```
rtc_patch_area -so rtc1M.so -size 1000000
```

9

Command Reference

check | uncheck

All forms of the `check` and `uncheck` commands are described below.

The `check` command prints the current status of RTC:

```
check
```

To turn on access checking:

```
check -access
```

To turn on leak checking:

```
check -leaks [-frames n] [-match m]
```

`-frames` *n*	Up to *n* distinct stack frames are displayed when showing the allocation trace of the leaked block.
`-match` *m*	Used for combining 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.

The default value of *n* is 8 or the value of *m* (whichever is larger), with a maximum value of 16. The default value of *m* is 2.

The command `check -memuse` implies `check -leaks`. In addition to a leak report at the program exit, you also get a memory use report. At any time during program execution, you can see where all the memory in the program has been allocated.

To turn on memory use checking:

```
check -memuse [-frames n] [-match m]
```

`-frames` *n*	Up to *n* distinct stack frames are listed when showing the allocation trace of the block in use.
`-match` *m*	There may be many blocks in use in the program, so RTC automatically combines the blocks allocated from the same execution trace into one report. The decision to combine the report is controlled by value of *m*. If the call stack at the allocation of two or more blocks matches *m* frames, then these blocks are reported in a combined block in use report. The way that blocks are combined is similar to the method used in a leak report.

The default value of *n* is 8 or the value of *m* (whichever is larger), with a maximum value of 16. The default value of *m* is 2.

These two commands are equivalent:

```
check -all [-frames n] [-match m]
check -access ; check -memuse [-frames n] [-match m]
```

To turn off access checking:

```
uncheck -access
```

To turn off leak checking:

```
uncheck -leaks
```

To turn off memory use checking (leak checking is also turned off):

```
uncheck -memuse
```

The following two commands are equivalent:

```
uncheck -all
uncheck -access; uncheck -memuse
```

These two commands are equivalent:

```
uncheck [funcs] [files] [loadobjects]
suppress all in funcs files loadobjects
```

This command allows you to turn on checking in specific functions, modules, and loadobjects while leaving it turned off for the rest of the program:

```
check function* file* loadobject*
```

This command is equivalent to:

```
suppress all
unsuppress all in function* file* loadobject*
```

The command operates cumulatively. For example, the first three commands are equivalent to the last four commands:

```
check main
check foo
check f.c

suppress all
unsuppress all in main
unsuppress all in foo
unsuppress all in f.c
```

Notice that the `suppress all` command is only applied once, leaving checking turned on for `main`, `foo`, and `f.c`.

These commands are also equivalent:

```
uncheck function* file* loadobject
suppress all in function* file* loadobject*
```

showblock

```
showblock -a addr
```

When memory use checking or memory leak checking is turned on, `showblock` shows the details about the heap block at address *addr*. The details include the location of the block's allocation and its size.

showleaks

To report new memory leaks since the last `showleaks` command:

```
showleaks [-a] [-m m] [-n num] [-v]
```

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.

`-a`	Shows all leaks generated so far (not just the leaks since the last `showleaks` command).
`-m` *m*	Used for combining leaks; if the call stack at the time of allocation for two or more leaks matches *m* frames, then the leaks are reported in a single combined leak report. The -m option overrides the global value of *m* specified with the `check` command. The default value of *m* is 2 or the global value last given with `check`.
`-n` *num*	Shows up to *num* records in the report. Default is to show all records.
`-v`	Generates verbose output. Default is to show non-verbose output.

showmemuse

This command generates a report showing all blocks of memory in use:

```
showmemuse [-a] [-m m] [-n num] [-v]
```

Report the top *num* blocks-in-use records sorted by size. Only blocks that are new since the last `showmemuse` command are shown.

`-a`	Shows all blocks in use so far.
`-m` *m*	Used for combining the blocks-in-use reports; if the call stack at the time of allocation for two or more blocks matches *m* frames, then the blocks are reported in a single combined report. The `-m` option overrides the global value of *m* specified with the `check` command. The default value of *m* is 2 or the global value last given with `check`.
`-n` *num*	Shows up to *num* records in the report. Default is to show 20.
`-v`	Generates verbose output. Default is to show non-verbose output.

When memory use checking is on, at program exit an implicit `showmemuse -a -n 20` is performed. You can get a verbose output at the program exit time by using the `rtc_biu_at_exit dbxenv` variable.

suppress|unsuppress

Some or all files in a loadobject may not be compiled with the `-g` switch. This implies that there is no debugging information available for functions that belong in these files. RTC uses some default suppression in these cases.

To get a list of these defaults:

```
{suppress | unsuppress} -d
```

To change the defaults for one loadobject:

```
{suppress | unsuppress} -d [error type] [in loadobject]
```

To change the defaults for all loadobjects:

```
{suppress | unsuppress} -d [error type]
```

To reset these defaults to the original settings:

```
suppress -reset
```

Use the following command to suppress or unsuppress the most recent error. The command applies only to access errors and not to leak errors:

```
{suppress | unsuppress} -last
```

To display the history of the suppress commands not including the `-d` and `-reset` commands:

```
{suppress | unsuppress}
```

Turn error reports on or off for the specified error types for the specified location.

```
{suppress | unsuppress} [error type... [location_specifier]]
```

To remove the `suppress` or `unsuppress` events as given by the *id*(s).

```
suppress -r id...
```

To remove all the suppress and unsuppress events as given by `suppress`.

```
suppress -r [0 | all | -all]
```

Error Type Location Specifier

The table shows the error type location specifier.

`in` *loadobject*	All functions in the designated program or library
`in` *file*	All functions in *file*
`in` *function*	Named *function*
`at` *line specifier*	At source *line*
`addr` *address*	At hex *address*

To see a list of the loadobjects, type `loadobjects` at the `dbx` prompt. If the *line specifier* is blank, the command applies globally to the program. Only one *line specifier* may be given per command.

RTC Errors

Address in Block (`aib`*)*

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.

```
char *ptr;
main()
{
        ptr = (char *)malloc(4);
        ptr++;              /* Address in Block */
}
```

Address in Register (`air`*)*

Problem: A possible memory leak. An allocated block has not been freed, and no reference to the block exists anywhere in program memory.
Possible causes: All references to an allocated block are contained in registers. 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.

`if (i == 0) {`
`  char *ptr = (char *)malloc(4);`
`  /* ptr is going out of scope */`
`}`
`/* Memory Leak or Address in Register */`

Bad Free (`baf`*)*

Problem: Attempt to free memory that has never been allocated.
Possible causes: Passing a non-heap data pointer to `free`() or `realloc()`.

`char a[4];`
`char *b = &a[0];`
 	
`free(b);  /* Bad free (baf) */`

Duplicate Free (`duf`*)*

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.

`char *a = (char *)malloc(1);`
`free(a);`
`free(a);  /* Duplicate free (duf) */`

*Misaligned Free (*maf*)*

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`.

`char *ptr = (char *)malloc(4);`
`ptr++;`
`free(ptr);  /* Misaligned free */`

*Misaligned Read (*mar*)*

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.

`char *s = “hello world”;`
`int *i = (int *)&s[1];`
`int j;`
 	
`j = *i;  /* Misaligned read (mar) */`

*Misaligned Write (*maw*)*

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.

`char *s = “hello world”;`
`int *i = (int *)&s[1];`
 	
`*i = 0;  /* Misaligned write (maw) */`

*Memory Leak (*mel*)*

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.

`char *ptr;`
`    ptr = (char *)malloc(1);`
`    ptr = 0;`
`/* Memory leak (mel) */`

*Out of Memory (*oom*)*

Problem: Attempt to allocate memory beyond physical memory available.
Cause: Program cannot obtain more memory from the system. Useful in tracking down problems that occur when the return value from `malloc()` is not checked for `NULL`, which is a common programming mistake.
`char *ptr = (char *)malloc(0x7fffffff);` `/* Out of Memory (oom), ptr == NULL */`

*Read from Unallocated Memory (*rua*)*

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.
`char c, *a = (char *)malloc(1);` `c = a[1];  /* Read from unallocated memory (rua) */`

*Read from Uninitialized Memory (*rui*)*

Problem: Attempt to read from uninitialized memory.
Possible causes: Reading local or heap data that has not been initialized.

`foo()`
`{  int i, j;`
`   j = i;  /* Read from uninitialized memory (rui) */`
`}`

*Write to Read-Only Memory (*wro*)*

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 has been `mmap`'ed as read-only.

`foo()`
`{  int *foop = (int *) foo;`
`   *foop = 0;  /* Write to read-only memory (wro) */`
`}`

*Write to Unallocated Memory (*wua*)*

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.
`char *a = (char *)malloc(1);` `a[1] = '\0';  /* Write to unallocated memory (wua) */`

`dbxenv` *Variables*

The following `dbxenv` variables control the operation of RTC. If you want to permanently change any of these variables from their default values, place the `dbxenv` commands in the `$HOME/.dbxrc` file. Then, your preferred values are used whenever you use RTC.

dbxenv rtc_auto_continue {on | off}

`rtc_auto_continue on` causes RTC not to stop upon finding an error, but to continue running.. It also causes all errors to be redirected to the `rtc_error_log_file_name`. The default is: `off`.

dbxenv rtc_auto_suppress {on | off}

`rtc_auto_suppress on` causes a particular access error at a particular location to be 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. The default is: `on`.

dbxenv rtc_biu_at_exit {on | off | verbose}

This variable is used when memory use checking is on. If the value of the variable is `on`, a non-verbose memory use (blocks in use) report is produced at program exit. The default is: `on`.

If the value is verbose, a verbose memory use report is produced at program exit. The value off causes no output. This variable has no effect on the `showmemuse` command.

dbxenv rtc_error_log_file_name filename

`rtc_error_log_file_name` redirects RTC error messages to the designated file instead of to the standard output of dbx. The default is: `/tmp/dbx.errlog.`*`uniqueid`*.

The program does not automatically stop when run time errors are detected in batch mode. All error output is directed to your `rtc_error_log_file_name` file. 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 `rtc_error_log_file_name` file. To redirect all errors to the terminal, set the `rtc_error_log_file_name` to `/dev/tty`.

dbxenv rtc_error_limit *n*

n is the maximum number of errors that RTC reports. 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 5 access errors and 5 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.

dbxenv rtc_mel_at_exit {on | off | verbose}

This variable is used when leak checking is on. If the value of the variable 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. This variable has no effect on the `showleaks` command. The default is: `on`.

Using `fix` *and* `continue` *10*

Using `fix` allows you to quickly recompile edited source code without stopping the debugging process.

This chapter is organized into the following sections:

Basic Concepts

The `fix` and `continue` feature allows you to modify and recompile a 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 `fix` if a build is in process; the output from the two processes will intermingle in the Building window.

How `fix` *and* `continue` *Operates*

Before applying the `fix` command you need to edit the source in the editor window. After saving changes, type `fix`.

Once `fix` has been invoked, `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 `fix` and `continue` for files originally compiled *without* debugging information. See the `-g` option description for more information.

You can fix shared objects (`.so`) files, but they have to be opened in a special mode. You can use either `RTLD_NOW|RTLD_GLOBAL` or `RTLD_LAZY|RTLD_GLOBAL` in the call to `dlopen`.

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

Restrictions

WorkShop might have problems 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 need to make any of the proceeding changes, rebuild your program.

Fixing Your Program

To fix your file:

1. Save the changes to your source.

   WorkShop automatically saves your changes if you forget this step.

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. `fix` 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.

`fix` 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 need to rebuild your program to merge the changes into the executable. When you quit debugging, a message reminds you to rebuild your program.

Continuing after Fixing

You can continue executing using `continue`.

Before resuming program execution, you should be aware of the following conditions:

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 `fix` then `run`. Using `run` is faster because it does not relink the program.

Changing a function not yet called

If you 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 made changes to the function currently being executed, `fix`'s impact depends on where the change is relative to the stopped in function:

- If the change is in already executed code, the code is not re-executed. Execute the code by popping the current function off the stack and continuing from where the changed function is called. You need to know your code well enough to figure out whether the function has side effects that can't be undone (for example, opening a file).
- If the change is in code yet to be executed, the new code is run.

Changing a function presently on the stack

If you made changes to a function presently on the stack, but not the stopped in function, the changed code will not be used for the present call of that function. When the stopped in function returns, the old versions of the function on the stack execute.

There are several ways to solve this problem:

- `pop` the stack until all changed functions are removed from the stack. You need to know your code to be sure that there are no ill effects.
- Use the `cont at` *linenum* command to continue from another line.

- Manually repair data structures (use the `assign` command) before continuing.
- Rerun the program using `start`.

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` or `fix` command. To manually reassign correct values to global variables, use the `assign` command.

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 SEGV 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 will run, but it will not print the correct value because the global variable `from` has already been incremented by one. The program would print `hip`s and not `ships`. Use the `assign` command to restore the global variable and then `continue`. Now the program prints the correct string:

```
(dbx) assign from = from-1
(dbx) cont
ships
```

Command Reference

The `fix` command takes the following options:

```
fix [options] [file1, file2,...]
```

`-a`	Fixes all modified files.
`-f`	Forces fixing the file, even if the source was not changed.
`-c`	Prints the compilation line, which may include some options added internally for use by `dbx`.
`-g`	Strips `-O` flags from the compilation line and adds the `-g` flag to it.
`-n`	Sets a no execution mode. Use this option when you want to list the source files to be fixed without actually fixing them.
`file1, file2,...`	Specifies a list of modified source files to fix.

If `fix` is invoked with an option other than `-a` and without a filename argument, only the current modified source file is fixed.

Note – 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.

Note – C++ template definitions cannot 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 `Templates.DB`. The `-ptr` compiler switch is not supported by the `fix` command in `dbx`.

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 pathname to another. Mapping is applied to source paths and object file paths.

10

Collecting Data 11

This chapter is organized into the following sections:

Basic Concepts

During the execution of an application, the Collector gathers behavior data and writes it to a file, called an experiment.

When collecting behavior data, you can define the following:

- The directory and file name to which you want experiment data written
- The period of time during which to sample data
- The type of data to collect

For best results, you should compile your application using the -g option, which allows the compiler to provide more detailed information for the Collector to gather.

For more information, see Collecting Performance Data in the WorkShop online help.

11

Using the Collector

Before you can collect data, runtime checking must be turned off.

There are a series of collector commands that you can use to collect performance data.

To start collecting data, use one of the following:

```
collector sample mode continuous
collector sample mode manual
```

Profiling

For non-multithreaded applications, you can get profiling for a selected region of the code by turing on profiling in the middle of the run and then turning it off at a convenient location. Given the restriction that you cannot turn on profiling in the middle of the run, you can get profiling for a selected region of code with the following:

You can't set this once the program starts running
 	
Don't want profiling for the whole application
 	
This is OK now because it was setup before the program run.
	`collector sample mode [ manual | continuous ]`
	`collector profile mode [ pc_only | stack ]`
	`stop in main`
	`run`
	`collector profile mode off`
	`....`
	
	`#When you reach the beginning of the interesting point:`
	
	`collector profile mode [ pc_only | stack ]`
	`....`
	`#When you reach the end of the interesting point:`
	
	`collector profile mode off`

Collecting Data for Multithreaded Applications

Data collection for multithreaded applications is supported. There are some restrictions and limitation about multithreaded collection:

- You cannot collect information about the code which executes in the init sections of shared libraries (a.out is ok).
- For profiling, there are some restrictions:
  - It is only supported on Solaris 2.5 or 2.6.
  - You cannot start profiling in the middle of the run. You have to start it in the beginning. However, you can disable it in the middle of run and then enable it again.
  - You cannot change the profile timer in the middle of the run.

For multithreaded collecting:

```
collectormt
```

Command Reference

The syntax of the command is:

```
collector collector_commands
```

To stop collecting data, use one of these commands:

```
collector sample mode off
quit
```

To show settings of one or more categories:

```
collector show options
```

no option	List setting options
`all`	Show all settings
`profile`	Show profile settings
`sample`	Show sample settings
`store`	Show store settings
`working_set`	Show working_set settings

To specify profile options:

```
collector profile options
```

`mode stack|pc_only|off`	Specifies profile data collection mode
`timer milliseconds`	Stops collecting performance data

To specify one or more sample options:

```
collector sample options
```

`mode continuous|manual|off`	Specifies data collection mode
`timer milliseconds`	Specifies data collecting period

To inquire status about current experiment:

```
collector status
```

To find the directory or file name of the stored experiment:

```
collector store options
```

`directory string`	Specifies directory where experiment is stored
`filename string`	Specifies experiment file name

To specify working_set options:

```
collector working_set <options>
```

`mode on|off`	Specifies data collection mode

Debugging Multithreaded Applications 12

This chapter describes how to find information about and debug threads using the `dbx thread` commands.

This chapter is organized into the following sections:

Basic Concepts

With `dbx`, you can examine stack traces of each thread, resume (`cont`) a specific thread or all threads, `step` or `next` a specific thread, and navigate between threads.

`dbx` can debug multithreaded applications that use either Solaris threads or POSIX 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`.

12

Understanding Multithreaded Debugging

When it detects a multithreaded program, `dbx` tries to `dlopen` `libthread_db.so`, a special system library for thread debugging located in `/usr/lib`.

`dbx` is a synchronous `dbx`; 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.

Thread Information

The following thread information is available:

```
(dbx) threads
    t@1 a |@1  ?()  running   in main()
    t@2      ?() asleep on 0xef751450  in_swtch()
    t@3 b |@2  ?()  running in sigwait()
    t@4     consumer()  asleep on 0x22bb0 in _lwp_sema_wait()
 *>t@5 b |@4 consumer()  breakpoint     in Queue_dequeue()
    t@6 b |@5 producer()     running       in _thread_start()
(dbx)
```

- The * (asterisk) indicates that an event requiring user attention has occurred in this thread.

  An 'o' instead of an asterisk indicates that a `dbx` internal event has occured.
- The arrow denotes the current thread.
- t@*num*, the thread id, refers to a particular thread. The *number* is the `thread_t` value passed back by `thr_create`.
- b l@*num* or a l@*num* means the thread is bound to or active on the designated LWP, meaning the thread is actually running.
- Start function of the thread as passed to `thr_create`. A ?() means the start function is not known.
- Thread state.
- The function the thread is currently executing.

Viewing the Context of Another Thread

To switch the viewing context to another thread, use the `thread` command. The syntax is:

```
thread [-info] [-hide] [-unhide] [-suspend] [-resume] tid
```

To display the current thread:

```
thread
```

To switch to thread *tid*:

```
thread tid
```

Viewing the Threads List

The following are commands for viewing the `threads` list. The syntax is:

```
threads [-all} [-mode [all|filter] [auto|manual]]
```

To print the list of all known threads:

```
threads
```

To print threads normally not printed (zombies):

```
threads -all
```

Resuming Execution

Use the `cont` command to resume program execution. Currently, threads use synchronous breakpoints so all threads resume execution.

12

Understanding LWP Information

Normally, you need not be aware of LWPs. There are times, however, when thread level queries cannot be completed. In this case, 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)
```

- The * (asterisk) indicates that an event requiring user attention has occurred in this LWP.
- The arrow denotes the current LWP.
- *l@num* refers to a particular LWP.
- The next item represents the LWP state.
- in *func_name()* identifies the function that the LWP is currently executing.

Command Reference

`thread`

In the following commands, a missing *tid* implies the current thread.

Print everything known about the given thread:

```
thread -info tid
```

Hide or unhide the given (or current) thread. It will or will not show up in the generic `threads` listing:

```
thread [-hide | -unhide] tid
```

Unhide all threads:

```
thread -unhide all
```

Keep the given thread from ever running. A suspended thread shows up with an S in the `threads` list:

```
thread -suspend tid
```

Undo the effect of `-suspend`:

```
thread -suspend tid
```

threads

Echoes the current modes:

```
threads -mode
```

Control whether threads prints all threads or filters them by default:

```
threads -mode [all | filter]
```

Thread and LWP States

Thread and LWP States	Description
suspended	Thread has been explicitly suspended.
runnable	Thread is runnable and is waiting for an LWP as a computational resource.

Thread and LWP States	Description
zombie	When a detached thread exits (`thr_exit`), it is in a zombie state until it has rendezvoused 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 is 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.

Customizing dbx 13

This chapter describes the dbxenv 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 .dbxrc

The dbx initialization file, .dbxrc, 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. Remember, if you customize dbx from the command line while you are debugging, those settings apply only to the current debugging session.

During startup, dbx searches for .dbxrc first. The search order is:

1. Current directory ./.dbxrc
2. Home directory $HOME/.dbxrc

If .dbxrc is not found, dbx prints a warning message and searches for .dbxinit (dbx mode).

The search order is:

1. Current directory `./.dbxinit`
2. Home directory `$HOME/.dbxinit`

To suppress the warning message, type in the command pane,

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

A Sample Initialization File

Here is a sample `.dbxrc` file:

```
dbxenv case input_case_sensitive false
catch FPE
```

The first line changes the default setting for the case sensitivity control. `dbxenv` refers to the set of debugging environment attributes. `input_case_sensitive` refers to the matching control. `false` is the control setting.

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.

Command Reference

The `dbxenv` syntax is:

```
dbxenv variable
```

To show all variables:

```
dbxenv
```

Note – Each variable has a corresponding ksh environment variable such as `DBX_trace_speed`. The variable may be assigned directly, or the `dbxenv` command may be used; they are equivalent.

This table shows all of the `dbxenv` variables that you can set:

dbxenv variable	What the dbxenv variable does
`aout_cache_size num`	Size of a.out loadobject cache; set this to *num* when debugging *num* programs serially from a single `dbx`. A *num* of zero still allows caching of shared objects. See entry for `locache_enable`. Default: 1
`array_bounds_check [on|off]`	If on, `dbx` checks the array bounds (Fortran only). Default: on
`cfront_demangling [on|off]`	Governs demangling of Cfront (SC 2.0 and SC 2.0.1) names while loading a program. It is necessary to set this parameter to `on` or start `dbx` with the `-F` option if debugging programs compiled with Cfront or programs linked with Cfront compiled libraries. If set to `off`, `dbx` loads the program approximately 15% faster. Default: off
`disassembler_version [autodetect|v8|v9|v9vis]`	SPARC: Set 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. Non-SPARC platforms: Only 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, inherit or do not inherit events. Default: off
`follow_fork_mode ...`	When process executes a fork / vfork / fork1 ... Default: parent
`follow_fork_mode parent`	... stay with parent.
`follow_fork_mode child`	... follow child.
`follow_fork_mode both`	... follow both parent and child (WorkShop only).
`follow_fork_mode ask`	... ask which of the above the user wants.
`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 77 files, otherwise `true`. If `true`, case matters in variable and function names; otherwise, case is not significant. Default: autodetect

dbxenv variable	What the dbxenv variable does
`language_mode [autodetect|main|c|ansic|c++|objc|pascal|fortran|fortran90]`	Governs the language used for parsing and evaluating expressions. `autodetect`: sets to language of current file. Useful if debugging programs with mixed languages (default mode). `main`: sets language of the main routine in the program. Useful if debugging homogeneous programs. `c`, `c++`, `ansic`, `c++`, `objc`, `pascal`, `fortran`, `fortran90`: sets to selected language.
`locache_enable [on|off]`	Enable or disable loadobject cache entirely. Default: on
`mt_watchpoints [on|off]`	Allow or disallow watchpoint facility for multithreaded programs. Default: off. Warning: Be aware that when using watchpoints on a multithreaded program that the application may deadlock if a semaphore occurs on a page that is being watched.
`output_auto_flush [on|off]`	Automatically call fflush() after each `call`. Default: on
`output_base [8|10|16]`	Default base for printing integer constants . Default: 10
`output_dynamic_type [on|off]`	When on, `-d` is the default for printing, displaying and inspecting. Default: off
`output_inherited_members [on|off]`	When 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 num`	Set # of chars 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]`	Display short pathnames for files. Default: on
`overload_function [on|off]`	For C++, if `on`, do automatic function overload resolution. Default: on
`overload_operator [on|off]`	For C++, if `on`, do automatic operator overload resolution. Default: on
`pop_auto_destruct [on|off]`	If `on`, automatically call appropriate destructors for locals when `pop`'ing a frame. 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 `on`, an RTC error at a given location is reported only once. Default: off
`rtc_biu_at_exit [on|off|verbose]`	Used when `check -memuse` 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

dbxenv variable	What the dbxenv variable does
`rtc_error_limit num`	Number of RTC errors to be reported. Default: 1000
`rtc_error_log_file_name filename`	Name of file where RTC errors are logged if `rtc_auto_continue` is set. Default: /tmp/dbx.errlog.*uniqueID*
`rtc_mel_at_exit [on|off|verbose]`	Used when leaks 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 `on` with no active program, `step`, `next`, `stepi`, and `nexti` implicitly run the program and stop at the language-dependent `main` routine. If `on`, `cont` implies `run` when necessary. Default: off
`run_io [stdio|pty]`	Governs whether the user program's I/O is redirected to `dbx`'s `stdio` or a specific pty. The pty is provided via `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 GUI wrappers.
`run_quick [on|off]`	If 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
`run_savetty [on|off]`	Multiplexes tty settings, process group, and keyboard settings (if -kbd was used on the command line) between dbx and the debuggee; useful when debugging editors and shells. Try setting to `on` if dbx gets SIGTTIN or SIGTTOU and pops back into the shell. Try setting to `off` to gain a slight speed advantage. The setting is irrelevant if dbx is attached to the debugee or is running under WorkShop. Default: on
`run_setpgrp [on|off]`	If `on`, when a program is `run` `setgprp(2)`is called right after the fork. Default: off
`scope_global_enums [on|off]`	If `on`, enumerators are put in global scope and not in file scope. Should be set before debugging information is processed (~/`.dbxrc`). Default: off
`scope_look_aside [on|off]`	Find file static symbols, even when not in scope. Default: on
`session_log_file_name filename`	Name of file where `dbx` logs all commands and their output. Output is appended to the file. Default: "" (no session logging)
`stack_max_size num`	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]`	Allows breakpoints while stepping and nexting.Default: off

dbxenv variable	What the dbxenv variable does
`suppress_startup_message num`	Sets the release level below which the startup message is not printed. Default: 3.01
`symbol_info_compression [on|off]`	Reads debugging information for each `include` file only once. Default: on
`trace_speed num`	Sets the speed of tracing execution. Value is the number of seconds to pause between steps. Default: 0.50

Debugging at the Machine-Instruction Level 14

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`. The `regs` command can be used to print out the contents of machine registers or the `print` command can be used 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.

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 *addr* for *count* items in format *fmt*. The default *addr* is the next one after the last address previously displayed. The default *count* is 1. The default *fmt* 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 [addr] [/ [count] [fmt]]
```

To display the contents of memory from *addr1* through *addr2* inclusive, in format *fmt*:

```
examine addr1, addr2 [/ [fmt]]
```

Display the address, instead of the contents of the address in the given format:

```
examine addr = [fmt]
```

To print the value stored at the next address after the one last displayed by examine:

```
examine +/ i
```

To print the value of an expression, enter the address as an expression:

```
examine addr=format
examine addr=
```

Addresses

The *addr* is any expression resulting in or usable as an address. The *addr* may be replaced with a + (plus sign) which displays the contents of the next address in the default format.

Example addresses are:

`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 *fmt* is the address display format in which `dbx` displays the results of a query. The output produced depends on the current display *fmt*. To change the display format, supply a different *fmt* code.

Set the *fmt* specifier to tell `dbx` how to display information associated with the addresses specified.

The default format set at the start of each dbx session is X, which displays an address/value as a 32-bit word in hexadecimal. The following memory display formats are legal.

i	Display as an assembly instruction
d	Display as a halfword in decimal
D	Display as a word in decimal
o	Display as a half-word in octal.
O	Display as a word in octal.
x	Display as a halfword in hexadecimal.
X	Display as a word in hexadecimal. (default format)
b	Display as a byte in octal
c	Display as a wide 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 as a decimal (4 bytes, same as D)
lo, lO	Display as an octal (4 bytes, same as O)
lx, LX	Display as a hexadecimal (4 bytes, same as X)
Ld, LD	Display as a decimal (8 bytes)
Lo, LO	Display as an octal (8 bytes)
Lx, LX	Display as a hexadecimal (8 bytes)

Count

The *count* is a repetition count in decimal. The increment size depends on the memory display format.

Examples

The following examples show how to use an address with *count* and *fmt* 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)
```

For PowerPC:

```
(dbx) x &malloc/5i
0x01f55088: malloc       :  mflr  %r0
0x01f5508c: malloc+0x0004:  stwu  %r1,0xffe0(%r1)
0x01f55090: malloc+0x0008:  stw   %r0,0x0024(%r1)
0x01f55094: malloc+0x000c:  stw   %r31,0x001c(%r1)
0x01f55098: malloc+0x0010:  stw   %r30,0x0018(%r1)
```

14

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 addr /[count]
```

The `dis` command without arguments displays 10 instructions starting at the address +. With only a *count*, the `dis` command displays count instructions starting at the address +.

Using the `listi` *Command*

To display source lines along with their corresponding assembly instructions, use `listi`, which is equivalent to `list -i`. See the discussion of `list -i` in Chapter 3, “Viewing and Visiting Code”.

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)
```

For PowerPC:

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x02000964: main+0x002c:  lwz  %r3,0xfff0(%r30)
0x02000968: main+0x0030:  addi %r3,%r3,0x0004
0x0200096c: main+0x0034:  lwz  %r3,000000(%r3)
0x02000970: main+0x0038:  bl   0x020110a4      [atoi [PLT]]
0x02000974: main+0x003c:  stw  %r3,0xffe8(%r30)
   14       j = foo(i);
0x02000978: main+0x0040:  lwz  %r3,0xffe8(%r30)
0x0200097c: main+0x0044:  bl   0x020008e0      [foo]
0x02000980: main+0x0048:  stw  %r3,0xffe4(%r30)
```

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 the Machine-Instruction Level

To single-step from one machine-instruction to the next machine-instruction:

♦ **Use** `nexti` **or** `stepi`

`nexti` and `stepi` behave the same as their source-code level counterparts: `nexti` steps *over* functions, `stepi` steps *into* a function called from the `next` instruction (stopping at the first instruction in the called function). The command forms are also the same. See `next` and `step` for a description.

The output from `nexti` and `stepi` differs from the corresponding source level commands in two ways. First, the output includes the *address* of the instruction at which the program is stopped (instead of the source code line number); secondly, the default output contains the *disassembled instruction*.

For example:

```
(dbx) func
hand::ungrasp
(dbx) nexti
ungrasp +0x18:  call support
(dbx)
```

Tracing at the Machine-Instruction Level

Tracing techniques at the machine instruction level work the same as at the source code level, except when you use `tracei`. For `tracei`, `dbx` executes a single instruction only after each check of the address being executed or the value of the variable being traced. `tracei` produces automatic `stepi`-like behavior: the program advances one instruction at a time, stepping into function calls.

When you use `tracei`, it causes the program to stop momentarily after each instruction while `dbx` checks for the address execution or the value of the variable or expression being traced. Using `tracei` can slow execution considerably.

For more information on trace and its event specifications and modifiers, see Chapter 7, “Setting Breakpoints and Traces.”

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 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) 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 Machine-Instruction Level

To set a breakpoint at machine-instruction level, use `stopi`. The command `stopi` 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]
```

Setting a Breakpoint at an Address

To set a breakpoint at a specific address:

```
(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 allows you to enter commands in an `adb(1)` syntax. You may also enter `adb` mode which interprets every command as adb syntax. Most adb commands are supported.

For more information on the `adb` command, see the `dbx` online help.

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); this is a SPARC only option.

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 PowerPC that can be used in expressions.

14

SPARC Register Information

The following register information is for SPARC systems.

Register	Description
`$g0 through $g7`	Global registers
`$o0 through $o7`	“out” registers
`$l0 through $l7`	“in” registers
`$i0 through $i7`	“local” registers
`$fp`	Frame pointer, equivalent to register $i6
`$sp`	Stack pointer, equivalent to register $o6
`$y`	Y register
`$psr`	Processor state register
`$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).

The following additional registers are available on SPARC V9 and V8+ hardware:

```
$xg0 $xg1 through $xg7
$xo0 $xo1 through $xo7
$xfsr $tstate $gsr
$f32f33 $f34f35 through $f62f63
```

See the *SPARC Architecture Reference Manual* and the *Sun-4 Assembly Language Reference Manual* for more information on SPARC registers and addressing.

Intel Register Information

The following register information is for Intel systems.

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
`$ebp`	Frame pointer
`$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:

`$sp`	Stack pointer; equivalent of `$uesp`
`$pc`	Program counter; equivalent of `$eip`
`$fp`	Frame pointer; equivalent of `$ebp`

Registers for the 80386 lower halves (16 bits) are:

`$ax`	General register
`$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:

`$al`	Lower (right) half of register	`$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:

`$fctrl`	Control register
`$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

PowerPC Register Information

The following register information is for PowerPC systems.

Register	**Description**
`$r0 through $r31`	General purpose registers
`$f0 through $f31`	Floating point (double precision) registers
`$cr`	Condition register
`$lr`	Link register
`$pc`	Program counter
`$msr`	Machine state register
`$ctr`	Count register
`$xer`	Integer exception register
`$mq`	Multiply quotient / divide dividend register
`$fpscr`	FP status and control register
`$sp`	Equivalent to $r1

14

Debugging Child Processes 15

This chapter describes how to debug a child process. `dbx` has several facilities to help you debug processes that create children via `fork` (2) and `exec` (2).

This chapter is organized into the following sections:

Attaching to Child Processes

You can attach to a running child in one of the following ways.

When starting `dbx`:

```
$ dbx progname pid
```

From the command line:

```
(dbx) debug progname pid
```

You can substitute *progname* with the name - (minus sign), so dbx finds the executable associated with the given process id (*pid*). After using a -, a subsequent run or rerun will not work because dbx does not know the full pathname of the executable.

Following the exec

If a child process executes a new program using exec(2) or one of its variations, the process id does not change, but the process image does. dbx automatically takes note of an exec() 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 fork

If a child process does vfork(), fork(1), or fork(2), the process id changes, but the process image stays the same. Depending on how the dbxenv variable follow_fork_mode is set, dbx does the following:

Parent—In the traditional behavior, dbx ignores the fork and follows the parent.

Child—In this mode, dbx automatically switches to the forked child using the new pid. All connection to and awareness of the original parent is lost.

Both—This mode is only available when using dbx through WorkShop.

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, in which case you will be prompted again 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 follow fork by setting the dbxenv variable follow_fork_inherit to on, or make them permanent using the -perm eventspec modifier. For more information on using event specification modifiers, see Chapter 8, “Event Management.”

Working With Signals 16

This chapter describes how to use `dbx` to work with signals. `dbx` supports a command named `catch`, which instructs `dbx` to stop a program when `dbx` detects any of the signals appearing on the `catch` list.

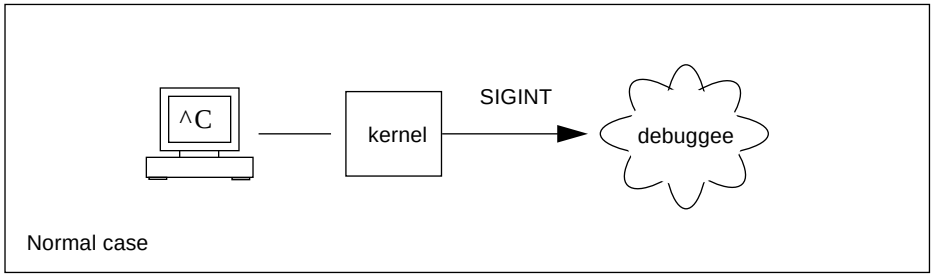
The `dbx` commands `cont`, `step`, and `next` support the `-sig` *signal_name* option, which allows you to resume execution of a program with the program behaving as if it had received the signal specified in the `cont -sig` command.

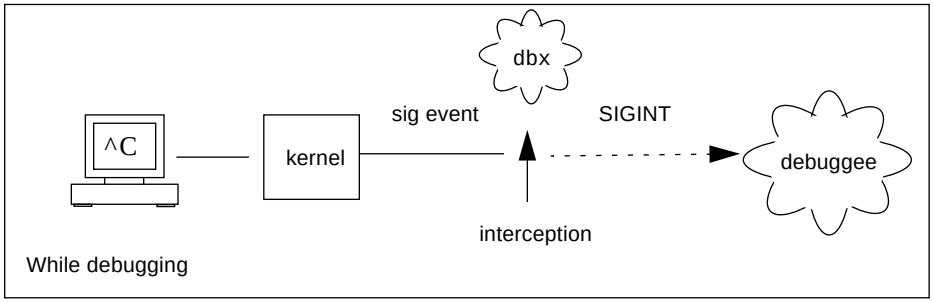
This chapter is organized into the following sections.

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 get a prompt. You then have two choices:

1. **"Cancel" the signal when the program is resumed—the default behavior of `cont`—facilitating easy interruption and resumption with `SIGINT` (Control-C).**





2. **"Forward" the signal to the process using:**

```
cont -sig sig
```

In addition, if a certain signal is received frequently, you can arrange for `dbx` to automatically forward the signal because you do not care to see it displayed:

```
ignore sig # "ignore"
```

However, the *sig* is still forwarded to the process. A default set of signals is automatically forwarded in this manner, see `ignore`.

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.

To see the list of signals currently being trapped:

- **Type `catch` with no signal-name 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-name 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 argument 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

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 allows the program to continue executing quietly. Solaris implements the IEEE Standard for Binary Floating Point Arithmetic definitions of reasonable answers for exceptions.

Since a reasonable answer for floating point exceptions is returned, exceptions do not automatically 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)` for an example of a trap handler.) When you set up a trap handler using `ieee_handler`, the trap enable mask in the hardware floating point status register is set. This trap enable mask causes the exception to raise `SIGFPE` at run time.

Once you have compiled the program with the trap handler, load the program into `dbx`. Before you can catch the `SIGFPE`, you must add `FPE` to the `dbx` signal `catch` list, using the command:

```
(dbx) catch FPE
```

By default, `FPE` is on the `ignore` list.

After adding `FPE` to the `catch` list, run the program in `dbx`. When the exception you are trapping occurs, `SIGFPE` is raised and `dbx` stops the program. Now 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.

For more discussion and examples, see the floating point manual, *Numerical Computation Guide*, which comes with the Sun Compiler documentation.

Sending a Signal in a Program

The `dbx cont` command supports the `-sig` *signal_name* option, which allows you to resume execution of a program with the program behaving as if it had received the system signal *signal_name*.

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 stop, next, and detach commands accept -sig as well.

Automatically Handling Signals

The event management commands can also deal with signals as events. These tow 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
```

16

Debugging C++ 17

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++*

Although this chapter concentrates on two specific aspects of debugging C++, dbx does allow you full functionality when debugging your C++ programs. You can:

- Find out about class definitions
- Print or display inherited data members
- Find out dynamic information about an object pointer
- Find out information about debugging virtual functions
- Using runtime type information
- Set breakpoints on all member functions of a class
- Set breakpoints on all overloaded member functions
- Set breakpoints on all overloaded non-member functions

- Deal with overloaded functions / data members
- Set breakpoints on all member functions of a particular object

For more information on any of these topics, please look in the index under C++.

Exception Handling in dbx

A program stops running if an exception occurs. Exceptions signal programming anomalies, such as division by zero or array overflow. To deal with exceptions, 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 `throw` would be caught if it occurred at a particular point in the program

If you `step` after stopping at a throw point, 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, `step` brings you to the catch block handling the throw.

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 int
(dbx) intercept
int
(dbx) stop in bar
(2) stop in bar(void)
(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 _ex_dbg_will_throw at 0xef76dac8
_ex_dbg_will_throw:     save    %sp, -96, %sp
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
- Printing a list of all class and function template instantiations
- Displaying the definitions of templates and instances
- Calling member template functions and function template instantiations
- Printing values of function template instantiations
- Displaying the source code for function template instantiations

Template Example

The following code example shows the class template Array and its instantiations and the function template square and its instantiations.

In the following 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)

```
 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     {
```

```
 1      template<class C> void square(C num, C *result
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 < iarray.getlength(); ++i)
51          {
52              iarray[i] = i * 2.1;
53          }
54
55          return 0;
56      }
```

Command Reference

You can invoke any of these exception handling commands when debugging a program.

Commands for Handling Exceptions

`exception [-d | +d]`

Use this command to display an exception's type. This variable can be used at any time during debugging. With `-d`, the derived type is shown; otherwise, the static type is shown. This command overwrites the `dbxenv` variable `output_dynamic_type` setting.

`intercept [-a | -x |` *typename*`]`

You can intercept, or catch, exceptions of a specific type before the stack has been unwound. Use this command with no arguments to list the types that are being intercepted. Use `-a` to intercepted all throws. 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 enter:

```
(dbx) intercept -a
(dbx) intercept -x int
```

`unintercept [-a | -x |` *typename*`]`

Use this command to remove exception types from the intercept list. Use this command with no arguments to list the types that are being intercepted (same as `intercept`). 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.

`whocatches` *typename*

This command reports where, if at all, 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* is displayed.

Commands for C++ Exceptions

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*

Use `whereis` 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
class template instance: a.out‘Array<int>
class template instance: a.out‘Array<double>
class template:a.out‘template_doc_2.cc‘Array
```

For a function template:

```
(dbx) whereis square
function template instance: a.out‘square(double, double*)
function template instance: a.out‘square(int, int*)
function template: a.out‘square
```

`whatis` *name*

Use `whatis` to print the definitions of function and class templates and instantiated functions and classes

For a class template:

```
(dbx) whatis Array
template<class T> class Array
To get the full template declaration, try ‘whatis -t Array<int>’;
```

For a function template:

```
(dbx) whatis square
More than one identifier ‘square’.
Select one of the following names:
 0) Cancel
 1) function template instance: ‘square(int, int*)
 2) function template instance: ‘square(double, double*)
 3) ‘a.out‘square
> 3
template<class C> void square(C num, C *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);
```

`stop inclass` *classname*

To stop in all member functions of a template class:

```
(dbx)stop inclass Array
(2) stop inclass Array
```

Use `stop inclass` to set breakpoints at all member functions of a particular template class:

```
(dbx) stop inclass Array<int>
(2) stop inclass Array<int>
```

`stop infunction` *name*

Use `stop infunction` to set breakpoints at all instances of the specified function template:

```
(dbx) stop infunction square
(9) stop infunction square
```

`stop in` *function*

Use `stop in` to set a breakpoint at a member function of a template class or at a template function.

For a class template instantiation:

```
(dbx) stop in Array<int>::Array<int>(int l)
(2) stop in Array<int>::Array<int>(int)
```

For a function instantiation:

```
(dbx) stop in square(double, double*)
(6) stop in square(double, double*)
```

`call` *process [parameters]*

Use `call` to explicitly call a function instantiation or a member function of a class template, provided you are stopped in scope. If `dbx` is unable to choose the correct instance, a menu allows you to choose the correct instance.

`print` *Expressions*

Use `print` 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
}
```

`list` *expressions*

Use `list` to print the source listing for the specified function instantiation:

```
(dbx) list square(int, int*)
```

17

Debugging Fortran Using dbx 18

This section introduces some dbx features likely to be used 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.

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 one of three different breakpoints, depending on whether the current file is `a1.f, a2.f`, or `a3.f`. dbx

Uppercase Letters (Fortran 77 only)

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. (The current release of `f90` is case-insensitive.)

Fortran 77 and dbx must be in the same case-sensitive or case-insensitive mode:

- To compile and debug in case-insensitive mode, do so without the `-U` option. The default then is `dbxenv case insensitive`.

  If the source has a variable named `LAST`, then in dbx, both the `print LAST` or `print last` commands work. Both `f77` and dbx consider `LAST` and `last` to be the same, as requested.

- To compile and debug in case-sensitive mode, use `-U`. The default is then `dbxenv case sensitive`.

  If the source has a variable named `LAST` and one named `last`, then in dbx, `print LAST` works, but `print last` does *not* work. Both `f77` 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 `dbxenv case insensitive` environment attribute.

Optimized Programs

To debug optimized programs:

- Compile the main program with `-g` but with no `-O`*n*.
- Compile every other routine of the program with the appropriate `-O`*n*.
- Start the execution under dbx.

- Use `fix -g` *any*`.f` on the routine you want to debug, but no `-O`*n*.
- Use `continue` with that routine compiled.

Main 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
```

Sample dbx *Session*

The following examples use a sample program called `my_program`.

1. Compile and link with the dbx`-g` flag. You can do this in one or two steps.

Compile and link *in one step*, with `-g`:

```
demo% f77 -o my_program -g a1.f a2.f a3.f
```

Or, compile and link *in separate steps*:

```
demo% f77 -c -g a1.f a2.f a3.f
demo% f77 -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 in uppercase, *subnam* can be uppercase or lowercase.

4. **Enter the `run` command, which runs the program in the executable files named when you started `dbx`.**

   Run the program from within `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, enter the `print` command.

Print value of `n`:

```
(dbx) print n
n = 2
```

Print the matrix `twobytwo`; the format may 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, enter 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 next with step. 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:

- step sets a breakpoint at the first source statement of the subprogram.
- next sets the breakpoint at the first source statement after the call, but still in the calling program.

7. **To quit dbx, enter 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 is defined.

Using dbx to Locate Problems

Use *dbx*dbx to find the source code line where a segmentation fault 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 dbxsegmentation fault:

```
demo% f77 -g -silent WhereSEGV.f
demo% a.out
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation fault (core dumped)
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 =%all` forces trapping on all exceptions.

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% f77 -g -o wh -ftrap=%all wh.f
wh.f:
 MAIN:
        joe:
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 brought 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:

.

```
demo% f77 -silent -g ShowTrace.f
demo% a.out
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quil 174% dbx a.out
Reading symbolic information for a.out
...
(dbx) run
Running: a.out
(process id 1089)
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)
```

Note the reverse order:

MAIN called calc
calc called calcb.

Execution stopped, line 23
calcB called from calc, line 9
calc called from MAIN, line 3

Working With Arrays

dbxdbx 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
```

Fortran 90 Allocatable Arrays

The following example shows how to work with allocated arrays in dbx.

```
demo% f90 -g Alloc.f90
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
```

Alloc.f90

	(dbx) **stop at 6** (2) stop at "alloc.f90":6 (dbx) **stop at 9** (3) stop at "alloc.f90":9 (dbx) **run** Running: a.out (process id 10749) Size? 1000 stopped in main at line 6 in file "alloc.f90" 6 ALLOCATE(buffer(n), STAT=status)
Unknown size is at line 6	(dbx) **whatis buffer** integer*4 , allocatable::buffer(:) (dbx) **next** continuing stopped in main at line 7 in file "alloc.f90" 7 IF (status /= 0) STOP 'cannot allocate buffer' (dbx) **whatis buffer** integer*4 buffer(1:1000)
Known size is at line 9	(dbx) **cont** stopped in main at line 9 in file "alloc.f90" 9 PRINT *, buffer(n) (dbx) **print n** n = 1000
buffer(1000) holds 1000	(dbx) **print buffer(n)** buffer(n) = 1000

Slicing Arrays

The syntax for Fortran array-slicing:

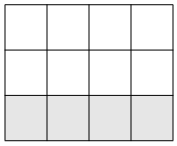
```
print arr-exp(first-exp:last-exp:stride-exp)
```

Variable	Description	Default
arr-exp	Expression that should evaluate to an array type	
first-exp	First element printed	Lower bound
last-exp	Last element printed	Upper bound
stride-exp	Stride	1

To Specify rows and columns:

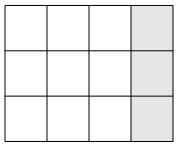
```
demo% f77 -g -silent ShoSli.f
demo% dbx a.out
Reading symbolic information for a.out
(dbx) list 1,12
    1       INTEGER*4 a(3,4), col, row
    2       DO row = 1,3
    3           DO col = 1,4
    4             a(row,col) = (row*10) + col
    5           END DO
    6       END DO
    7       DO 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
```

Print row 3:



```
(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)
```

Print column 4:



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

dbxShowing Intrinsic Functions

dbx recognizes Fortran intrinsic functions.

To show an intrinsic function in dbx:

```
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)
```

18

dbxShowing Complex Expressions

dbx also recognizes Fortran complex expressions.

To show a complex expression in dbx:

```
demo% cat ShowComplex.f
    COMPLEX z
    z = ( 2.0, 3.0 )
    END
demo% f77 -g -silent 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%
```

dbxShowing Logical Operators

dbx can locate Fortran logical operators and print them.

To show logical operators in dbx:

```
demo% cat ShowLogical.f
        LOGICAL a, b, y, z
        a = .true.
        b = .false.
        y = .true.
        z = .false.
        END
demo% f77 -g -silent 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%
```

18

Viewing Fortran 90 Derived Types

You can show structures—`f90` derived types with `dbx`.

```
demo% f90 -g DebStruc.f90
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.f90":17
(dbx) run
Running: a.out
(process id 12326)
stopped in main at line 17 in file "Struct.f90"
   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 90 Derived Type

You can show structures—`f90` derived types, and pointers with `dbx`.

```
demo% f90 -o debstr -g DebStruc.f90
demo% dbx debstr
(dbx) stop in main
(2) stop in main
(dbx) list 1,99
```

DebStruc.f90

```
    1   PROGRAM DebStruPtr! Debug structures & pointers
    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), TARGET :: prod1, prod2
   11      TYPE(product), POINTER :: curr, prior
   12
   13      curr => prod2
   14      prior => prod1
   15      prior%id = 82
   16      prior%name = "Coffee Cup"
   17      prior%model = "XL"
   18      prior%cost = 24.0
   19      prior%price = 104.0
   20      curr = prior
   21      WRITE ( *, * ) curr%name, " ", prior%name
   22   END PROGRAM DebStruPtr
(dbx) stop at 21
(1) stop at "DebStruc.f90":21
(dbx) run
Running: debstr
```

Declare a derived type.

Declare *prod1* and *prod2* targets.
Declare *curr* and *prior* pointers.

Make *curr* point to *prod1*.
Make *prior* point to *prod1*.
Initialize *prior*.

Set *curr* to *prior*.
Print *name* from *curr* and *prior*.

```
(process id 10972)
stopped in main at line 21 in file "DebStruc.f90"
   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 `f90` derived type.

Ask about the variable

Ask about the type (-t)

```
(dbx) whatis prod1
product prod1
(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:

`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
)
```

Fortran 90 Generic Functions

To work with Fortran 90 generic functions:

```
(dbx) list 1,99
    1   MODULE cr
    2     INTERFACE cube_root
    3       FUNCTION s_cube_root(x)
    4         REAL :: s_cube_root
    5         REAL, INTENT(IN) :: x
    6       END FUNCTION s_cube_root
    7       FUNCTION d_cube_root(x)
    8         DOUBLE PRECISION :: d_cube_root
    9         DOUBLE PRECISION, INTENT(IN) :: x
   10       END FUNCTION d_cube_root
   11     END INTERFACE
   12   END MODULE cr
   13   FUNCTION s_cube_root(x)
   14       REAL :: s_cube_root
   15       REAL, INTENT(IN) :: x
   16       s_cube_root = x ** (1.0/3.0)
   17   END FUNCTION s_cube_root
   18   FUNCTION d_cube_root(x)
   19       DOUBLE PRECISION :: d_cube_root
   20       DOUBLE PRECISION, INTENT(IN) :: x
   21       d_cube_root = x ** (1.0d0/3.0d0)
   22   END FUNCTION d_cube_root
   23   USE cr
   24     REAL :: x, cx
   25     DOUBLE PRECISION :: y, cy
   26     WRITE(*,"('Enter a SP number: ')")
   27     READ (*,*) x
   28     cx = cube_root(x)
   29     y = x
   30     cy = cube_root(y)
   31     WRITE(*,'("Single: ",F10.4, ", ", F10.4)') x, cx
   32     WRITE(*,'("Double: ",F12.6, ", ", F12.6)') y, cy
   33     WRITE(*,"('Enter a DP number: ')")
   34     READ (*,*) y
   35     cy = cube_root(y)
   36     x = y
   37     cx = cube_root(x)
   38     WRITE(*,'("Single: ",F10.4, ", ", F10.4)') x, cx
   39     WRITE(*,'("Double: ",F12.6, ", ", F12.6)') y, cy
   40     END
```

To use dbx with a generic function, cube root.

```
(dbx) stop at 26
(2) stop at "Generic.f90":26
(dbx) run
Running: Generic
(process id 14633)
stopped in main at line 26 in file "Generic.f90"
   26     WRITE(*,"('Enter a SP number : ')")
(dbx) whatis cube_root
More than one identifier 'cube_root.'
Select one of the following names:
 1) 'Generic.f90'cube_root s_cube_root ! real*4 s_cube_root
 2) 'Generic.f90'cube_root s_cube_root ! real*8 d_cube_root
> 1
 real*4 function cube_root (x)
 (dummy argument) real*4 x
(dbx) print cube_root(8.0)
More than one identifier 'cube_root.'
Select one of the following names:
 1) 'Generic.f90'cube_root ! real*4 s_cube_root
 2) 'Generic.f90'cube_root ! real*8 d_cube_root
> 1
cube_root(8) = 2.0
(dbx) stop in cube_root
More than one identifier 'cube_root.'
Select one of the following names:
 1) 'Generic.f90'cube_root ! real*4 s_cube_root
 2) 'Generic.f90'cube_root ! real*8 d_cube_root
> 1
(3) stop in cube_root
(dbx) cont
continuing
Enter a SP number:
8
stopped in cube_root at line 16 in file "Generic.f90"
   16       s_cube_root = x ** (1.0/3.0)
(dbx) print x
x = 8.0
```

If asked "What is cube_root?", select one.

If asked for cube_root(8), dbx asks you to select which one.

If told to stop in cube_root, dbx asks you to select which one.

From inside s_cube_root, show current value of x.

dbx *and the Dynamic Linker* 19

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:

Basic Concepts

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:

1. **Program startup**
   At program startup, rtld runs first and dynamically loads all shared objects specified at link time. These are *preloaded* shared objects and commonly include libc.so, libC.so, and libX.so. (Use ldd(1) to find out what shared objects a program will load.)

2. **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 load objects to refer to a shared object (.so) or executable (a.out).

19

The dynamic linker maintains a list of all loaded objects in a list called a link map, which maintained in user memory, and is indirectly accessed through `libthread_db.so`, a special system library for thread debugging.

`dbx` traverses the link map to see:

- Which objects were loaded
- What their corresponding binaries are
- At what base address they were loaded

Corruption of these data structures can at times confuse `dbx`.

Debugging Support for Shared Objects

`dbx` can debug shared objects, both preloaded and those opened with `dlopen()`. Some restrictions and limitations are described in the following sections.

Startup Sequence

To put breakpoints in preloaded shared objects, the address of the routines has to be known to `dbx`. For `dbx` to know the address of the routines, it must know the shared object base address. Doing something as simple as:

```
stop in printf
run
```

requires special consideration by `dbx`. Whenever you load a new program, `dbx` automatically executes the program up to the point where `rtld` has completed construction of the link map. `dbx` then reads the link map and stores the base addresses. After that, the process is killed and you see messages and the prompt. These `dbx` tasks are completed silently.

At this point, the symbol table for `libc.so` is available as well as its base load address. Therefore, the address of `printf` is known.

The activity of `dbx` *waiting* for `rtld` to construct the link map and accessing the head of the link map is known as the rtld handshake. The event `syncrtld` occurs when `rtld` is done with the link map and `dbx` has read all of the symbol tables.

With this scheme, dbx depends on the fact that when the program is rerun, the shared libraries are loaded at the same base address. The assumption that shared libraries are loaded at the same base address is seldom violated; usually only if you change LD_LIBRARY_PATH between loading of the program and running it. In such cases, dbx takes note of the new address and prints a message. However, breakpoints in the moved shared object may be incorrect.

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.

dlopen() *and* dlclose()

dbx automatically detects that a dlopen or a dlclose has occurred and loads the symbol table of the loaded object. You can put breakpoints in and debug the loaded object like any part of your program.

When a shared object is unloaded, the symbol table is discarded and the breakpoints are marked as "(defunct)" when you request status. There is no way to automatically re-enable the breakpoints if the object is opened again on a consecutive run.

Two events, dlopen and dlclose, can be used with the when command and some shell programming, to help ease the burden of managing breakpoints in dlopen type shared objects.

fix *and* continue

Using fix and continue with shared objects requires a change in how they are opened in order for fix and continue to work correctly. Use mode RTLD_NOW|RTLD_GLOBAL or RTLD_LAZY|RTLD_GLOBAL.

Procedure Linkage Tables (PLT)

PLTs are structures used by the `rtld` to facilitate calls across shared object boundaries. For instance, the call to `printf` goes via 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.

Setting a Breakpoint in a Dynamically Linked Library

`dbx` provides full debugging support for code that makes use of the programmatic interface to the run-time linker; that is, 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()/dlclose()` allows you to 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.

Three Exceptions

- You cannot set a breakpoint in a `dlopen`'ed library before that library is loaded by `dlopen()`.
- You cannot set a breakpoint in a `dlopen`'ed "filter" library until the first function in it is called.
- When a library is loaded by `dlopen()`, an initialization routine named `_init()` is called. This routine may 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`.

Using the Korn Shell 20

The dbx command language is based on the syntax of the Korn Shell[1] (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:

Features of ksh-88 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

1. The Korn Shell Command and Programming Language, Morris I. Bolsky and David G. Korn, Prentice Hall,

- backquote (`...`) for command substitution (use `$(...)` instead)
- [[*expr*]] 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.
- *addr/fmt* is now `examine` *addr/fmt*.
- */pattern* is now `search` *pattern*, and ?*pattern* is now `bsearch` *pattern*.

Modifying a Program State 21

This chapter focuses on `dbx` usage and commands that change your program or change the behavior of your program 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:

Basic Concepts

Your application might behave differently when run under `dbx`. Although `dbx` strives to minimize its impact on the child, you should be aware of the following:

- You may forget to take out a -C or disable RTC. Just having the RTC support library `librtc.so` loaded into a program can cause the program to behave differently.
- Your `dbx` initialization scripts may 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 contents of `argv[]`, forcing local variables to be allocated a bit differently. If they're not initialized, they will get different random numbers. This problem can be detected via runtime checking.

- The program does not initialize `malloc`()'ed memory before use; a situation similar to the previous one. This problem can be detected via runtime checking.
- If the program references an uninitialized memory location via a bad pointer; a situation similar to the previous one. This problem can be detected via runtime checking.
- `dbx` has to catch LWP creation and `dlopen` events, which might seriously affect the timings of things.
- `dbx` does context switching on signals, so if you have heavy use of signals which are timing sensitive, things might work differently.
- The program expects 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 in your program, look at all uses of `mmap`() and ensure that the address returned is actually used by the program, rather than a hard-coded address.
- If the program is multithreaded, it may 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, see if running with `adb` or `truss` causes the same problems.

Also, `dbx` should impose minimal perturbation when it attaches to a running process.

Using Commands

`assign`

The `dbx assign` command assigns a value to the *exp* to *var*. Using it in `dbx` permanently alters the value of *var*, which could modify your program.

```
assign var = exp
```

pop

The dbx pop command pops a frame or frames from the stack:

Pop current frame Pop *num* frames Pop frames until specified frame number	pop pop *num* pop -f *num*

Any calls popped are reexecuted upon resumption, which may result in undesirable program changes. pop also calls destructors for objects local to the popped functions.

call

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 actually making the call as if you had written it into your program source.

print

To print the value of the expression(s):

```
print exp, ...
```

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.

when

The when command has a general syntax as follows:

```
when event-specification [modifier] {cmd ... ;}
```

When the event occurs, the *cmd*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.

fix

You can use `fix` to make on-the-fly changes to your program:

```
fix
```

It is a very useful tool, but remember that `fix` 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, “Using fix and continue.”

cont at

This `dbx` 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.

User Tips 22

This chapter includes the following tips:

Using dbx *Equivalents for Common* GDB *Commands*

The following table lists approximate equivalent dbx commands for some common GNU Debugging (GDB) commands:

GDB	DBX
`break` *line*	`stop at` *line*
`break` *func*	`stop in` *func*
`break` **addr*	`stopi at` *addr*
`break ... if` *expr*	`stop ... -if` *expr*

GDB	DBX
cond *n*	stop ... -if *expr*
tbreak	stop ... -temp
watch *expr*	stop *expr* [slow]
watch *var*	stop modify &*var* [fast]
catch *x*	intercept *x*
info break	status
info watch	status
clear	clear
clear *fun*	delete *n*
delete	delete all
disable	handler -disable all
disable *n*	handler -disable *n*
enable	handler -enable all
enable *n*	handler -enable *n*
ignore *n cnt*	handler -count *n cnt*
commands *n*	when ... { cmds; }
backtrace *n*	where n
frame *n*	frame n
info reg *reg*	print *$reg*
finish	step up
signal *num*	cont sig *num*
jump *line*	cont at *line*
set *var=expr*	assign *var=expr*
x/*fmt addr*	x *addr/fmt*
disassem *addr*	dis *addr*
shell *cmd*	sh *cmd* [if needed]
info func *regex*	funcs *regexp*
ptype *type*	whatis -t *type*

GDB	DBX
`define cmd`	`function cmd`
`handle` *sig*	`stop sig` *sig*
`info signals`	`status; catch`
`attach` *pid*	`debug -` *pid*
`attach` *pid*	`debug a.out` *pid*
`file` *file*	`[unnecessary]`
`exec` *file*	`debug` *file*
`core` *file*	`debug` *a.out corefile*
`set editing on`	`set -o emacs`
`set language` *x*	`language` *x*
`set prompt` *x*	`PS1=`*x*
`set history size` *x*	`HISTSIZE=`*x*
`set print object on`	`dbxenv output_dynamic_type on`
`show commands`	`history`
`dir` *name*	`pathmap` *name*
`show dir`	`pathmap`
`info line <`*n*	`listi` *n*
`info source`	`file`
`info sources`	`files; modules`
`forw` *regex*	`search` *regexp*
`rev` *regex*	`bsearch` *regexp*

Reviewing dbx *Changes*

Using the .dbxinit *File*

Long-time users of dbx might be using the .dbxinit file instead of the newer .dbxrc file. If you have a .dbxrc file, dbx reads it and ignores any .dbxinit file present. If necessary, you can get dbx to read both by adding the following

lines to your `.dbxrc` file:

```
kalias alias=dalias
source ~/.dbxinit
kalias alias=kalias
```

If you don't have a `.dbxrc` file, but do have a `.dbxinit` file, you should see the warning message:

```
Using .dbxinit compatibility mode. See `help .dbxrc' for more
information.
```

Currently, `dbx` still reads your `.dbxinit` file. although this feature may disappear in a future release.

If you still use a dbxinit file, review Chapter 13, “Customizing dbx” for information about using the `.dbxrc` file instead.

Alias Definition

The `alias` command is now a pre-defined alias for `dalias` or `kalias`.

The Symbols / and ?

With the introduction of a KornShell-based parser, the / (forward slash) command had to be renamed because it cannot be distinguished from a UNIX pathname. Use `search` instead.

This example now means to execute the file `abc` from the root directory.

```
(dbx) /abc
```

Similarly, ? (question mark) had to be renamed because it is now a shell metacharacter. Use `bsearch` instead.

This example reads expand the pattern that matches all files in the current directory with a four-character filename having abc as the last three characters, then execute the resulting command.

```
(dbx) ?abc
```

If you use these commands frequently, you may wish to create aliases for them:

```
alias ff=search         find forward
alias fb=bsearch        find backward
```

Embedded Slash Command

The embedded slash command was renamed. This is no longer valid:

```
0x1234/5X
```

Use the `examine` command or its alias, `x`:

```
examine 0x1234/5X
x 0x1234/5X
```

Using `assign` *Instead of* `set`

`set` is now the KornShell `set` command, and is no longer an alias for `assign`.

Enabling Command-Line Editing

You can enable command-line editing in several ways. First, if `$FCEDIT`, `$EDITOR`, or `$VISUAL` is set, its value is checked. If the last component, the component after the last slash, contains the string `emacs`, then `emacs`-mode is enabled. If it contains `vi`, `vi`-mode is enabled. If none of the three environment variables are set or if the first one in the list that *is* set does not contain `emacs` or `vi`, then command-line editing is disabled.

You can enable `emacs` mode or `vi` mode from the command line or in your `.dbxrc` file:

```
set -o emacs
set -o vi
```

To disable command-line editing,:

```
set +o emacs +o vi
```

Being In Scope

If `dbx` claims that *abc* is not defined in the current scope, it means that no symbol named *abc* is accessible from the current point of execution. See Chapter 3, “Viewing and Visiting Code” for details.

Locating Files

All files created by `dbx` are placed in the directory `/top` unless the environment variable `TMPDIR` is set, in which case the directory `$TMPDIR` is used.

If your source files are not where they were when they were compiled, or if you compiled on a different machine than you are debugging on and the compile directory is not mounted as the same pathname, `dbx` cannot find them. See the pathmap command in Chapter 2, “Starting dbx” for a solution.

Reaching Breakpoints

If you do not reach the breakpoint you expected to reach, consider the following possibilities:

- If you've placed a breakpoint on a function, it might not be the function you think. Use `whereis` *funcname* to find out how many functions of a given name exist. Then use the exact syntax as presented by the output of `whereis` in your `stop` command.

- If the breakpoint is on a while loop, the compiler logically converts code of the following form to:

```
while (condition)
       statement
```

```
if (condition) {
again:
   statement
   if condition
        goto again;
}
```

In this situation, the breakpoint you put on the `while` amounts to putting a breakpoint only on the outer `if`, because `dbx` cannot deal with source lines that map to two or more different addresses.

C++ member and `whatis` *Command*

Sometimes the type of a C++ member is missing in the output of `whatis`. In the following example, the type of member *stackcount* is missing:

```
(dbx) whatis stack
class stack {
...
static stackcount;       /* Never defined or allocated */
...
};
```

When a class member is not defined or allocated, `dbx` cannot determine its type so there is no type to print.

Runtime Checking 8Megabyte Limit

Only access checking has this limit. Leak checking is not affected by this limit.

For access checking, RTC replaces each load and store instruction with a branch instruction that branches to a patch area. This branch instruction has an 8Mb range. If the debugged program has used up all the address space within 8Mb of the particular load / store instruction being replaced, there is no place to put the patch area. If RTC can't intercept ALL loads and stores to memory it cannot provide accurate information and so disables access checking completely.

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; it turns off access checking after printing an error message. For workarounds, see Chapter 9, “Using Runtime Checking.”

Locating Floating-Point Exceptions with dbx

You need to do two things. First, to stop the process whenever an FP exception occurs, type:

```
(dbx) catch FPE
```

Next, add the following to your Fortran application:

```
integer ieeer, ieee_handler, myhandler
ieeer = ieee_handler('set', 'all', myhandler)
...

integer function myhandler(sig, code, context)
integer sig, code(5)
call abort()
end
```

This is necessary because the ieee software typically sets all errors to be silent (not raising signals). This causes *all* ieee exceptions to generate a SIGFPE as appropriate, which is probably too much.

You can further tailor which exceptions you see by adjusting the parameters of `ieee_handler()` or by using an alternative to the `dbx catch` command:

```
stop sig FPE
```

which acts just like `catch FPE`, or

```
stop sig FPE subcode
```

For finer control, *subcode* can be one of the following:

FPE_INTDIV	1	integer divide by zero
FPE_INTOVF	2	integer overflow
FPE_FLTDIV	3	floating point divide by zero
FPE_FLTOVF	4	floating point overflow
FPE_FLTUND	5	floating point underflow
FPE_FLTRES	6	floating point inexact result
FPE_FLTINV	7	invalid floating point operation
FPE_FLTSUB	8	subscript out of range

Note that `stop` and `catch` are independent and that if you use `stop FPE` you should also `ignore FPE`.

Using dbx *with Multithreaded Programs*

Multithreaded features are an inherent part of the standard `dbx`.

The major multithreaded features offered by `dbx` are:

- The `threads` command will give a list of all known threads, with their current state, base functions, and current functions.
- You can examine stack traces of each thread.
- You can resume (`cont`) a specific thread or all threads.
- You can `step` or `next` a specific thread.
- The `thread` command helps navigating between threads.

You can limit your scope to a specific thread. `dbx` maintains a cursor to the "current" or "active" thread. It is manipulable by the `thread` command. The only commands that use the current thread as the default thread are `where` and `thread -info`.

Thread Numbering

`dbx` knows the id of each thread (the type `thread_t`) as returned by `thr_create()`. The syntax is `t@`*`number`*.

LWP Numbering

`dbx` knows the id of each LWP (the type `lwpid_t`) as presented by the /proc (man procfs(4)) interface. The syntax is `l@`*`number`*.

Breakpoints on a Specific Thread

You can have a breakpoint on a specific thread by filtering a regular breakpoint:

```
stop in foo -thread t@4
```

Where the `t@4` refers to the thread with id 4.

When a thread hits a breakpoint, all threads stop. This is knows as "sympathetic stop", or "stop the world".

From the point of view of `/proc` and LWPs this is synchronous debugging.

To ease thread navigation, put this in your `.dbxrc`.

```
_cb_prompt() {
        if [ $mtfeatures = "true"]
        then
                PS1='[$thread $lwp]: '
        else
                PS1="(dbx-$proc) "
        fi
}
```

dbx *Identification of Multithreaded Applications*

If an application is linked with `-lthread`, `dbx` assumes it is multithreaded.

The Collector, RTC, `fix` *and* `continue`*, and Watchpoints*

The collector and `fix` and `continue` work with multithreaded applications. RTC works with multithreaded applications, however a libthread patch is needed.

The implementation of watchpoints does not depend on the operating system, and has the potential of too easily getting the multithreaded application into a deadlock or other obscure problems.

Multithreaded Pitfalls

It is very easy to get your program to deadlock by resuming only a specific thread while other threads are still and hold a resource that the resumed thread might need.

libthread data structures are in user space and might get corrupted by bugs involving rogue pointers. In such cases one suggestion is to work at the LWP level with commands like `lwps' and `lwp' which are analogous to their thread equivalents.

Sleeping Threads

You cannot “force” a sleeping thread to run. In general, when debugging multithreaded applications it is recommended that you take a “stand back and watch” approach rather than trying to alter the program’s natural execution flow.

`thr_join`, `thr_create()`,*and* `thr_exit`

Starting from the threads list, you can determine which thread id came from which start function. The “base function” as it is known, is printed in the thread listing.

When you attach to an existing multithreaded process, it is non-deterministic which thread becomes the active thread.

When the active thread does a thr_create, the current threads stays with the “creating thread”. In the follow_fork analogy, it would be parent.

The Sun multithreaded model doesn't have true fork semantics for threads. There is no thread tree, and no parent-child relationships as there is with processes. `thr_join()` is only a simplified veneer.

When the active thread does a `thr_exit`, `dbx` makes a dummy “dead” thread as the active thread. This thread is represented as `t@X`.

Part 2 — Using Multithreaded Tools

Using LoopReport 23

The Fortran MP and MP C compilers automatically parallelize loops that they determine safe and profitable to parallelize. LoopReport is a performance analysis tool that reads loop timing files created by these compilers.

This chapter is organized as follows:

Basic Concepts

LoopReport is the command line version of LoopTool. LoopReport produces an ASCII file of loop times.

LoopReport's main features include the ability to:

- Time all loops, whether serial or parallel
- Produce a table of loop timings

- Collect hints from the compiler, during compilation. These hints can help you parallelize loops that were not parallelized. Hints are described further in “Understanding Compiler Hints” on page 255.

Using LoopReport is similar to using `gprof`. The three major steps are: compile, run, and analyze.

Note – The following examples use the Fortran MP (f77) compiler. The options shown (such as `-xparallel` and `-Zlp`) work also for MP C.

Setting Up Your Environment

1. **Before compiling, set the environment variable** PARALLEL **to the number of processors on your machine.**
   The following command makes use of `psrinfo`, a system utility. *Note the backquotes*:

```
% setenv PARALLEL ‘/usr/sbin/psrinfo | wc -l‘
```

Note – If you have installed LoopReport in a non-default directory, substitute that path for the one shown here.

---

2. **Before starting LoopReport, make sure the environment variable** XUSERFILESEARCHPATH **is set:**

```
% setenv XUSERFILESEARCHPATH \
    /opt/SUNWspro/lib/sunpro_defaults/looptool.res
```

3. **Set** LD_LIBRARY_PATH**.**
   If you are running Solaris 2.5:

```
% setenv LD_LIBRARY_PATH /usr/dt/lib:$LD_LIBRARY_PATH
```

If you are running Solaris 2.4:

```
% setenv LD_LIBARY_PATH \
 /opt/SUNWspro/Motif_Solaris24/dt/lib:$LD_LIBRARY_PATH
```

You may want to put these commands in a shell start-up file (such as `.cshrc` or `.profile`).

Creating a Loop Timing File

To compile for automatic parallelization, typical compilation options are `-xparallel` and `-xO4`. To compile for LoopReport, add `-Zlp`, as shown in the following example:

```
% f77 -xO4 -xparallel -Zlp source_file
```

Note – All examples apply to Fortran77, Fortran90 and C programs.

After compiling with `-Zlp`, run the instrumented executable. This creates the loop timing file, *program.looptimes*. LoopReport processes two files: the instrumented executable and the loop timing file.

Starting LoopReport

When it starts up, LoopReport expects to be given the name of your program. Type `loopreport` and the name of the program (an executable) you want examined.

```
% loopreport program
```

You can also start LoopReport with no file specified. However, if you invoke LoopReport without giving it the name of a program, it looks for a file named `a.out` in the current working directory.

```
% loopreport > a.out.loopreport
```

You can also direct the output into a file, or pipe it into another command:

```
% loopreport program > program.loopreport
% loopreport program | more
```

Timing File

LoopReport also reads the *timing file* associated with your program. The timing file is created when you use the `-zlp` option, and contains information about loops. Typically, this file has a name of the format *program*`.looptimes`, and is found in the same directory as your program.

However, there are four ways to specify the location of a timing file. LoopReport chooses a timing file according to the rules listed below.

- If a timing file is named on the command line, LoopReport uses that file.

```
% loopreport program  newtimes > program.loopreport
```

- If the command-line option `-p` is used, LoopReport looks in the directory named by `-p` for a timing file.

```
% loopreport program -p /home/timingfiles > program.loopreport
```

- If the environment variable LVPATH is set, LoopReport looks in that directory for a timing file.

```
% setenv LVPATH /home/timingfiles
% loopreport program > program.loopreport
```

- LoopReport writes the table of loop statistics to `stdout`—the standard output. You can also redirect the output to a file, or pipe it into another command:

```
% loopreport  program > program.loopreport
% loopreport program | more
```

Other Compilation Options

To compile for automatic parallelization, typical compilation switches are `-xparallel` and `-x04`. To compile for LoopReport, add `-Zlp`.

```
% f77 -x04 -xparallel -Zlp source_file
```

There are several other useful options for examining and parallelizing loops.

Option	Effect
`-o` *program*	Renames the executable to *program*
`-xexplicitpar`	Parallelizes loops marked with `DOALL` pragma
`-xloopinfo`	Prints hints to `stderr` for redirection to files

Either `-xO3` or `-xO4` can be used with `-xparallel`. If you don't specify `-xO3` or `-xO4` but you do use `-xparallel`, then `-xO3` is added. The table below summarizes how switches are added.

You type:	Bumped Up:
`-xparallel`	`-xparallel -xO3`
`-xparallel -Zlp`	`-xparallel -xO3 -Zlp`
`-xexplicitpar`	`-xexplicitpar -xO3`
`-xexplicitpar -Zlp`	`-xexplicitpar -xO3 -Zlp`
`-Zlp`	`-xdepend -xO3 -Zlp`

The `-xexplicitpar` and `-xloopinfo` have specific applications.

`-xexplicitpar`

The Fortran MP compiler switch `-xexplicitpar` is used with the pragma `DOALL`. If you insert `DOALL` before a loop in your source code, you are explicitly marking that loop for parallelization. The compiler will parallelize this loop when you compile with `-xexplicitpar`.

The following code fragment shows how to mark a loop explicitly for parallelization.

```
    subroutine adj(a,b,c,x,n)
      real*8 a(n), b(n), c(-n:0), x
        integer n
c$par DOALL
    do 19 i = 1, n*n
      do 29 k = i, n*n
        a(i) = a(i) + x*b(k)*c(i-k)
29    continue
19  continue
    return
    end
```

When you use `-Zlp` by itself, `-xdepend` and `-xO3` are added. The switch `-xdepend` instructs the compiler to perform the data dependency analysis that it needs to do to identify loops. The switch `-xparallel` includes `-xdepend`, but `-xdepend` does not imply (or trigger) `-xparallel`.

-xloopinfo

The `-xloopinfo` option prints hints about loops to `stderr` (the UNIX standard error file, on file descriptor 2) when you compile your program. The hints include the routine names, line number for the start of the loop, whether the loop was parallelized, and, if appropriate, the reason it was not parallelized.

The following example redirects hints about loops in the source file `gamteb.F` to the file named `gamteb.loopinfo`.

```
% f77 -xO3 -parallel -xloopinfo -Zlp gamteb.F 2> gamteb.loopinfo
```

The main difference between `-Zlp` and `-xloopinfo` is that in addition to providing you with compiler hints about loops, `-Zlp` also instruments your program so that timing statistics are recorded at runtime. For this reason, also, `LoopReport` analyzes only programs that have been compiled with `-Zlp`.

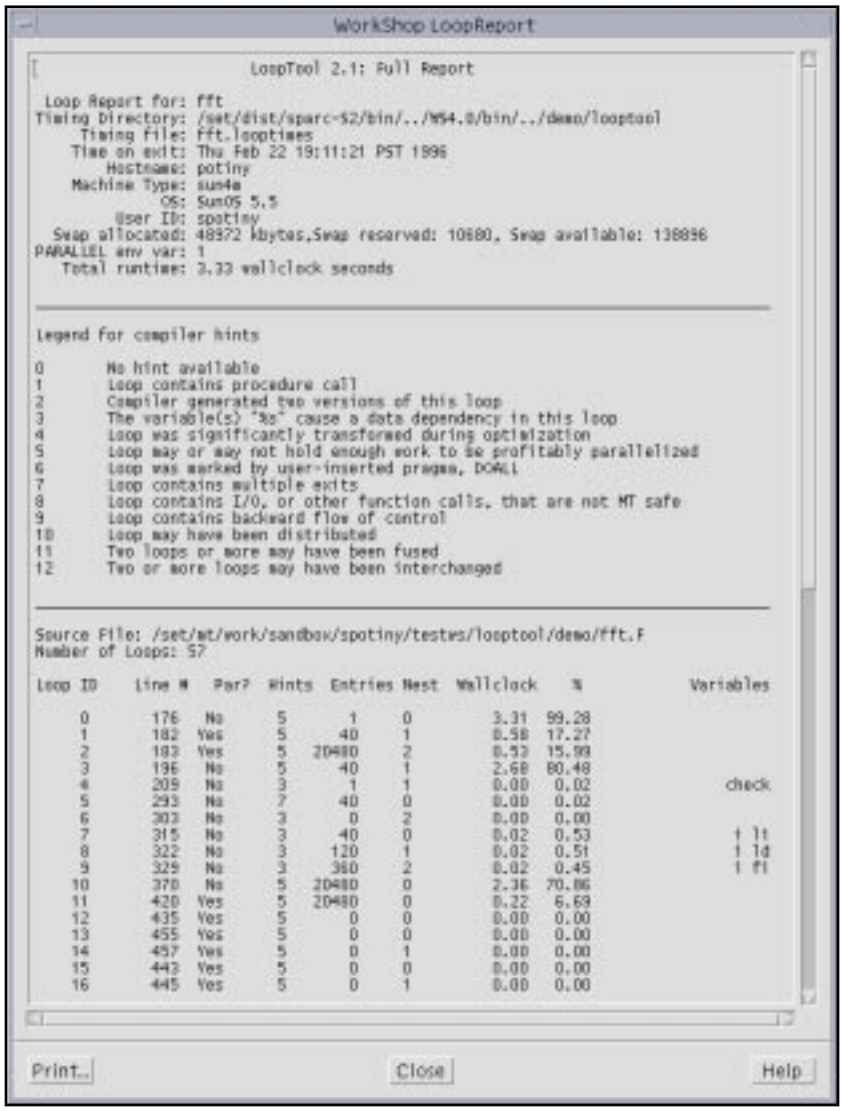


Figure 23-1 Sample Loop Report

23

Fields in the Loop Report

The loop report contains the following information:

- Loopid
  An arbitrary number, assigned by the compiler during compile time. This is just an internal loopid, useful for talking about loops, but not really related in any way to the user's program.

- Line#
  The line number of the first statement of the loop in the source file.

- Par?

  "Parallelized by the compiler?" *Y* means that this loop was marked for parallelization; *N* means that the loop was not.

- Entries

  Number of times this loop was entered from above. This is distinct from the number of loop iterations, which is the total number of times a loop executes. For example, these are two loops in Fortran.

```
do 10 i=1,17
    do 10  j=1,50
        ...some code...
    10 continue
```

  The first loop is entered once, and it iterates 17 times. The second loop is entered 17 times, and it iterates 17*50 = 850 times.

- Nest
  Nesting level of the loop. If a loop is a top-level loop, its nesting level is 0. If the loop is the child of another loop, its nesting level is 1.

  For example, in this C code, the `i` loop has a nesting level of 0, the `j` loop has a nesting level of 1, and the `k` loop has a nesting level of 2.

```
for (i=0; i<17; i++)
    for (j=0; j<42; j++)
        for (k=0; k<1000; k++)
            do something;
```

- Wallclock

  The total amount of elapsed wallclock time spent executing this loop for the whole program. The elapsed time for an outer loop includes the elapsed time for an inner loop. For example:

```
for (i=1; i<10; i++)
    for (j=1; j<10; j++)
        do something;
```

  The time assigned to the outer loop (the `i` loop) might be 10 seconds, and the time assigned to the inner loop (the `j` loop) might be 9.9 seconds.

- Percentage

  The percentage of total program runtime measured as wallclock time spent executing this loop. As with wallclock time, outer loops are credited with time spent in loops they contain.

- Variable

  The names of the variables that cause a data dependency in this loop. This field only appears when the compiler hint indicates that this loop suffers from a data dependency. The following illustrates a data dependency:

```
for  (i=0; i<10; i++) {
    a[i] = b * c;
    d[i] = a[i] + e;
}
```

  This loop contains a data dependency—the variable `a[i]` must be computed before the variable `d[i]` can be computed. The variable `d[i]` is dependent on `a[i]`.

Understanding Compiler Hints

LoopReport present you with somewhat cryptic hints about the optimizations applied to a particular loop, and the reason why a particular loop may not have been parallelized.

Note – The hints are gathered by the compiler during the optimization pass. They should be understood in that context; they are *not* absolute facts about the code generated for a given loop. However, the hints are often very useful indications of how you can transform your code so that the compiler can perform more aggressive optimizations, including parallelizing loops.

Some of the hints are redundant; that is, two hints may appear to mean essentially the same thing.

Let Sun know which of the hints help you or what other sorts of hints you need from the compiler. You can send feedback by using the Comment form available from the About box in the LoopTool GUI. See *WorkShop: Beyond the Basics* for more information about the LoopTool GUI.

Finally, read the sections in the *Fortran User's Guide* and *C User's Guide* that address parallelization. There are useful explanations and tips inside these manuals.

The table lists the optimization hints applied to loops.

Hint #	Hint Definition
0	No hint available
1	Loop contains procedure call
2	Compiler generated two versions of this loop
3	Loop contains data dependency
4	Loop was significantly transformed during optimization
5	Loop may or may not hold enough work to be profitably parallelized
6	Loop was marked by user-inserted pragma, `DOALL`
7	Loop contains multiple exits
8	Loop contains I/O, or other function calls, that are not MT safe
9	Loop contains backward flow of control
10	Loop may have been distributed
11	Two or more loops may have been fused
12	Two or more loops may have been interchanged

0. No hint available

None of the other hints applied to this loop. That does not mean that none of the other hints might apply—it simply means that the compiler did not infer any of those hints.

1. Loop contains procedure call

The loop could not be parallelized since it contains a procedure call that is not MT safe. If such a loop were parallelized, there is a chance that multiple copies of the loop could instantiate the function call simultaneously, trample on each other's use of any variables local to that function, trample on return values, and generally invalidate the function's purpose. If you are certain that the procedure calls in this loop are MT safe, you can direct the compiler to parallelize this loop by inserting the `DOALL` pragma before the body of the loop. For example, if `foo` is an MT-safe function call, then you can force this inner loop to be parallelized by inserting `c$par DOALL`:

```
c$par DOALL
     do 19 i = 1, n*n
         do 29 k = i, n*n
            a(i) = a(i) + x*b(k)*c(i-k)
            call foo()
 29     continue
 19  continue
```

The `DOALL` pragmas are interpreted by the compiler only when you compile with `-parallel` or with `-explicitpar`; if you compile with `-autopar`, the compiler ignores the `DOALL` pragmas. This can be handy for debugging or fine-tuning.

2. Compiler generated two versions of this loop

The compiler couldn't tell, at compile time, if the loop contained enough work to be profitable to parallelize. The compiler generated two versions of the loop, a serial version and a parallel version, and a runtime check that will choose, at runtime, which version should execute. The runtime check determines the amount of work that the loop has to do by checking the loop iteration values.

3. **Loop contains data dependency**
   A variable inside the loop is affected by the value of a variable in a previous iteration of the loop. For example:

```
do 99 i=1,n
    do 99 j = 1,m
        a[i, j+1] = a[i,j] + a[i,j-1]
99 continue
```

   This is a contrived example because, for such a simple loop, the optimizer would simply swap the inner and outer loops so that the inner loop could be parallelized. But this example demonstrates the concept of data dependency, often referred to as *data-carried dependency.*

   The compiler will often be able to tell you the names of the variables that cause the data-carried dependency. If you rearrange your program to remove (or minimize) such dependencies, the compiler will be able to perform more aggressive optimizations.

4. **Loop was significantly transformed during optimization**
   The compiler performed some optimizations on this loop that might make it almost impossible to associate the generated code with the source code. For this reason, line numbers may be incorrect. Examples of optimizations that can radically alter a loop are loop distribution, loop fusion, and loop interchange (see Hint 10, Hint 11, and Hint 12).

5. **Loop may or may not hold enough work to be profitably parallelized**
   The compiler was not able to determine at compile time whether this loop definitely held enough work to warrant the overhead of parallelizing. Often, loops that are labeled with this hint may also be labeled as “parallelized,” meaning that the compiler generated two versions of the loop (see Hint 2), and that it will be decided at runtime whether the parallel version or the serial version should be used.

   All the compiler hints, including the flag that indicates whether or not a loop is parallelized, are generated at compile time. There’s no way to be certain that a loop labeled as “parallelized” actually executes in parallel. You need to perform additional runtime tracing, such as can be accomplished with the Thread Analyzer. You can compile your programs

with both `-Zlp` (for LoopReport) and with `-Ztha` (for Thread Analyzer) and compare the analysis of both tools to get as much information as possible about the runtime behavior of your program.

6. **Loop was marked by user-inserted pragma,** `DOALL`
   This loop was parallelized because the compiler was instructed to do so by the `DOALL` pragma. This hint helps you easily identify those loops that you explicitly wanted to parallelize.

   The `DOALL` pragmas are interpreted by the compiler only when you compile with `-parallel` or `-explicitpar`; if you compile with `-autopar`, then the compiler will ignore the `DOALL` pragmas, which can be handy for debugging or fine-tuning.

7. **Loop contains multiple exits**
   The loop contains a `GOTO` or some other branch out of the loop other than the natural loop end point. For this reason, it is not safe to parallelize the loop, since the compiler has no way of predicting the runtime behavior of the loop.

8. **Loop contains I/O, or other function calls, that are not MT safe**
   This hint is similar to Hint 1; the difference is that this hint often focuses on I/O that is not MT safe, whereas Hint 1 could refer to any sort of MT-unsafe function call.

9. **Loop contains backward flow of control**
   The loop contains a `GOTO` or other control flow up and out of the body of the loop. That is, some statement inside the loop jumps back to some previously executed portion of code, as far as the compiler control flow can determine. As with the case of a loop that contains multiple exits, this condition means that the loop is not safe to parallelize.

   If you can reduce or minimize backward flows of control, the compiler will be able to perform more aggressive optimizations.

10. **Loop may have been distributed**
    The contents of the loop may have been distributed over several iterations of the loop. That is, the compiler may have been able to rewrite the body of the loop so that the loop could be parallelized. However, since this rewriting takes place in the language of the internal representation of the optimizer, it's very difficult to associate the original source code with the rewritten version. For this reason, hints about a distributed loop may refer to line numbers that don't correspond to line numbers in your source code.

11. **Two or more loops may have been fused**
    Two consecutive loops were combined into one, so that the resulting larger loop contains enough work to be profitably parallelized. Again, in this case, source line numbers for the loop may be misleading.

12. **Two or more loops may have been interchanged**
    The loop indices of an inner and an outer loop have been swapped, to move data dependencies as far away from the inner loop as possible, and to enable this nested loop to be parallelized. In the case of deeply nested loops, the interchange may have occurred with more than two loops.

Compiler Optimizations and How They Affect Loops

As you might infer from the descriptions of the compiler hints, it can be tricky to associate optimized code with source code. Clearly, you would prefer to see information from the compiler presented to you in a way that relates as directly as possible to your source code. After all, most people don't care about the compiler's internal representation of their programs. Unfortunately, the compiler optimizer is "reading" your program in terms of this internal language. Although it tries its best to relate that to your source code, it is not always successful.

Inlining

Inlining is an optimization applied only at optimization level `-O4` and only for functions contained with one file. Suppose one file contains 17 Fortran functions, and 16 of those can be inlined into the first function. If you compile the file using `-O4`, the file with the source code for those 16 functions could be copied into the body of the first function. When further optimizations are applied, and it becomes difficult to say which loop on which source line was subjected to which optimization.

If the compiler hints seem particularly opaque, consider compiling with `-O3 -parallel -Zlp`, so that you can see what the compiler has to say about your loops before it tries to inline any of your functions.

In particular, if you notice "phantom" loops—that is, loops that the compiler claims to exist, but which you know do not exist in your source code—this could well be a symptom of inlining.

Loop Transformations — Unrolling, Jamming, Splitting, and Transposing

The compiler performs many optimizations on loops that radically change the body of a loop. These include loop unrolling, loop jamming, loop splitting, and loop transposition.

LoopReport attempts to provide you with hints that make as much sense as possible. Given the nature of the problem of associating optimized code with source code, however, the hints may be misleading. If you are interested in this topic, refer to compiler books such as *Compilers: Principles, Techniques and Tools* by Aho, Sethi, and Ullman, for more information on what these optimizations do for your code.

Parallel Loops Nested Inside Serial Loops

If a parallel loop is nested inside a serial loop, the runtime information reported by LoopReport may be misleading.

Each loop is stipulated to use the wallclock time of each of its loop iterations. If an inner loop is parallelized, it is assigned the wallclock time of each iteration, although some of those iterations are running in parallel.

However, the outer loop is only assigned the runtime of its child, the parallel loop, which will be the runtime of the longest parallel instantiating of the inner loop.

This leads to the anomaly of the outer loop consuming “less” time than the inner loop. Keep in mind the funny nature of time when measuring events that occur in parallel, and this will help keep all wallclock times in perspective.

23

Using LockLint 24

This chapter is organized as follows:

Basic Concepts

In the multithreading model, a *process* consists of one or more threads of control that share a common address space and most other process resources. Threads must acquire and release locks associated with the data they share. If they fail to do so, a *data race* may ensue—a situation in which a program may produce different results when run repeatedly with the same input.

Data races are easy problems to introduce. Simply accessing a variable without first acquiring the appropriate lock can cause one. Data races are generally very difficult to find. Symptoms generally manifest themselves only if two threads access the improperly protected data at nearly the same time; hence a data race may easily run correctly for months without showing any signs of a

problem. It is extremely difficult to exhaustively test all concurrent states of a program for even a simple multithreaded program, so conventional testing and debugging are not an adequate defense against data races.

Most processes share several resources. Operations within the application may require access to more than one of those resources. This means that the operation will need to grab a lock for each of the resources before performing the operation. If different operations use a common set of resources, but the order in which they acquire the locks is inconsistent, there is a potential for *deadlock*. For example, the simplest case of deadlock occurs when two threads hold locks for different resources and each thread tries to acquire the lock for the resource held by the other thread.

LockLint Overview

LockLint is a utility that analyzes the use of mutex and multiple readers / single writer locks, and looks for inconsistent use of these locking techniques.

When analyzing locks and how they are used, LockLint detects a common cause of data races: failure to hold the appropriate lock while accessing a variable.

Table 24-1, Table 24-2, and Table 24-3 list the routines of the Solaris and POSIX `libthread` APIs recognized by LockLint.

Table 24-1 Calls for Manipulating Mutex Locks

Solaris	POSIX	Kernel (Solaris only)
`mutex_lock`	`pthread_mutex_lock`	`mutex_enter`
`mutex_unlock`	`pthread_mutex_unlock`	`mutex_exit`
`mutex_trylock`	`pthread_mutex_trylock`	`mutex_tryenter`

Table 24-2 Calls for Manipulating Readers-Writer Locks (Solaris API)

User	Kernel
`rw_rdlock, rw_wrlock`	`rw_enter`
`rw_unlock`	`rw_exit`
`rw_tryrdlock, rw_trywrlock`	`rw_tryenter`
	`rw_downgrade`
	`rw_tryupgrade`

Table 24-3 Calls for Manipulating Condition Variables

Solaris	POSIX	Kernel (Solaris only)
`cond_broadcast`	`pthread_cond_broadcast`	`cv_broadcast`
`cond_wait`	`pthread_cond_wait`	`cv_wait`
		`cv_wait_sig`
		`cv_wait_sig_swap`
`cond_timedwait`	`pthread_cond_timedwait`	`cv_timedwait`
		`cv_timedwait_sig`
`cond_signal`	`pthread_cond_signal`	`cv_signal`

Additionally, LockLint recognizes the thread types shown in Table 24-4.

Table 24-4 Thread Types Recognized by LockLint

Solaris	POSIX	Kernel (Solaris only)
`mutex_t`	`pthread_mutex_t`	`kmutex_t`

LockLint reports several kinds of basic information about the modules it analyzes, including:

- Locking side effects of functions. Unknown side effects can lead to data races or deadlocks.
- Accesses to variables that are not consistently protected by at least one lock, and accesses that violate assertions about which locks protect them. This information can point to a potential data race.
- Cycles and inconsistent lock-order acquisitions. This information can point to potential deadlocks.
- Which variables were protected by a given lock. This can assist in judging the appropriateness of the chosen *granularity.*

LockLint provides a wealth of subcommands for specifying assertions about the application. During the *analysis* phase, LockLint reports any violation of the assertions.

Note – Add assertions liberally, and use the analysis phase to refine assertions and to make sure that new code does not violate the established locking conventions of the program.

24

Collecting Information for LockLint

The compiler gathers the information used by LockLint. More specifically, you specify a command-line option, `-Zll`, to the C compiler to generate a `.ll` file for each `.c` source code file. The `.ll` file contains information about the flow of control in each function and about each access to a variable or operation on a mutex or readers-writer lock.

Note – No `.o` file is produced when the `-Zll` flag is passed.

LockLint User Interface

There are two ways for you to interact with LockLint: source code annotations and the command-line interface.

- Source code annotations are assertions and `NOTE`s that you place in your source code to tell LockLint things it would not otherwise know. LockLint can verify certain assertions about the states of locks at specific points in your code, and annotations can be used to verify that locking behavior is correct or avoid unnecessary error warnings.

  See “Source Code Annotations” on page 281 for more information.

- Alternatively, you can use LockLint subcommands to load the relevant `.ll` files and make assertions. The important features of the subcommand interface are:
  - Using subcommands, you can exercise a few additional controls which have no corresponding annotations.
  - You can make a number of useful queries about the functions, variables, function pointers, and locks in your program.

  LockLint subcommands help you analyze your code and discover which variables are not consistently protected by locks. You may make assertions about which variables are supposed to be protected by a lock and which locks are supposed to be held whenever a function is called. Running the analysis with such assertions in place will show you where the assertions are violated.

  See “Command Reference” on page 297.

Most programmers report that they find source code annotations preferable to command-line subcommands. However, there is not necessarily a one-to-one correspondence between the two.

How to Use LockLint

This section is organized as follows:

Use LockLint to refine the set of assertions you maintain for the implementation of your system. Maintaining a rich set of assertions enables LockLint to validate existing and new source code as you work.

Using LockLint consists of two major steps:

- Compiling the source code to be analyzed, producing the LockLint database files (`.ll` files)
- Using the `lock_lint` command to run a LockLint session

Figure 24-1 shows a general scheme for using LockLint:

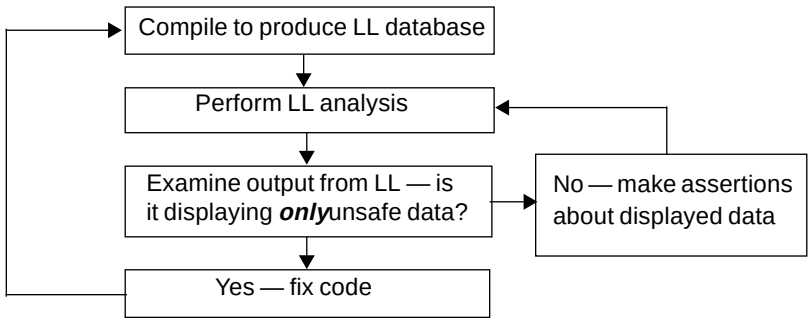


Figure 24-1 LockLint Control Flow

24

Managing LockLint's Environment

The LockLint interface is provided through the `lock_lint` command, which is executed in a shell. By default, LockLint uses `$SHELL`. Alternatively, LockLint can execute any shell by specifying the shell to use on the `lock_lint start` command. For example:

```
% lock_lint start /bin/ksh
```

starts a LockLint session up in the Korn shell.

LockLint creates an environment variable called `LL_CONTEXT`, which is visible in the child shell. If you are using a shell that provides for initialization, you can arrange to have `lock_lint` source a `.ll_init` file in your home directory, and then execute a `.ll_init` file in the current directory if it exists. If you use `csh`, you can do this by inserting the following code into your `.cshrc` file:

```
if ($?LL_CONTEXT) then
    if ( -x $(HOME)/.ll_init ) source $(HOME)/.ll_init
endif
```

It is better *not* to have your `.cshrc` source the file in your current working directory, since others may want to run LockLint on those same files, and they may not use the same shell you do. Since you are the only one who is going to use your `$(HOME)/.ll_init`, you *should* source that one, so that you can change the prompt and define aliases for use during your LockLint session. The following version of `~/.ll_init` does this for `csh`:

```
# Cause analyze subcommand to save state before analysis.
alias analyze "lock_lint save before analyze;\
    lock_lint analyze"
# Change prompt to show we are in lock_lint.
set prompt="lock_lint~$prompt"
```

(Also see "start" on page 326.)

When executing subcommands, remember that you can use pipes, redirection, backticks, and so on to accomplish your aims. For example, the following command asserts that lock `foo` protects all global variables (the formal name for a global variable begins with a colon):

```
% lock_lint assert foo protects `lock_lint vars | grep ^:`
```

In general, the subcommands are set up for easy use with filters such as `grep` and `sed`. This is particularly true for `vars` and `funcs`, which put out a single line of information, for each variable or function. Each line contains the attributes (defined and derived) for that variable or function. The following example shows which members of struct `bar` are supposed to be protected by member `lock`:

```
% lock_lint vars -a `lock_lint members bar` | grep =bar::lock
```

Since you are using a shell interface, a log of user commands can be obtained by using the shell's history function (the history level may need to be made large in the `.ll_init` file).

Temporary Files

LockLint puts temporary files in `/var/tmp` unless `$TMPDIR` is set.

Makefile Rules

To modify your makefile to produce `.ll` files, first use the rule for creating a `.o` from a `.c` to write a rule to create a `.ll` from a `.c`. For example, from:

```
# Rule for making .o from .c in ../src.
%.o: ../src/%.c
    $(COMPILE.c) -o $@ $<
```

you might write:

```
# Rule for making .ll from .c in ../src.
%.ll: ../src/%.c
    cc $(CFLAGS) $(CPPFLAGS) $(FOO) $<
```

If you use a suffix rule, you will need to define `.ll` as a suffix. For that reason some prefer to use % rules.

If the appropriate `.o` files are contained in a make variable `FOO_OBJS`, you can create `FOO_LLS` with the line:

```
FOO_LLS = ${FOO_OBJS:%.o=%.ll}
```

or, if they are in a subdirectory `ll`:

```
FOO_LLS = ${FOO_OBJS:%.o=ll/%.ll}
```

If you want to keep the `.ll` files in subdirectory `ll/`, you can have the makefile automatically create this file with the label:

```
.INIT:
    @if [ ! -d ll ]; then mkdir ll; fi
```

Compiling Code

For LockLint to analyze your source code, you must first compile it using the `-Zll` option of the SunSoft ANSI C compiler. The SunSoft ANSI C compiler will then produce the LockLint database files (`.ll` files), one for each `.c` file compiled. Later you load the `.ll` files into LockLint with the `load` subcommand.

LockLint sometimes needs a simpler view of the code to return meaningful results during analysis. To allow you to provide this simpler view, the `-Zll` option automatically defines the macro `__lock_lint`.

Further discussions of the likely uses of `__lock_lint` can be found in “Limitations of LockLint” on page 277.

LockLint Subcommands

The interface to LockLint consists of a set of subcommands that can be specified with the `lock_lint` command:

```
lock_lint [subcommand]
```

In this example *subcommand* is one of a set of subcommands used to direct the analysis of the source code for data races and deadlocks. More information about subcommands can be found in “Command Reference” on page 297.

Starting and Exiting LockLint

The first subcommand of any LockLint session must be `start`, which starts a subshell of your choice with the appropriate LockLint context. Since a LockLint session is started within a subshell, you exit by exiting that subshell. For example, to exit LockLint when using the C shell, use the command `exit`.

Setting the Tool State

LockLint’s *state* consists of the set of databases loaded and the specified assertions. Iteratively modifying that state and rerunning the analysis can provide optimal information on potential data races and deadlocks. Since the analysis can be done only once for any particular state, the `save`, `restore`, and `refresh` subcommands are provided as a means to reestablish a state, modify that state, and retry the analysis.

Suggested Approach for Checking an Application

Here are the basic steps involved in checking an application:

1. **Annotate your source code and compile it to create `.ll` files.**
   See “Source Code Annotations” on page 281.
2. **Load the `.ll` files using the `load` subcommand.**
3. **Make assertions about locks protecting functions and variables using the `assert` subcommand.**

Note – These specifications may also be conveyed using source code annotations. See “Source Code Annotations” on page 281.

4. **Make assertions about the order in which locks should be acquired in order to avoid deadlocks, using the `assert order` subcommand.**

Note – These specifications may also be conveyed using source code annotations. See “Source Code Annotations” on page 281.

5. **Check that LockLint has the right idea about which functions are roots.**
   If the `funcs -o` subcommand does not show a root function as root, use the `declare root` subcommand to fix it. If `funcs -o` shows a non-root function as root, it’s likely that the function should be listed as a function target using the `declare ... targets` subcommand. See “declare root func” on page 310 for a discussion of root functions.

6. **Describe any hierarchical lock relationships (if you have any—they are rare) using the `rwlock` subcommand.**

Note – These specifications may also be conveyed using source code annotations. See “Source Code Annotations” on page 281.

7. **Ignore any functions or variables you want to exclude from the analysis using the `ignore` subcommand.**
   Be conservative in your use of the `ignore` command. Make sure you should not be using one of the source code annotations instead (for example, `NO_COMPETING_THREADS_NOW`).

8. **Run the analysis using the `analyze` subcommand.**

9. **Deal with the errors.**
   This may involve modifying the source using `#ifdef __lock_lint` (see “Limitations of LockLint” on page 277) or adding source code annotations to accomplish steps 3, 4, 6, and 7 (see “Source Code Annotations” on page 281).

   Restore LockLint to a preanalysis state and rerun the analysis as necessary.

Note – It is best to handle the errors in order. Otherwise, problems with locks not being held on entry to a function, or locks being released while not held, can cause lots of misleading messages about variables not being properly protected.

10. **Run the analysis using the** `analyze -v` **subcommand and repeat the above step.**

11. **When the errors from the** `analyze` **subcommand are gone, check for variables that are not properly protected by any lock.**
    Use the command:

    ```
    lock_lint vars -h | fgrep \*
    ```

    Rerun the analysis using appropriate assertions to find out where the variables are being accessed without holding the proper locks.

    Remember that you cannot run `analyze` twice for a given state, so it will probably help to save the state of LockLint using the save subcommand before running `analyze`. Then restore that state using `refresh` or `restore` before adding more assertions. You may want to set up an alias for `analyze` that automatically does a `save` before analyzing.

Program Knowledge Management

LockLint acquires its information on the sources to be analyzed with a set of databases produced by the C compiler. The LockLint database for each source file is stored in a separate file. To analyze a set of source files, use the `subcommandsload` subcommand to load their associated database files. The `files` subcommand can be used to display a list of the source files represented by the loaded database files. Once a file is loaded, LockLint knows about all the functions, global data, and external functions referenced in the associated source files.

Function Management

As part of the analysis phase, LockLint builds a *call graph* for all the loaded sources. Information about the functions defined is available via the `funcs` subcommand. It is extremely important for a meaningful analysis that LockLint have the correct call graph for the code to be analyzed.

All functions that are not called by any of the loaded files are called *root* functions. You may want to treat certain functions as root functions even though they are called within the loaded modules (for example, the function is an entry point for a library that is also called from within the library). Do this by using the `declare root` subcommand. You may also remove functions from the call graph by issuing the `ignore` subcommand.

LockLint knows about all the references to *function pointers* and most of the assignments made to them. Information about the function pointers in the currently loaded files is available through the `funcptrs` subcommand. Information about the calls made via function pointers is available via the `pointer calls` subcommand. If there are function pointer assignments that LockLint could not discover, they may be specified with the `declare ... targets` subcommand.

By default, LockLint tries to examine all possible execution paths. If the code uses function pointers, it's possible that many of the execution paths will not actually be followed in normal operation of the code. This can result in the reporting of deadlocks that do not really occur. To prevent this, use the `lock_lint disallow` and `reallow` subcommands to inform LockLint of execution paths that will never occur. To print out existing constraints, use the `reallows` and `disallows` subcommands.

Variable Management

The LockLint database also contains information about all global variables accessed in the source code. Information about these variables is available via the `vars` subcommands.

One of LockLint's jobs is to determine if variable accesses are consistently protected. If you are unconcerned about accesses to a particular variable, you can remove it from consideration by using the `ignore` subcommand.

You may also consider using one of the source code annotations shown in Table 24-5, as appropriate.

Table 24-5 Variable Management Source Code Annotations

`SCHEME_PROTECTS_DATA`
`read_only_data`

Table 24-5 Variable Management Source Code Annotations *(Continued)*

`data_readable_without_lock`
`now_invisible_to_other_threads`
`now_visible_to_other_threads`

For more information, see “Source Code Annotations” on page 281.

Lock Management

Source code annotations are an efficient way to refine the assertions you make about the locks in your code. There are three types of assertions: *protection*, *order*, and *side effects*.

Protection assertions state what is protected by a given lock. For example, the source code annotations shown in Table 24-6 can be used to assert how data is protected.

Table 24-6 Data Protection Source Code Annotations

`MUTEX_PROTECTS_DATA`
`RWLOCK_PROTECTS_DATA`
`SCHEME_PROTECTS_DATA`
`DATA_READABLE_WITHOUT_LOCK`
`RWLOCK_COVERS_LOCK`

A variation of the `assert` subcommand is used to assert that a given lock protects some piece of data or a function. Another variation, `assert ... covers`, asserts that a given lock protects another lock; this is used for hierarchical locking schemes.

Order assertions specify the order in which the given locks must be acquired. The source code annotation `LOCK_ORDER` or the `assert order` subcommand can be used to specify lock orderings.

Side effect assertions state that a function has the side effect of releasing or acquiring a given lock. Use the source code annotations listed in Table 24-7.

Table 24-7 Functions With Locking Side Effects

`MUTEX_ACQUIRED_AS_SIDE_EFFECT`
`READ_LOCK_ACQUIRED_AS_SIDE_EFFECT`
`WRITE_LOCK_ACQUIRED_AS_SIDE_EFFECT`
`LOCK_RELEASED_AS_SIDE_EFFECT`
`LOCK_UPGRADED_AS_SIDE_EFFECT`
`LOCK_DOWNGRADED_AS_SIDE_EFFECT`
`NO_COMPETING_THREADS_AS_SIDE_EFFECT`
`COMPETING_THREADS_AS_SIDE_EFFECT`

You can also use the `assert side effect` subcommand to specify side effects. In some cases you may want to make side effect assertions about an external function and the lock is not visible from the loaded module (for example, it is static to the module of the external function). In such a case, you can "create" a lock by using a form of the `declare` subcommand.

Analysis

LockLint's primary role is to report on lock usage inconsistencies that may lead to data races and deadlocks. The analysis of lock usage occurs when you use the `analyze` subcommand. The result is a report on the following problems:

- Functions that produce side effects on locks or violate assertions made about side effects on locks (for example, a function that changes the state of a mutex lock from locked to unlocked). The most common unintentional side effect occurs when a function acquires a lock on entry, and then fails to release it at some return point. That path through the function is said to acquire the lock as a side effect. This type of problem may lead to both data races and deadlocks.
- Functions that have inconsistent side effects on locks (that is, different paths through the function) yield different side effects. This may be a limitation of LockLint (see "Limitations of LockLint" on page 277) and a common cause

of errors. LockLint cannot handle such functions. It always reports them as errors and does not correctly interpret them. For example, one of the returns from a function may forget to unlock a lock acquired in the function.

- Violations of assertions about which locks should be held upon entry to a function. This problem may lead to a data race.
- Violations of assertions that a lock should be held when a variable is accessed. This problem may lead to a data race.
- Violations of assertions that specify the order in which locks are to be acquired. This problem may lead to a deadlock.
- Check to make sure the same, or asserted, mutex lock is used for all waits on a particular condition variable.
- Miscellaneous problems related to analysis of the source code in relation to assertions and locks.

Postanalysis Queries

After analysis, you can use LockLint subcommands for:

- Finding additional locking inconsistencies.
- Forming appropriate `declare`, `assert`, and `ignore` subcommands. These can be specified after you've restored LockLint's state, prior to rerunning the analysis.

One such subcommand is `order`, which you can use to make inquiries about the order in which locks have been acquired. This information is particularly useful in understanding lock ordering problems and making assertions about those orders so that LockLint can more accurately diagnose potential deadlocks.

Another such subcommand is `vars`. The `vars` subcommand reports which locks are consistently held when a variable is read or written (if any). This information can be useful in determining the protection conventions in code where the original conventions were never documented, or the documentation has become outdated.

Limitations of LockLint

There are limitations to LockLint's powers of analysis. At the root of many of its difficulties is the fact that LockLint doesn't know the values of your variables.

LockLint solves some of these problems by ignoring the likely cause or making simplifying assumptions. Some other problems can be avoided by using conditionally compiled code in the application. Towards this end, the compiler always defines the preprocessor macro `__lock_lint` when the `-Zll` switch is used. You can use this macro to make your code less ambiguous.

LockLint has trouble deducing:

- Which functions your function pointers point to. There are some assignments LockLint cannot deduce (see "declare" on page 308). The `declare` subcommand can be used to add new possible assignments to the function pointer.

  When LockLint sees a call through a function pointer, it will test that call path for every possible value of that function pointer. If you know or suspect that some calling sequences is never executed, use the `disallow` and `reallow` subcommands to specify which sequences will be.

- Whether or not you locked a lock in code like this:

```
if (x) pthread_mutex_lock(&lock1);
```

  In this case, two execution paths are created, one holding the lock, and one not holding the lock, which will probably cause the generation of a side effect message at the `unlock` call. You may be able to work around this problem by using the `__lock_lint` macro to force LockLint to treat a lock as unconditionally taken. For example:

```
#ifdef __lock_lint
pthread_mutex_lock(&lock1);
#else
if (x) pthread_mutex_lock(&lock1);
#endif
```

Note that LockLint has no problem analyzing code like this:

```
if (x) {
    pthread_mutex_lock(&lock1);
    foo();
    pthread_mutex_unlock(&lock1);
}
```

In this case, there is only one execution path, along which the lock is held and released, causing no side effects.

- Whether or not a lock was acquired in code like this:

```
rc = pthread_mutex_trylock(&lock1);
if (rc) ...
```

- Which lock is being locked in code like this:

```
pthread_mutex_t* lockp;
pthread_mutex_lock(lockp);
```

In such cases, the lock call is ignored.

- Which variables and locks are being used in code like this:

```
struct foo* p;
pthread_mutex_lock(p->lock);
p->bar = 0;
```

(So that it uses the names `foo::lock` and `foo::bar` — see “Inversions” on page 331.)

- Which element of an array is being accessed. This is treated analogously to the previous case; the index is ignored.
- Anything about `longjmps`.
- When you would exit a loop or break out of a recursion (so it just stops proceeding down a path as soon as it finds itself looping or after one recursion).

Some other LockLint difficulties:

- LockLint only analyzes the use of mutex locks and readers-writer locks. LockLint performs limited consistency checks of mutex locks as used with condition variables. However, semaphores and condition variables are not recognized as locks by LockLint. Even with this analysis, there are limits to what LockLint can make sense of.
- There are situations where LockLint thinks two different variables are the same variable, or that a single variable is two different variables. (“Inversions” on page 331.)
- It is possible to share automatic variables between threads (via pointers), but LockLint assumes that automatics are unshared, and generally ignores them (the only situation in which they are of interest to LockLint is when they are function pointers).
- LockLint complains about any functions that are not consistent in their side effects on locks. `#ifdef`’s and assertions must be used to give LockLint a simpler view of functions that may or may not have such a side effect.

During analysis, LockLint may produce messages about a lock operation called `rw_upgrade`. Such a call does not really exist, but LockLint rewrites code like

```
if (rw_tryupgrade(&lock1)) { ...  }
```

as

```
if () { rw_tryupgrade(&lock1); ...  }
```

such that, wherever `rw_tryupgrade()` occurs, LockLint always assumes it succeeds.

One of the errors LockLint flags is an attempt to acquire a lock that is already held. However, if the lock is unnamed (e.g. `foo::lock`), this error is suppressed, since the name refers not to a single lock but to a set of locks. However, if the unnamed lock always refers to the same lock, use the `declare one` subcommand so that LockLint can report this type of potential deadlock.

If you have constructed your own locks out of these locks (for example, recursive mutexes are sometimes built from ordinary mutexes), LockLint will not know about them. Generally you can use `#ifdef` to make it appear to

LockLint as though an ordinary mutex is being manipulated. For recursive locks, use an unnamed lock for this deception, since errors won't be generated when it is recursively locked. For example:

```
void get_lock() {
     #ifdef __.
           struct bogus *p;
           pthread_mutex_lock(p->lock);
     #else
           <the real recursive locking code>
     #endif
}
```

Source Code Annotations

This section is organized as follows:

Assertions and NOTEs

An *annotation* is some piece of text inserted into your source code. (Generally, it has some semantic information that normal analyses cannot deduce.) You use annotations to tell LockLint things about your program that it cannot deduce for itself, either to keep it from excessively flagging problems or to have LockLint test for certain conditions. Annotations also serve to document code, in much the same way that comments do.

Annotations are similar to some of the LockLint subcommands described in "Command Reference" on page 297. In general, it's preferable to use source code annotations over these subcommands, as explained in "Why Use Source Code Annotations?" on page 282.

There are two types of ssource code annotations: *assertions* and NOTEs.

24

Why Use Source Code Annotations?

There are several reasons to use source code annotations. In many cases, such annotations are preferable to using a script of LockLint subcommands.

- Annotations, being mixed in with the code that they describe, will generally be better maintained than a script of LockLint subcommands.
- With annotations, you can make assertions about lock state at any point within a function—wherever you put the assertion is where the check occurs. With subcommands, the finest granularity you can achieve is to check an assertion on entry to a function.
- Functions mentioned in subcommands can change. If someone changes the name of a function from `func1` to `func2`, a subcommand mentioning `func1` will fail (or worse, might work but do the wrong thing, if a different function is given the name `func1`).
- Some annotations, such as `NOTE(NO_COMPETING_THREADS_NOW)`, have no subcommand equivalents.
- Annotations provide a good way to document your program. In fact, even if you are not using LockLint often, annotations are worthwhile just for this purpose. For example, a header file declaring a variable can document what lock or convention protects the variable, or a function that acquires a lock and deliberately returns without releasing it can have that intent clearly declared in an annotation.

The Annotations Scheme

LockLint shares the source code annotations scheme with several other tools. When you install the SunSoft ANSI C Compiler, you also automatically install the file `/usr/lib/note/SUNW_SPRO-cc-ssbd`, which contains the names of all the annotations that LockLint understands. However, the SunSoft ANSI C Compiler also checks all the files in `/usr/lib/note` and `/opt/SUNWspro/SC4.0/note` for *all* valid annotations.

You may specify a location other than `/usr/lib/note` by setting the environment variable `NOTEPATH`, as in:

```
setenv NOTEPATH $NOTEPATH:other_location
```

To use source code annotations, include the file `note.h` in your source or header files:

```
#include <note.h>
```

LockLint NOTE*s*

NOTE *Syntax*

Many of the note-style annotations accept names—of locks or variables—as arguments. Names are specified using the syntax shown in Table 24-8.

Table 24-8 NOTE Syntax

Syntax	Meaning
Var	Named variable
Var.Mbr.Mbr...	Member of a named `struct`/`union` variable
Tag	Unnamed `struct/union` (with this tag)
Tag::*Mbr.Mbr...*	Member of an unnamed `struct/union`
Type	Unnamed `struct/union` (with this typedef)
Type::*Mbr.Mbr...*	Member of an unnamed `struct/union`

In C, structure tags and types are kept in separate namespaces, making it possible to have two different `struct`s by the same name as far as LockLint is concerned. When LockLint sees `foo::bar`, it will first look for a `struct` with *tag* `foo`; if it does not find one, it will look for a *type* `foo` and check that it represents a `struct`.

However, the proper operation of LockLint requires that a given variable or lock be known by exactly one name. Therefore *type* will be used only when no *tag* is provided for the `struct`, and even then only when the `struct` is defined as part of a `typedef`.

For example, `Foo` would serve as the type name in this example:

```
typedef struct { int a, b; } Foo;
```

These restrictions ensure that there is only one name by which the `struct` is known.

Name arguments do not accept general expressions. It is not valid, for example, to write:

```
NOTE(MUTEX_PROTECTS_DATA(p->lock, p->a p->b))
```

However, some of the annotations do accept expressions (rather than names); they are clearly marked.

In many cases an annotation accepts a list of names as an argument. Members of a list should be separated by white space. To simplify the specification of lists, a generator mechanism similar to that of many shells is understood by all annotations taking such lists. The notation for this is:

Prefix{A B ...}Suffix

where *Prefix*, *Suffix*, *A*, *B*, ... are nothing at all, or any text containing no white space. The above notation is equivalent to:

PrefixASuffix PrefixBSuffix ...

For example, the notation:

```
struct_tag::{a b c d}
```

is equivalent to the far more cumbersome text:

```
struct_tag::a struct_tag::b struct_tag::c struct_tag::d
```

This construct may be nested, as in:

```
foo::{a b.{c d} e}
```

Where an annotation refers to a lock or another variable, a declaration or definition for that lock or variable should already have been seen.

If a name for data represents a structure, it refers to all non-lock (mutex or readers-writer) members of the structure. If one of those members is itself a structure, then all of its non-lock members are implied, and so on. However, LockLint understands the abstraction of a condition variable and therefore does not break it down into its constituent members.

NOTE *and* _NOTE

The NOTE interface is used so that you can easily turn off annotations when they are not being parsed. The basic syntax of a note-style annotation is either:

NOTE(*NoteInfo*)

or:

_NOTE(*NoteInfo*)

The preferred use is NOTE rather than _NOTE. Header files that are to be used in multiple, unrelated projects, should use _NOTE to avoid conflicts. If NOTE has already been used, and you do not want to change, you should define some other macro (such as ANNOTATION) using _NOTE. For example, you might define an include file (say, annotation.h) that contains the following:

```
#define ANNOTATION _NOTE
#include <sys/note.h>
```

The *NoteInfo* that gets passed to the NOTE interface must syntactically fit one of the following:

NoteName

NoteName(*Args*)

NoteName is simply an identifier indicating the type of annotation. *Args* can be anything, so long as it can be tokenized properly and any parenthesis tokens are matched (so that the closing parenthesis can be found). Each distinct *NoteName* will have its own requirements regarding arguments.

This text uses NOTE to mean both NOTE and _NOTE, unless explicitly stated otherwise.

Where NOTE *May Be Used*

NOTE may be invoked only at certain well-defined places in source code:

- At the top level; that is, outside of all function definitions, type and `struct` definitions, variable declarations, and other constructs. For example:

```
struct foo { int a, b; mutex_t lock; };
NOTE(MUTEX_PROTECTS_DATA(foo::lock, foo))
bar() {...}
```

- At the top level within a block, among declarations or statements. Here too, the annotation must be outside of all type and `struct` definitions, variable declarations, and other constructs. For example:

```
foo() { ...; NOTE(...) ...; ...; }
```

  A note-style annotation is not a statement; `NOTE()` may not be used inside an `if`/`else`/`for`/`while` body unless braces are used to make a block. For example, the following causes a syntax error:

```
if (x)
    NOTE(...)
```

- At the top level within a `struct` or `union` definition, among the declarations. For example:

```
struct foo { int a; NOTE(...) int b; };
```

`NOTE()` may be used only in the locations described above. For example, the following are invalid:

```
a = b NOTE(...) + 1;
typedef NOTE(...) struct foo Foo;
for (i=0; NOTE(...) i<10; i++) ...
```

How Data Is Protected

The following annotations are allowed both outside and inside a function definition. Remember that any name mentioned in an annotation must already have been declared.

NOTE(MUTEX_PROTECTS_DATA(*Mutex*, *DataNameList*))

NOTE(RWLOCK_PROTECTS_DATA(*Rwlock*, *DataNameList*))

NOTE(SCHEME_PROTECTS_DATA("*description*", *DataNameList*))

The first two annotations tell LockLint that the lock should be held whenever the specified data is accessed.

The third annotation, SCHEME_PROTECTS_DATA, describes how data are protected if it does not have a mutex or readers-writer lock. The *description* supplied for the scheme is simply text and is not semantically significant; LockLint responds by ignoring the specified data altogether. You may make *description* anything you like.

Some examples will help show how these annotations are used. The first example is very simple, showing a lock that protects two variables:

```
mutex_t lock1;
int a,b;
NOTE(MUTEX_PROTECTS_DATA(lock1, a b))
```

In the next example, a number of different possibilities are shown. Some members of struct foo are protected by a static lock, while others are protected by the lock on foo. Another member of foo is protected by some convention regarding its use.

```
mutex_t lock1;
struct foo {
    mutex_t lock;
    int mbr1, mbr2;
    struct {
        int mbr1, mbr2;
        char* mbr3;
    } inner;
    int mbr4;
};
NOTE(MUTEX_PROTECTS_DATA(lock1, foo::{mbr1 inner.mbr1}))
NOTE(MUTEX_PROTECTS_DATA(foo::lock, foo::{mbr2 inner.mbr2}))
NOTE(SCHEME_PROTECTS_DATA("convention XYZ", inner.mbr3))
```

A datum can only be protected in one way. If multiple annotations about protection (not only these three but also READ_ONLY_DATA) are used for a single datum, later annotations silently override earlier annotations. This allows for easy description of a structure in which all but one or two members are protected in the same way. For example, most of the members of `struct BAR` below are protected by the lock on `struct foo`, but one is protected by a global lock.

```
mutex_t lock1;
typedef struct {
    int mbr1, mbr2, mbr3, mbr4;
} BAR;
NOTE(MUTEX_PROTECTS_DATA(foo::lock, BAR))
NOTE(MUTEX_PROTECTS_DATA(lock1, BAR::mbr3))
```

Read-Only Variables

`NOTE(READ_ONLY_DATA(`*DataNameList*`))`

This annotation is allowed both outside and inside a function definition. It tells LockLint how data should be protected. In this case, it tells LockLint that the data should only be read, and not written.

Note – No error will be signaled if read-only data is written while it is considered invisible. Data is considered *invisible* when other threads cannot access it; for example, if other threads do not know about it.

This annotation is often used with data that is initialized and never changed thereafter. If the initialization is done at runtime before the data is visible to other threads, use annotations to let LockLint know that the data is invisible during that time.

LockLint knows that `const` data is read-only.

Allowing Unprotected Reads

`NOTE(DATA_READABLE_WITHOUT_LOCK(`*DataNameList*`))`

This annotation is allowed both outside and inside a function definition. It informs LockLint that the specified data may be read without holding the protecting locks. This is useful with an atomically readable datum that stands alone (as opposed to a set of data whose values are used together), since it is valid to peek at the unprotected data if you do not intend to modify it.

Hierarchical Lock Relationships

`NOTE(RWLOCK_COVERS_LOCKS(`*RwlockName*`, `*LockNameList*`))`

This annotation is allowed both outside and inside a function definition. It tells LockLint that a hierarchical relationship exists between a readers-writer lock and a set of other locks. Under these rules, holding the cover lock for write access affords a thread access to all data protected by the covered locks. Also, a thread must hold the cover lock for read access whenever holding any of the covered locks.

Using a readers-writer lock to cover another lock in this way is simply a convention; there is no special lock type. However, if LockLint is not told about this coverage relationship, it assumes that the locks are being used according to the usual conventions and generates errors.

The following example specifies that member `lock` of unnamed `foo` structures covers member `lock` of unnamed `bar` and `zot` structures:

```
NOTE(RWLOCK_COVERS_LOCKS(foo::lock, {bar zot}::lock))
```

Functions With Locking Side Effects

`NOTE(MUTEX_ACQUIRED_AS_SIDE_EFFECT(`*MutexExpr*`))`

`NOTE(READ_LOCK_ACQUIRED_AS_SIDE_EFFECT(`*RwlockExpr*`))`

`NOTE(WRITE_LOCK_ACQUIRED_AS_SIDE_EFFECT(`*RwlockExpr*`))`

`NOTE(LOCK_RELEASED_AS_SIDE_EFFECT(`*LockExpr*`))`

`NOTE(LOCK_UPGRADED_AS_SIDE_EFFECT(`*RwlockExpr*`))`

`NOTE(LOCK_DOWNGRADED_AS_SIDE_EFFECT(`*RwlockExpr*`))`

`NOTE(NO_COMPETING_THREADS_AS_SIDE_EFFECT)`

`NOTE(COMPETING_THREADS_AS_SIDE_EFFECT)`

These annotations are allowed only inside a function definition. Each tells LockLint that the function has the specified side effect on the specified lock—that is, that the function deliberately leaves the lock in a different state

on exit than it was in when the function was entered. In the case of the last two of these annotations, the side effect is not about a lock but rather about the state of concurrency.

When stating that a readers-writer lock is acquired as a side effect, it you must specify whether the lock was acquired for read or write access.

A lock is said to be *upgraded* if it changes from being acquired for read-only access to being acquired for read / write access. *Downgraded* means a transformation in the opposite direction.

LockLint analyzes each function for its side effects on locks (and concurrency). Ordinarily, LockLint expects that a function will have no such effects; if the code has such effects intentionally, you must inform LockLint of that intent using annotations. If it finds that a function has different side effects from those expressed in the annotations, an error message will result.

The annotations described in this section refer generally to the function's characteristics and not to a particular point in the code. Thus, these annotations are probably best written at the top of the function. There is, for example, no difference (other than readability) between this:

```
foo() {
    NOTE(MUTEX_ACQUIRED_AS_SIDE_EFFECT(&foo))
    ...
    if (x && y) {
        ...
    }
}
```

and this:

```
foo() {
    ...
    if (x && y) {
    NOTE(MUTEX_ACQUIRED_AS_SIDE_EFFECT(&foo))
        ...
    }
}
```

If a function has such a side effect, the effect should be the same on every path through the function. LockLint will complain about and refuse to analyze paths through the function that have side effects other than those specified.

Single-Threaded Code

`NOTE(COMPETING_THREADS_NOW)`

`NOTE(NO_COMPETING_THREADS_NOW)`

These two annotations are allowed only inside a function definition. The first annotation tells LockLint that after this point in the code, other threads exist that might try to access the same data that this thread will access. The second function specifies that this is no longer the case; either no other threads are running or whatever threads are running will not be accessing data that this thread will access. While there are no competing threads, LockLint does not complain if the code accesses data without holding the locks that ordinarily protect that data.

These annotations are useful in functions that initialize data without holding locks before starting up any additional threads. Such functions may access data without holding locks, after waiting for all other threads to exit. So one might see something like this:

```
main() {
    <initialize data structures>
    NOTE(COMPETING_THREADS_NOW)
    <create several threads>
    <wait for all of those threads to exit>
    NOTE(NO_COMPETING_THREADS_NOW)
    <look at data structures and print results>
}
```

Note – LockLint realizes that when `main()` starts, no other threads are running.

LockLint does not issue a warning if, during analysis, it encounters a `COMPETING_THREADS_NOW` annotation when it already thinks competing threads are present. The condition simply nests. No warning is issued because the annotation may mean different things in each use (that is the notion of

which threads compete may differ from one piece of code to the next). On the other hand, a `NO_COMPETING_THREADS_NOW` annotation that does not match a prior `COMPETING_THREADS_NOW` (explicit or implicit) will cause a warning.

Unreachable Code

`NOTE(NOT_REACHED)`

This annotation is allowed only inside a function definition. It tells LockLint that a particular point in the code cannot be reached, and therefore LockLint should ignore the condition of locks held at that point. This annotation need not be used after every call to `exit()`, for example, the way the `lint` annotation `/* NOTREACHED */` is used. Simply use it in *definitions* for `exit()` and the like (primarily in LockLint libraries), and LockLint will figure out that code following calls to such functions is not reached. This annotation should seldom appear outside LockLint libraries. An example of its use (in a LockLint library) would be:

```
exit(int code) { NOTE(NOT_REACHED) }
```

Lock Order

`NOTE(LOCK_ORDER(`*LockNameList*`))`

This annotation, which is allowed either outside or inside a function definition, specifies the order in which locks should be acquired. It is similar to the `assert order` and `order` subcommands. See “Command Reference” on page 297.

To avoid deadlocks, LockLint assumes that whenever multiple locks must be held at once they are always acquired in a well-known order. If LockLint has been informed of such orderings using this annotation, an informative message will be produced whenever the order is violated.

This annotation may be used multiple times, and the semantics will be combined appropriately. For example, given the annotations

```
NOTE(LOCK_ORDER(a b c))
NOTE(LOCK_ORDER(b d))
```

LockLint will deduce the ordering:

```
NOTE(LOCK_ORDER(a d))
```

It is not possible to deduce anything about the order of `c` with respect to `d` in this example.

If a cycle exists in the ordering, an appropriate error message will be generated.

Variables Invisible to Other Threads

```
NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(DataExpr, ...))
NOTE(NOW_VISIBLE_TO_OTHER_THREADS(DataExpr, ...))
```

These annotations, which are allowed only within a function definition, tell LockLint whether or not the variables represented by the specified expressions are *visible* to other threads; that is, whether or not other threads could access the variables.

Another common use of these annotations is to inform LockLint that variables that it would ordinarily assume are visible are in fact not visible, because no other thread has a pointer to it. This frequently occurs when allocating data off the heap—you can safely initialize the structure without holding a lock, since no other thread can yet see the structure.

```
Foo* p = (Foo*) malloc(sizeof(*p));
NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*p))
p->a = bar;
p->b = zot;
NOTE(NOW_VISIBLE_TO_OTHER_THREADS(*p))
add_entry(&global_foo_list, p);
```

Calling a function never has the side effect of making variables visible or invisible. Upon return from the function, all changes in visibility caused by the function are reversed.

Assuming Variables Are Protected

```
NOTE(ASSUMING_PROTECTED(DataExpr, ...))
```

This annotation, which is allowed only within a function definition, tells LockLint that this function assumes that the variables represented by the specified expressions are protected in one of the following ways:

- The appropriate lock is held for each variable
- The variables are invisible to other threads
- There are no competing threads when the call is made

LockLint will issue an error if none of these conditions is true.

```
f(Foo* p, Bar* q) {
    NOTE(ASSUMING_PROTECTED(*p, *q))
    p->a++;
    ...
}
```

Assertions Recognized by LockLint

Locklint recognizes some assertions as relevant to the state of threads and locks. (For more information, see the `assert` man page.)

Assertions may be made only within a function definition, where a statement is allowed.

Note – `ASSERT()` is used in kernel and driver code, whereas `assert()` is used in user (application) code. For simplicity's sake, this document uses `assert()` to refer to either one, unless explicitly stated otherwise.

Making Sure All Locks Are Released

```
assert(NO_LOCKS_HELD);
```

LockLint recognizes this assertion to mean that, when this point in the code is reached, no locks should be held by the thread executing this test. Violations will be reported during analysis. A routine that blocks might want to use such an assertion to ensure that no locks are held when a thread blocks or exits.

The assertion also clearly serves as a reminder to someone modifying the code that any locks acquired must be released at that point.

It is really only necessary to use this assertion in leaf-level functions that block. If a function blocks only inasmuch as it calls another function that blocks, the caller need not contain this assertion as long as the callee does. Therefore this assertion will probably see its heaviest use in versions of libraries (for example, `libc`) written specifically for LockLint (like lint libraries).

The file `synch.h` defines `NO_LOCKS_HELD` as 1 if it has not already been otherwise defined, causing the assertion to succeed; that is, the assertion is effectively ignored at runtime. You can override this default runtime meaning by defining `NO_LOCKS_HELD` before you include either `note.h` or `synch.h` (which may be included in either order). For example, if a body of code uses only two locks called `a` and `b`, the following definition would probably suffice:

```
#define NO_LOCKS_HELD (!MUTEX_HELD(&a) && !MUTEX_HELD(&b))
#include <note.h>
#include <synch.h>
```

Doing so will not affect LockLint's testing of the assertion; that is, LockLint will still complain if *any* locks are held (not just `a` or `b`).

Making Sure No Other Threads Are Running

```
assert(NO_COMPETING_THREADS);
```

LockLint recognizes this assertion to mean that, when this point in the code is reached, no other threads should be competing with the one running this code. Violations (based on info provided by certain `NOTE`-style assertions) are reported during analysis. Any function that accesses variables without holding their protecting locks (operating under the assumption that no other relevant threads are out there touching the same data), should be so marked.

By default, this assertion is ignored at runtime—that is, it always succeeds. No generic runtime meaning for `NO_COMPETING_THREADS` is possible, since the notion of which threads compete involves knowledge of the application. For example, a driver might make such an assertion to say that no other threads are running in this driver for the same device. Because no generic meaning is possible, `synch.h` will define `NO_COMPETING_THREADS` as 1 if it has not already been otherwise defined.

However, you can override the default meaning for NO_COMPETING_THREADS by defining it before including either note.h or synch.h (which may be included in either order). For example, if the program keeps a count of the number of running threads in a variable called num_threads, the following definition might suffice:

```
#define NO_COMPETING_THREADS (num_threads == 1)
#include <note.h>
#include <synch.h>
```

Doing so will not affect LockLint's testing of the assertion.

Asserting Lock State

```
assert(MUTEX_HELD(lock_expr) && ...);
```

This assertion is widely used within the kernel. It performs runtime checking if assertions are enabled. The same capability exists in user code.

This code does roughly the same thing during LockLint analysis as it does when the code is actually run with assertions enabled; that is, it reports an error if the executing thread does not hold the lock as described.

Note – The threads library performs a weaker test, only checking that *some* thread holds the lock. LockLint performs the stronger test.

LockLint recognizes the use of MUTEX_HELD(), RW_READ_HELD(), RW_WRITE_HELD(), and RW_LOCK_HELD() macros, and negations thereof. Such macro calls may be combined using the && operators. For example, the following assertion will cause LockLint to check that a mutex is not held and that a readers-writer lock is write-held:

```
assert(p && !MUTEX_HELD(&p->mtx) && RW_WRITE_HELD(&p->rwlock));
```

LockLint also recognizes expressions like:

```
MUTEX_HELD(&foo) == 0
```

Command Reference

This section is organized as follows:

Subcommand Summary

Table 24-9 identifies the LockLint subcommands.

Table 24-9 LockLint Subcommands

Subcommand	Effect
`analyze`	Tests the loaded files for lock inconsistencies; also validates against assertions
`assert`	Specifies what LockLint should expect to see regarding accesses and modifications to locks and variables
`declare`	Passes information to LockLint that it cannot deduce
`disallow`	Excludes the specified calling sequence in the analysis
`disallows`	Lists the calling sequences that are excluded from the analysis
`files`	Lists the source code files loaded via the `load` subcommand
`funcptrs`	Lists information about function pointers
`funcs`	Lists information about specific functions
`help`	Provides information about the specified keyword
`ignore`	Excludes the specified functions and variables from analysis
`load`	Specifies the `.ll` files to be loaded
`locks`	Lists information about locks
`members`	Lists members of the specified struct

Table 24-9 LockLint Subcommands *(Continued)*

Subcommand	Effect
`order`	Shows information about the order in which locks are acquired
`pointer calls`	Lists calls made through function pointers
`reallow`	Allows exceptions to the `disallow` subcommand
`reallows`	Lists the calling sequences reallowed through the `reallow` subcommand
`refresh`	Restores and resaves the latest saved state
`restore`	Restores the latest saved state
`save`	Saves the current state on a stack
`saves`	Lists the states saved on the stack through the `save` subcommand
`start`	Starts a LockLint session
`sym`	Lists the fully qualified names of functions and variables associated with the specified name
`unassert`	Removes some assertions specified through the `assert` subcommand
`vars`	Lists information about variables

Many LockLint subcommands require you to specify names of locks, variables, pointers, and functions. In C, it is possible for names to be ambiguous. See “Naming Conventions” on page 299 for details on specifying names to LockLint subcommands.

Exit Status of LockLint Subcommands

Table 24-10 lists the exit status of LockLint subcommands.

*Table 24-10*Subcommand Exit Status

Value	Meaning
0	Normal
1	System error
2	User error, such as incorrect options or undefined name

Table 24-10 Subcommand Exit Status *(Continued)*

Value	Meaning
3	Multiple errors
5	LockLint detected error: violation of an assertion, potential data race or deadlock may have been found, unprotected data references, and so on.
10	Licensing error

Naming Conventions

Many LockLint subcommands require you to specify names of locks, variables, pointers, and functions. In C, it is possible for names to be ambiguous; for example, there may be several variables named `foo`, one of them `extern` and others `static`.

The C language does not provide a way of referring to ambiguously named variables that are hidden by the scoping rules. In LockLint, however, a way of referring to such variables is needed. Therefore, every symbol in the code being analyzed is given a formal name, a name that LockLint uses when referring to the symbol. Table 24-11 lists some examples of formal names for a function.

Table 24-11 LockLint Formal Names of Functions

Formal Name	Meaning
`:func`	`extern function`
`file:func`	`static function`

Table 24-12 lists the formal names for a variable, depending on its use as a lock, a pointer, or an actual variable.

*Table 24-12*LockLint Formal Names of Variables

Formal Name	Meaning
`:var`	`extern` variable
`file:var`	`static` variable with file scope
`:func/var`	Variable defined in an `extern` function
`file:func/var`	Variable defined in a `static` function
`tag::mbr`	Member of an unnamed `struct`
`file@line::mbr`	Member of an unnamed, untagged `struct`

In addition, any of these may be followed by an arbitrary number of `.mbr` specifications to denote members of a structure.

Table 24-13 contains some examples of the LockLint naming scheme.

*Table 24-13*Examples of LockLint Naming Conventions

Example	Meaning
`:bar`	External variable or function `bar`
`:main/bar`	`static` variable `bar` that is defined within `extern` function `main`
`zot.c:foo/bar.zot`	Member `zot` of static variable `bar`, which is defined within static function `foo` in file `zot.c`
`foo::bar.zot.bim`	Member `bim` of member `zot` of member `bar` of a `struct` with tag `foo`, where no name is associated with that instance of the `struct` (it was accessed through a pointer)

While LockLint refers to symbols in this way, you are not required to. You may use as little of the name as is required to unambiguously identify it. For example, you could refer to `zot.c:foo/bar` as `foo/bar` as long as there is only one function `foo` defining a variable `bar`. You can even refer to it simply as `bar` as long as there is no other variable by that name.

C allows the programmer to declare a structure without assigning it a tag. When you use a pointer to such a structure, LockLint must make up a tag by which to refer to the structure. It generates a tag of the format

filename@line_number. For example, if you declare a structure without a tag at line 42 of file `foo.c`, and then refer to member `bar` of an instance of that structure using a pointer, as in:

```
typedef struct { ...  } foo;
foo *p;
func1() { p->bar = 0; }
```

LockLint sees that as a reference to `foo.c@42::bar`.

Because members of a `union` share the same memory location, LockLint treats all members of a `union` as the same variable. This is accomplished by using a member name of `%` regardless of which member is accessed. Since bit fields typically involve sharing of memory between variables, they are handled similarly: `%` is used in place of the bit field member name.

When you list locks and variables, you are only seeing those locks and variables that are actually used within the code represented by the `.ll` files. No information is available from LockLint on locks, variables, pointers, and functions that are declared but not used. Likewise, no information is available for accesses through pointers to simple types, such as this one:

```
int *ip = &i;
*ip = 0;
```

When simple names (for example, `foo`) are used, there is the possibility of conflict with keywords in the subcommand language. Such conflicts can be resolved by surrounding the word with double quotes, but remember that you are typing commands to a shell, and shells typically consume the outermost layer of quotes. Therefore you have to escape the quotes, as in this example:

```
% lock_lint ignore foo in func \"func\"
```

If two files with the same base name are included in an analysis, and these two files contain `static` variables by the same name, confusion can result. LockLint will think the two variables are the same.

If you duplicate the definition for a `struct` with no tag, LockLint will not recognize the definitions as the same `struct`. The problem is that LockLint makes up a tag based on the file and line number where the `struct` is defined (such as `x.c@24`), and that tag will differ for the two copies of the definition.

If a function contains multiple automatic variables of the same name, LockLint cannot tell them apart. Since LockLint ignores automatic variables except when they are used as function pointers, this does not come up often. In the following code, for example, LockLint uses the name `:foo/fp` for both function pointers:

```
int foo(void (*fp)()) {
    (*fp)();
    {
        void (*fp)() = get_func();
        (*fp)();
         ...
```

LockLint Subcommands

Some of these are equivalent to subcommands such as `assert`. Source code annotations are often preferable to subcommands, because they

- Have finer granularity
- Are easy to maintain
- Serve as comments on the code in question

See also “Source Code Annotations” on page 281 for information on annotations you can put directly into your source code.

`analyze`

`analyze` has the following syntax:

```
analyze [-hv]
```

Analyzes the loaded files for lock inconsistencies that may lead to data races and deadlocks. This subcommand may produce a great deal of output, so you may want to redirect the output to a file. This subcommand can be run only once for each saved state. (See “save” on page 325).

`-h` (history) produces detailed information for each phase of the analysis. No additional errors are issued.

`-v` (verbose) generates additional messages during analysis:

- `Writable variable read while no locks held!`
- `Variable written while no locks held!`
- `No lock consistently held while accessing variable!`

Output from the `analyze` subcommand can be particularly abundant if:

- the code has not been analyzed before
- the `assert read only` subcommand was not used to identify read-only variables
- no assertions were made about the protection of writable variables
- The output messages are likely to reflect situations that are not real problems; therefore, it is often helpful to first analyze the code without the `analyze` option, to show just the messages that are likely to represent real problems.

LockLint `analyze` *Phases*

Each problem encountered during analysis is reported on one or more lines, the first of which begins with an asterisk. Where possible, LockLint provides a complete traceback of the calls taken to arrive at the point of the problem. The analysis goes through the following phases:

1. **Checking for functions with variable side effects on locks**
   If a `disallow` sequence specifies that a function with locking side effects should not be analyzed, LockLint will produce incorrect results. If such `disallow` sequences are found, they are reported and analysis will not proceed.

2. **Preparing locks to hold order info**
   LockLint processes the asserted lock order information available to it. If LockLint detects a cycle in the asserted lock order, the cycle is reported as an error.

3. **Checking for function pointers with no targets**
   LockLint cannot always deduce assignments to function pointers. During this phase, LockLint reports any function pointer for which it does not think there is at least one target, whether deduced from the source or declared a `func.ptr` target.

4. **Removing accesses to ignored variables**
   To improve performance, LockLint removes references to ignored variables at this point. (This affects the output of the `vars` subcommands.)

5. **Preparing functions for analysis**
   During this phase, LockLint determines what side effects each function has on locks. (A *side effect* is a change in a lock's state that is not reversed before returning.) An error results if

   - The side effects do not match what LockLint expects
   - The side effects are different depending upon the path taken through the function
   - A function with such side effects is recursive

   LockLint expects that a function will have no side effects on locks, except where side effects have been added using the `assert side effect` subcommand.

6. **Preparing to recognize calling sequences to `allow/disallow`**
   Here LockLint is processing the various `allow/disallow` subcommands that were issued, if any. No errors or warnings are reported here.

7. **Checking locking side effects in function pointer targets**
   Calls through function pointers may target several functions. All functions that are targets of a particular function pointer must have the same side effects on locks (if any). If a function pointer has targets that differ in their side effects, analysis does not proceed.

8. **Checking for consistent use of locks with condition variables**
   Here LockLint checks that all waits on a particular condition variable use the same mutex. Also, if you assert that particular lock to protect that condition variable, LockLint makes sure you use that lock when waiting on the condition variable.

9. **Determining locks consistently held when each function is entered**
   During this phase, LockLint reports violations of assertions that locks should be held upon entry to a function (see `assert` subcommand). Errors such as locking a mutex lock that is already held, or releasing a lock that is not held, are also reported. Locking an anonymous lock, such as `foo::lock`, more than once is not considered an error, unless the `declare one` command has been used to indicate otherwise. (See "Inversions" on page 331 for details on anonymous data.)

10. **Determining locks consistently held when each variable is accessed**
    During this phase, LockLint reports violations of assertions that a lock should be held when a variable is accessed (see the `assert` subcommand). Also, any writes to read-only variables are reported.

    Occasionally you may get messages that certain functions were never called. This can occur if a set of functions (none of which are root functions) call each other. If none of the functions is called from outside the set, LockLint reports that the functions were never called at all. The `declare root` subcommand can be used to fix this situation for a subsequent analysis.

    Using the `disallow` subcommand to disallow all sequences that reach a function will also cause a message that the function is never called.

Once the analysis is done, you can find still more potential problems in the output of the `vars` and `order` subcommands.

assert

`assert` has the following syntax:

```
assert side effect    mutex           acquired in    func ...
assert side effect    rwlock [read]   acquired in    func ...
assert side effect    lock            released in    func ...
assert side effect    rwlock          upgraded in    func ...
assert side effect    rwlock          downgraded in  func ...
assert mutex|rwlock   protects                       var ...
assert mutex          protects                       func ...
assert rwlock         protects        [reads in]     func ...
assert order                                         lock lock ...
assert read only                                     var ...
assert rwlock         covers                         lock ...
```

These subcommands tell LockLint how the programmer expects locks and variables to be accessed and modified in the application being checked. During analysis any violations of such assertions are reported.

Note – If a variable is asserted more than once, only the last `assert` takes effect.

assert side effect

The following assertions state that given functions have side effects with respect to lock states:

```
assert side effect mutex acquired in
assert side effect rwlock [read] acquired in
assert side effect lock released in
assert side effect rwlock upgraded in
assert side effect rwlock downgraded in
```

A *side effect* is a change made by a function in the state of a lock, a change that is not reversed before the function returns. If a function contains locking side effects and no assertion is made about the side effects, or the side effects differ from those that are asserted, a warning will be issued during the analysis. The analysis then continues as if the unexpected side effect never occurred.

Note – There is another kind of side effect called an *inversion*. See "Inversions" on page 331, and the `locks` or `funcs` subcommands, for more details.

Warnings are also issued if the side effects produced by a function could differ from call to call (for example, conditional side effects). The keywords `acquired in`, `released in`, `upgraded in`, and `downgraded in` describe the type of locking side effect being asserted about the function. The keywords correspond to the side effects available via the threads library interfaces and the DDI and DKI Kernel Functions (see `mutex`(3T), `rwlock`(3T), `mutex`(9F) and `rwlock`(9F)).

The side effect assertion for `rwlocks` takes an optional argument `read`; if `read` is present, the side effect is that the function acquires read-level access for that lock. If `read` is not present, the side effect specifies that the function acquires write-level access for that lock.

assert mutex|rwlock protects

Asserting that a mutex lock protects a variable causes an error whenever the variable is accessed without holding the mutex lock. Asserting that a readers-writer lock protects a variable causes an error whenever the variable is read without holding the lock for read access or written without holding

the lock for write access. Subsequent assertions as to which lock protects a variable override any previous assertions; that is, only the last lock asserted to protect a variable is used during analysis.

`assert mutex protects`

Asserting that a mutex lock protects a function causes an error whenever the function is called without holding the lock. For root functions, the analysis is performed as if the root function were called with this assertion being true.

`assert rwlock protects`

Asserting that a readers-writer lock protects a function causes an error whenever the function is called without holding the lock for write access. Asserting that a readers-writer lock protects reads in a function causes an error whenever the function is called without holding the lock for read access. For root functions, the analysis is performed as if the root function were called with this assertion being true.

Note – To avoid flooding the output with too many violations of a single `assert... protects` subcommand, a maximum of 20 violations of any given assertion is shown. This limit does not apply to the `assert order` subcommand.

`assert order`

Informs LockLint of the order in which locks should be acquired. That is, LockLint assumes that the program avoids deadlocks by adhering to a well-known lock order. Using this subcommand, you can make LockLint aware of the intended order so that violations of the order can be printed during analysis.

`assert read only`

States that the given set of variables should never be written by the application; LockLint reports any writes to the variables. Unless a variable is read-only, reading the variable while no locks are held will elicit an error since LockLint assumes that the variable could be written by another thread at the same time.

`assert` *rwlock* `covers`

Informs LockLint of the existence of a hierarchical locking relationship. A readers-writer lock may be used in conjunction with other locks (mutex or readers-writer) in the following way to increase performance in certain situations:

- A certain readers-writer lock, called the *cover*, must be held while any of a set of other covered locks is held. That is, it is illegal (under these conventions) to hold a covered lock while not also holding the cover, with at least read access.
- While holding the cover for write access, you can access any variable protected by one of the covered locks without holding the covered lock. This works because it is impossible for another thread to hold the covered lock (since it would also have to be holding the cover). The time saved by not locking the covered locks can increase performance if there is not excessive contention over the cover.

Using `assert` *rwlock* `covers` prevents LockLint from issuing error messages when a thread accesses variables while holding the cover for write access but not the covered lock. It also enables checks to ensure that a covered lock is never held when its cover is not.

`declare`

`declare` has the following syntax:

```
declare    mutex          mutex ...
declare    rwlocks        rwlock ...
declare    func_ptr       targets    func ...
declare    nonreturning   func ...
declare    one            tag ...
declare    readable       var ...
declare    root           func ...
```

These subcommands tell LockLint things that it cannot deduce from the source presented to it.

`declare mutex` *mutex*

`declare rwlocks` *rwlock*

These subcommands (along with `declare root`, below) are typically used when analyzing libraries without a supporting harness. The subcommands `declare mutex` and `declare rwlocks` create mutex and reader-writer locks of the given names. These symbols can be used in subsequent `assert` subcommands.

`declare` *func_ptr* `targets` *func*

Adds the specified functions to the list of functions that could be called through the specified function pointer.

LockLint manages to gather a good deal of information about function pointer targets on its own by watching initializations and assignments. For example, for the code

```
struct foo { int (*fp)(); } foo1 = { bar };
```

LockLint does the equivalent of the command

```
% lock_lint declare foo::fp targets  bar
```

Caution – LockLint does not yet do the following (for the above example):

```
% lock_lint declare foo1.fp targets  bar
```

However, it does manage to do both for assignments to function pointers. See “Inversions” on page 331.

`declare nonreturning` *func*

Tells LockLint that the specified functions do not return. Armed with this knowledge, LockLint will not give errors about lock state after calls to such functions.

`declare one` *tag*

Tells LockLint that only one unnamed instance exists of each structure whose tag is specified. This knowledge makes it possible for LockLint to give an error if a lock in that structure is acquired multiple times without being released. Without this knowledge, LockLint does not complain about multiple acquisitions of anonymous locks (for example, `foo::lock`), since two different instances of the structure could be involved. (See “Inversions” on page 331.)

`declare readable` *var*

Tells LockLint that the specified variables may be safely read without holding any lock, thus suppressing the errors that would ordinarily occur for such unprotected reads.

`declare root` *func*

Tells LockLint to analyze the given functions as a root function; by default, if a function is called from any other function, LockLint will not attempt to analyze that function as the root of a calling sequence.

A *root function* is a starting point for the analysis; functions that are not called from within the loaded files are naturally roots. This includes, for example, functions that are never called directly but are the initial starting point of a thread (for example, the target function of a `thread_create` call). However, a function that *is* called from within the loaded files might also be called from outside the loaded files, in which case you should use this subcommand to tell LockLint to use the function as a starting point in the analysis.

`disallow`

`disallow` has the following syntax:

```
disallow func ...
```

Tells LockLint that the specified calling sequence should not be analyzed. For example, to prevent LockLint from analyzing any calling sequence in which `f()` calls `g()` calls `h()`, use the subcommand

```
% lock_lint disallow f g h
```

Function pointers can make a program appear to follow many calling sequences that do not in practice occur. Bogus locking problems, particularly deadlocks, can appear in such sequences. (See also the description of the subcommand “reallow” on page 324.) `disallow` prevents LockLint from following such sequences.

`disallows`

`disallows` has the following syntax:

```
disallows
```

Lists the calling sequences that are disallowed by the `disallow` subcommand.

`exit`

Actually, there *is* no exit subcommand for LockLint. To exit LockLint, use the exit command for the shell you are using.

`files`

`files` has the following syntax:

```
files
```

Lists the `.ll` versions of the source code files loaded with the `load` subcommand.

funcptrs

`funcptrs` has the following syntax:

```
funcptrs [-botu] func_ptr ...
funcptrs [-blotuz]
```

Lists information about the function pointers used in the loaded files. One line is produced for each function pointer.

`-b`

(*bound*) This option lists only function pointers to which function targets have been bound, that is it suppresses the display of function pointers for which there are no bound targets.

`-l`

(*long*) Equivalent to `-ot`.

`-o`

(*other*) This presents the following information about each function pointer:

`Calls=#`

Indicates the number of places in the loaded files this function pointer is used to call a function.

`=nonreturning`

Indicates that a call through this function pointer never returns (none of the functions targeted ever return).

`-t`

(*targets*) This option lists the functions currently bound as targets to each function pointer listed, as follows:

```
targets={ func ... }
```

`-u`

(*unbound*) This lists only those function pointers to which no function targets are bound. That is, suppresses the display of function pointers for which there are bound targets.

`-z`

(*zero*) This lists function pointers for which there are no calls. Without this option information is given only on function pointers through which calls are made.

You can combine various options to `funcptrs`:

1. **This example lists information about the specified function pointers. By default, this variant of the subcommand gives all the details about the function pointers, as if `-ot` had been specified.**

```
funcptrs [-botu] func_ptr ...
```

2. **This example lists information about all function pointers through which calls are made. If `-z` is used, even function pointers through which no calls are made are listed.**

```
funcptrs [-blotuz]
```

`funcs`

`funcs` has the following syntax:

```
funcs [-adehou]func ...
funcs [-adehilou]
funcs [-adehlou] [directly] called byfunc ...
funcs [-adehlou] [directly] callingfunc ...
funcs [-adehlou] [directly] readingvar ...
funcs [-adehlou] [directly] writingvar ...
funcs [-adehlou] [directly] accessingvar ...
funcs [-adehlou] [directly] affectinglock ...
funcs [-adehlou] [directly] invertinglock ...
```

`funcs` lists information about the functions defined and called in the loaded files. Exactly one line is printed for each function.

`-a`

(*asserts*) This option shows information about which locks are supposed to be held on entry to each function, as set by the `assert` subcommand. When such assertions have been made, they show as:

```
asserts={ lock ... }
read_asserts={ lock ... }
```

An asterisk will appear before the name of any lock that was not consistently held upon entry (after analysis).

`-e`

(*effects*) This option shows information about the side effects each function has on locks (for example, “acquires mutex lock foo”). If a function has such side effects, they are shown as:

```
side_effects={ effect [, effect] ... }
```

Using this option prior to analysis shows side effects asserted by an `assert side effect` subcommand. After analysis, information on side effects discovered during the analysis is also shown.

`-d`

(*defined*) This option shows only those functions which are *defined* in the loaded files. That is, that it suppresses the display of undefined functions.

`-h`

(*held*) This option shows information about which locks were consistently held when the function was called (after analysis). Locks consistently held for read (or write) on entry show as:

```
held={ lock ... }+{ lock ... }
read_held={ lock ... }+{ lock ... }
```

The first list in each set is the list of locks *consistently* held when the function was called; the second is a list of *inconsistently* held locks—locks that were sometimes held when the function was called, but not every time.

`-i`

(*ignored*) This option lists ignored functions.

`-l`

(*long*) Equivalent to `-aeoh`.

`-o`

(*other*) This option causes LockLint to present, where applicable, the following information about each function:

`=ignored`
Indicates that LockLint has been told to ignore the function using the `ignore` subcommand.

`=nonreturning`
Indicates that a call through this function never returns (none of the functions targeted ever return).

`=rooted`
Indicates that the function was made a root using the `declare root` subcommand.

`=root`
Indicates that the function is naturally a root (is not called by any function).

`=recursive`
Indicates that the function makes a call to itself.

`=unanalyzed`
Indicates that the function was never called during analysis (and is therefore unanalyzed). This differs from `=root` in that this can happen when `foo` calls `bar` and `bar` calls `foo`, and no other function calls either `foo` or `bar`, and neither have been rooted (see `=rooted`). So, because `foo` and bar are not roots, and they can never be reached from any root function, they have not been analyzed.

`calls=#`
Indicates the number of places in the source code, as represented by the loaded files, where this function is called. These calls may not actually be analyzed; for example, a `disallow` subcommand may prevent a call from ever really taking place.

`-u`
(*undefined*) This option shows only those functions that are *undefined* in the loaded files.

`funcs [-adehou]` *func*`...`
Lists information about individual functions. By default, this variant of the subcommand gives all the details about the functions, as if `-aeho` had been specified.

`funcs [-adehilou]`
Lists information about all unignored functions. If `-i` is used, even ignored functions are listed.

`funcs [-adehlou] [directly] called by` *func*`...`
Lists only those functions that may be called as a result of calling the specified functions. If `directly` is used, only those functions called by the specified functions are listed. If `directly` is not used, any functions *those* functions called are also listed, and so on.

`funcs [-adehlou] [directly] calling` *func*`...`
Lists only those functions that, when called, may result in one or more of the specified functions being called. See notes below on `directly`.

`funcs [-adehlou] [directly] reading` *var*`...`
Lists only those functions that, when called, may result in one or more of the specified variables being read. See notes below on `directly`.

`funcs [-adehlou] [directly] writing` *var*`...`
Lists only those functions that, when called, may result in one or more of the specified variables being written. See notes below on `directly`.

`funcs [-adehlou] [directly] accessing` *var* **...**

Lists only those functions that, when called, may result in one or more of the specified variables being accessed (read or written). See notes below on `directly`.

`funcs [-adehlou] [directly] affecting` *lock* **...**

Lists only those functions that, when called, may result in one or more of the specified locks being affected (acquired, released, upgraded, or downgraded). See notes below on `directly`.

`funcs [-adehlou] [directly] inverting` *lock* **...**

Lists only those functions that invert one or more of the specified locks. (See "Inversions" on page 331.) If `directly` is used, only those functions that themselves invert one or more of the locks (actually release them) are listed. If `directly` is not used, any function that is called with a lock already held, and then calls another function that inverts the lock, is also listed, and so on.

For example, in the following code, `f3()` directly inverts lock `m`, and `f2()` indirectly inverts it:

```
f1() { pthread_mutex_unlock(&m); f2(); pthread_mutex_lock(&m); }
f2() { f3(); }
f3() { pthread_mutex_unlock(&m); pthread_mutex_lock(&m); }
```

About `directly`

Except where stated otherwise, variants that allow the keyword `directly` only list the functions that *themselves* fit the description. If `directly` is not used, all the functions that call those functions will be listed, and any functions that call *those* functions, and so on.

`help`

`help` has the following syntax:

```
help [keyword]
```

Without a keyword, `help` displays the subcommand set.

With a keyword, `help` gives helpful information relating to the specified keyword. The keyword may be the first word of any LockLint subcommand. There are also a few other keywords for which help is available:

`condvars`	`locking`
`example`	`makefile`
`ifdef`	`names`
`inversions`	`overview`
`limitations`	`shell`

If environment variable `PAGER` is set, that program is used as the pager for `help`. If `PAGER` is not set, `more` is used.

ignore

`ignore` has the following syntax:

```
ignore func|var ...  [ in func ... ]
```

Tells LockLint to exclude certain functions and variables from the analysis. This exclusion may be limited to specific functions using the `in` *func* ... clause; otherwise the exclusion applies to all functions.

The commands

```
% lock_lint funcs -io | grep =ignored
% lock_lint vars -io | grep =ignored
```

show which functions and variables are ignored.

load

`load` has the following syntax:

```
load file ...
```

Loads the specified `.ll` files. The extension may be omitted, but if an extension is specified, it must be `.ll`. Absolute and relative paths are allowed. You are talking to a shell, so the following are perfectly legal (depending upon your shell's capabilities):

```
% lock_lint load *.ll
% lock_lint load ../foo/abcdef{1,2}
% lock_lint load `find .  -name \*.ll -print`
```

The text for load is changed extensively. To set the new text, type:

```
% lock_lint help load
```

locks

`locks` has the following syntax:

```
locks [-co] lock ...
locks [-col]
locks [-col] [directly] affected by func ...
locks [-col] [directly] inverted by func ...
```

Lists information about the locks of the loaded files. Only those variables that are actually *used* in lock manipulation routines are shown; locks that are simply declared but never manipulated will not be shown.

`-c`

(*cover*) This option shows information about lock hierarchies. Such relationships are described using the `assert` *rwlock* `covers` subcommand. (When locks are arranged in such a hierarchy, the covering lock must be

held, at least for read access, whenever any of the covered locks is held. While holding the covering lock for write access, it is unnecessary to acquire any of the covered locks.) If a lock covers other locks, those locks show as:

```
covered={ lock ...  }
```

If a lock is covered by another lock, the covering lock shows as

```
cover=lock
```

`-l`

(*long*) Equivalent to `-co`.

`-o`

(*other*) Causes the type of the lock to be shown as (*type*) where *type* is *mutex*, *rwlock*, or *ambiguous type* [used as a mutex in some places and as a rwlock (readers-writer) in other places].

`locks [-co]` *lock* `...`

Lists information about individual locks. By default, this variant of the subcommand gives all the details about the locks, as if `-co` had been specified.

`locks [-col]`

Lists information about all locks.

`locks [-col] [directly] affected by` *func* **...**

Lists only those locks that may be affected (acquired, released, upgraded, or downgraded) as a result of calling the specified functions. If the keyword `directly` is used, only functions that use the threads library routines directly to affect a lock (acquire, release, upgrade, or downgrade) are listed. If the keyword `directly` is not used, any function that calls a function that affects a lock will be listed, and any function calling that function will be listed, and so on.

`locks [-col] [directly] inverted by` *func* ...

Lists only those locks that may be inverted by calling one of the specified functions. (See “Inversions” on page 331.)

If the keyword `directly` is used, only those locks that are directly inverted by the specified functions (that is, the functions that actually release and reacquire locks using a threads library routine) are listed. If the keyword `directly` is not used, a lock that is held by one of the specified functions and inverted by some function called from it (and so on) is also listed. For example, in the following code `f1` directly inverts `m1`, and indirectly inverts `m2`.

```
f1() { pthread_mutex_unlock(&m1); f2(); pthread_mutex_lock(&m1); }
f2() { f3(); }
f3() { pthread_mutex_unlock(&m2); pthread_mutex_lock(&m2); }
```

`members`

`members` has the following syntax:

```
members struct_tag
```

Lists the members of the `struct` with the specified tag, one per line. For structures that were not assigned a tag, the notation file@line is used (for example, `x.c@29`), where the file and line number are the source location of the `struct`’s declaration.

`members` is particularly useful to use as input to other LockLint subcommands. For example, when trying to assert that a lock protects all the members of a `struct`, the following command suffices:

```
% lock_lint assert foo::lock protects `lock_lint members foo`
```

Note – The `members` subcommand does not list any fields of the `struct` that are defined to be of type `mutex_t`, `rwlock_t`, `krwlock_t`, or `kmutex_t`.

order

`order` has the following syntax:

```
order [lock [lock]]
order summary
```

The `order` subcommand lists information about the order in which locks are acquired by the code being analyzed. It may be run only after the `analyze` subcommand.

order [lock [lock]]

Shows the details about lock pairs. For example, the command

```
% lock_lint order foo bar
```

shows whether an attempt was made to acquire lock `bar` while holding lock `foo`. The output looks something like the following:

```
:foo :bar seen (first never write-held), valid
```

First the output tells whether such an attempt actually occurred (`seen` or `unseen`). If the attempt occurred, but never with one or both of the locks write-held, a parenthetical message to that effect appears, as shown. In this case, `foo` was never write-held while acquiring `bar`.

If an assertion was made about the lock order, the output shows whether the specified order is *valid* or *invalid* according to the assertion. If there was no assertion about the order of `foo` and `bar`, or if both orders were asserted (presumably because the user wanted to see all places where one of the locks was held while acquiring the other), the output will indicate neither valid nor invalid.

`order summary`

Shows in a concise format the order in which locks are acquired. For example, the subcommand might show

```
:f :e :d :g :a
:f :c :g :a
```

In this example, there are two orderings because there is not enough information to allow locks `e` and `d` to be ordered with respect to lock `c`.

Some cycles will be shown, while others won't. For example,

```
:a :b :c :b
```

is shown, but

```
:a :b :c :a
```

(where no other lock is ever held while trying to acquire one of these) is not. Deadlock information from the analysis will still be reported.

`pointer calls`

`pointer calls` has the following syntax:

```
pointer calls
```

Lists calls made through function pointers in the loaded files. Each call is shown as:

function [*location of call*] `calls through funcptr` *func_ptr*

For example,

```
foo.c:func1 [foo.c,84] calls through funcptr bar::read
```

means that at line 84 of `foo.c`, in `func1` of `foo.c`, the function pointer `bar::read` (member `read` of a pointer to `struct` of type `bar`) is used to call a function.

reallow

`reallow` has the following syntax:

```
reallow func ...
```

Allows you to make exceptions to `disallow` subcommands. For example, to prevent LockLint from analyzing any calling sequence in which `f()` calls `g()` calls `h()`, except when `f()` is called by `e()` which was called by `d()`, use the commands

```
% lock_lint disallow f g h
% lock_lint reallow d e f g h
```

In some cases you may want to state that a function should only be called from a particular function, as in this example:

```
% lock_lint disallow f
% lock_lint reallow e f
```

Note – A `reallow` subcommand only suppresses the effect of a `disallow` subcommand if the sequences end the same. For example, after the following commands, the sequence `d e f g h` would still be disallowed:

```
% lock_lint disallow e f g h
% lock_lint reallow d e f g
```

reallows

`reallows` has the following syntax:

```
reallows
```

Lists the calling sequences that are reallowed, as specified using the `reallow` subcommand.

`refresh`

`refresh` has the following syntax:

```
refresh
```

1. Pops the saved-state stack, restoring LockLint to the state of the top of the saved-state stack
2. Prints the description, if any, associated with that state
3. Immediately resaves (pushes) the state again (see the `restore` and `save` subcommands)

`restore`

`restore` has the following syntax:

```
restore
```

Pops the saved state stack, restoring LockLint to the state of the top of the saved-state stack, and prints the description, if any, associated with that state.

The saved state stack is a LIFO (Last-In-First-Out) stack. Once a saved state is restored (popped) from the stack, that state is no longer on the saved-state stack. If the state needs to be saved and restored repeatedly, simply save the state again immediately after restoring it, or use the `refresh` subcommand.

`save`

`save` has the following syntax:

```
save description
```

Saves the current state of the tool on a stack. The user-specified *description* is attached to the state. Saved states form a LIFO (Last-In-First-Out) stack, so that the last state saved is the first one restored.

This subcommand is commonly used to save the state of the tool before running the `analyze` subcommand, which may be run only once on a given state. For example, you may do the following:

```
%: lock_lint load *.ll
%: lock_lint save Before Analysis
%: lock_lint analyze
    <output from analyze>
%: lock_lint vars -h | grep \*
    <apparent members of struct foo are not consistently protected>
%: lock_lint refresh Before Analysis
%: lock_lint assert lock1 protects `lock_lint members foo`
%: lock_lint analyze
    <output now contains info about where the assertion is violated>
```

saves

`saves` has the following syntax:

```
saves
```

Lists the descriptions of the states saved on the saved stack via the `save` subcommand. The descriptions are shown from top to bottom, with the first description being the most recently saved state that has not been restored, and the last description being the oldest state saved that has not been restored.

start

`start` has the following syntax:

```
start [cmd]
```

Starts a LockLint session. A LockLint session must be started prior to using any other LockLint subcommand. By default, `start` establishes LockLint's context and starts a subshell for the user, as specified via `$SHELL`, within that context. The only piece of the LockLint context exported to the shell is the environment variable `LL_CONTEXT`. `LL_CONTEXT` contains the path to the temporary directory of files used to maintain a LockLint session.

cmd specifies a command and its path and options. By default, if *cmd* is not specified, the value of `$SHELL` is used.

Note – To exit a LockLint session use the `exit` command of the shell you are using.

See "Limitations of LockLint" on page 277, for more on setting up the LockLint environment for a `start` subcommand.

Examples

The following examples show variations of the `start` subcommand.

1. Using the default

```
% lock_lint start
```

LockLint's context is established and `LL_CONTEXT` is set. Then the program identified by `$SHELL` is executed. Normally, this will be your default shell. LockLint subcommands can now be entered. Upon exiting the shell, the LockLint context will be removed.

2. Using a pre-written script

```
% lock_lint start foo
```

The LockLint context is established and `LL_CONTEXT` is set. Then, the command `/bin/csh -c foo` is executed. This results in executing the C shell command file `foo`, which contains LockLint commands. Upon completing the execution of the commands in foo by `/bin/csh`, the LockLint context is removed.

If you use a shell script to start LockLint, insert `#!` in the first line of the script to define the name of the interpreter that processes that script. For example, to specify the C-shell the first line of the script is:

```
#! /bin/csh
```

3. **Starting up with a specific shell**
   In this case, the user starts LockLint with the Korn shell:

```
% lock_lint start /bin/ksh
```

After establishing the LockLint context and setting `LL_CONTEXT`, the command `/bin/ksh` is executed. This results in the user interacting with an interactive Korn shell. Upon exiting the Korn shell, the LockLint context is removed.

sym

`sym` has the following syntax:

```
sym name ...
```

Lists the fully qualified names of various things the specified names could refer to within the loaded files. For example, `foo` might refer both to variable `x.c:func1/foo` and to function `y.c:foo`, depending on context.

unassert

`unassert` has the following syntax:

```
unassert vars var ...
```

Undoes any assertion about locks protecting the specified variables. There is no way to remove an assertion about a lock protecting a *function*.

vars

`vars` has the following syntax:

```
vars [-aho] var ...
vars [-ahilo]
vars [-ahlo] protected by lock
vars [-ahlo] [directly] read by func ...
vars [-ahlo] [directly] written by func ...
vars [-ahlo] [directly] accessed by func ...
```

Lists information about the variables of the loaded files. Only those variables that are actually *used* are shown; variables that are simply declared in the program but never accessed will not be shown.

`-a`

(*assert*) Shows information about which lock is supposed to protect each variable, as specified by the `assert` *mutex*|*rwlock* `protects` subcommand. The information is shown as follows:

```
assert=lock
```

If the assertion is violated, then after analysis this will be preceded by an asterisk, such as `*assert=<`*lock*`>`.

`-h`

(*held*) Shows information about which locks were consistently held when the variable was accessed. This information is shown after the `analyze` subcommand has been run. If the variable was never accessed, this information is not shown. When it is shown, it looks like this:

```
held={ <lock> ... }
```

If no locks were consistently held and the variable was written, this is preceded by an asterisk, such as `*held={ }`.

Unlike `funcs`, the `vars` subcommand lists a lock as protecting a variable even if the lock was not actually held, but was simply covered by another lock. (See `assert` *rwlock* `covers` in the description of "assert" on page 305.)

`-i`

(*ignored*) causes even *ignored* variables to be listed.

`-l`

(*long*) Equivalent to `-aho`.

`-o`

(*other*) Where applicable, shows information about each variable:

`=cond_var`
Indicates that this variable is used as a condition variable.

`=ignored`
Indicates that LockLint has been told to ignore the variable explicitly via an `ignore` subcommand.

`=read-only`
Means that LockLint has been told (by `assert read only`) that the variable is read-only, and will complain if it is written. If it is written, then after analysis this will be followed by an asterisk, such as `=read-only*` for example.

`=readable`
Indicates that LockLint has been told by a `declare readable` subcommand that the variable may be safely read without holding a lock.

`=unwritten`
May appear after analysis, meaning that while the variable was not declared read-only, it was never written.

`vars [-aho]` *var* ...

Lists information about individual variables. By default, this variant of the subcommand gives all the details about the variables, as if `-aho` had been specified.

`vars [-ahilo]`

Lists information about all unignored variables. If `-i` is used, even ignored variables are listed.

`vars [-ahlo] protected by` *lock*

Lists only those variables that are protected by the specified lock. This subcommand may be run only after the `analyze` subcommand has been run.

`vars [-ahlo] [directly] read by` *func* ...

Lists only those variables that may be read as a result of calling the specified functions. See notes below on `directly`.

`vars [-ahlo] [directly] written by` *func* ...

Lists only those variables that may be written as a result of calling the specified functions. See notes below on `directly`.

`vars [-ahlo] [directly] accessed by` *func* ...

Lists only those variables that may be accessed (read or written) as a result of calling the specified functions.

About `directly`

Those variants that list the variables accessed by a list of functions can be told to list only those variables that are *directly* accessed by the specified functions. Otherwise the variables accessed by those functions, the functions they call, the functions *those* functions call, and so on, will be listed.

Inversions

A function is said to *invert* a lock if the lock is already held when the function is called, and the function releases the lock, such as:

```
foo() {
    pthread_mutex_unlock(&mtx);
    ...
    pthread_mutex_lock(&mtx);
}
```

Lock inversions are a potential source of insidious race conditions, since observations made under the protection of a lock may be invalidated by the inversion. In the following example, if `foo()` inverts `mtx`, then upon its return `zort_list` may be `NULL` (another thread may have emptied the list while the lock was dropped):

```
ZORT* zort_list;
    /* VARIABLES PROTECTED BY mtx: zort_list */

void f() {
    pthread_mutex_lock(&mtx);
    if (zort_list == NULL) /* trying to be careful here */
    return;
    foo();
    zort_list->count++; /* but zort_list may be NULL here!! */
    pthread_mutex_unlock(&mtx);
}
```

Lock inversions may be found using the commands:

```
% lock_lint funcs [directly] inverting lock ...
% lock_lint locks [directly] inverted by func ...
```

An interesting question to ask is “Which functions acquire locks that then get inverted by calls they make?” That is, which functions are in danger of having stale data? The following (Bourne shell) code can answer this question:

```
$ LOCKS=`lock_lint locks‘
$ lock_lint funcs calling `lock_lint funcs inverting $LOCKS`
```

The following gives similar output, separated by lock:

```
for lock in `lock_lint locks`
do
    echo “functions endangered by inversions of lock $lock”
    lock_lint funcs calling `lock_lint funcs inverting $lock`
done
```

Part 3 — Using Source Browsing

Browsing Source With sbquery 25

This chapter describes one of the command-line utilities for browsing source code. The `sbquery` command provides you with a command-line browsing environment that you can access from terminals and from workstations emulating terminals.

The information contained in this chapter pertains mainly to the use of the command line to complete tasks also available from within WorkShop. For more conceptual information on using source browsing, see the online help.

This chapter is organized into the following sections:

Basic Concepts

The command-line interface to the Source Browsing mode of WorkShop is `sbquery`.

To issue a query from the command line:

- **Type** `sbquery`**, followed by any command-line options and their arguments, followed by the symbol you want to search for.**

```
sbquery [options] symbols
```

All lines that contain *symbols* are displayed.

`sbquery` displays a list of matches that includes the file in which the symbol appears, the line number, the function containing the symbol, and the source line containing the symbol.

By default, `sbquery` searches for symbols in the database in the current working directory. If you want to browse a database stored in another directory, see Chapter 26, “Controlling the Browser Database With .sbinit.”

Command Reference

To obtain a list of the `sbquery` command-line options:

♦ **Type** `sbquery` **at the shell prompt.**

Two types of options are available to help you narrow your search: filter options and focus options. The filter options are used to search for symbols based on how they are used in a program. For example, you could limit your search to declarations of variables. The focus options limit your search to specific classes of code, such as particular programs, functions, or libraries.

Arguments	Description
`-break_lock`	Breaks the lock on a locked database. This argument might be needed if the update of the index file is aborted, the next time you issue a query you might get a message telling you that the database is locked. After using this option, your database may be in an inconsistent state. To ensure consistency, remove the `.sb` subdirectory and recompile your program.
`-help`	Displays a synopsis of the `sbquery` command. Equivalent to typing `sbquery` with no options.
`-help_focus`	Displays a list of the focus options available for querying only specific program types in a directory. Use focus options to issue a query limited to specific units of code such as programs or functions.

Arguments	Description *(Continued)*
`-help_filter`	Displays a list of the languages for which filter options are available. Use filtering options to search for symbols based on how they are used in a program.
`-max_memory <size>`	Sets the approximate amount of memory, in megabytes, that should be allocated before `sbquery` uses temporary files when building the index file.
`-no_update`	Doesn't rebuild the index file when you issue a query following compilation. If you do not include this option and issue a query following compilation or recompilation, then the database updates and processes your query.
`-o <file>`	Sends query output to a file.
`-show_db_dirs`	Lists all `.sb` directories scanned when you issue a query. The list includes the following: the `.sb` directory in the current working directory and all other `.sb` directories specified by the `import` or `export` commands in your `.sbinit` file.
`-version | -V`	Displays the current version number.
`-files_only`	Lists only the files where the symbols you are searching for appear.
`-no_secondaries`	Returns only the primary match. A secondary match is an identifier inside a macro. You might want to turn off secondary matches if you are doing a lot of filtered querying, and the symbols you are querying on are used in a lot of macros.
`-no_source`	Displays only the file name and line number associated with each match.
`-symbols_only`	Displays a list of all symbols that match the patterns in your search pattern. Useful when you use wildcards in a query.

Arguments	Description *(Continued)*
`-pattern <sym>`	Queries on `symbol`, which may contain special characters, including a leading dash (-). This option allows you to query on a symbol that looks like a command-line option. For instance, you can query on the symbol `-help`, and `sbquery` distinguishes it from the regular option `-help`.
`-no_case`	Makes the query case-insensitive.
`-sh_pattern`	Use shell-style expressions when issuing a query that includes wildcards. This wildcard setting is the default; include this option if you are doing other pattern matching on the same command line. See `sh`(1) for more information about shell-style pattern matching.
`-reg_expr`	Use regular expressions when issuing a query that includes wildcards. If you do not include this option, shell-style patterns are assumed.
`-literal`	Do not use any wildcard expressions for the query. Useful when you want to search for a string that contains meta characters from other wildcard schemes.

Filter Language Options

```
sbquery -help_filter language
```

Options
`ansi_c`
`sun_as`
`sun_c_plus_plus`
`sun_f77`
`sun_pascal`

Focus Options

```
sbquery focus_option symbol
```

Focus Option	Description
`-in_program` *Program*	Limit query to matches in *program*.
`-in_directory` *Directory*	Limit query to matches in *directory*.
`-in_source_file` *Source_File*	Limit query to matches in *source_file*.
`-in_function` *Function*	Limit query to matches in *function*.
`-in_class` *Class*	Limit query to matches in *class*.
`-in_template` *Template*	Limit query to matches in *template*.
`-in_language` *Language*	Limit query to matches in *language*.

Note – If you include two or more focus options, a match is returned if it is found with any of the supplied focus options.

Environment Variables

Environment variables provide information that affects the operation of `sbquery` (and of source browsing in WorkShop). For more information on `.sbinit`, see Chapter 26, “Controlling the Browser Database With .sbinit.”

Variable	Description
`HOME`	The name of your login directory.
`PWD`	The full path name of the current directory.
`SUNPRO_SB_ATTEMPTS_MAX`	The maximum number of times the index builder tries to access a locked database.
`SUNPRO_SB_EX_FILE_NAME`	The absolute pathname of the file `sun_source_browser.ex`.
`SUNPRO_SB_INIT_FILE_NAME`	The absolute pathname of the `.sbinit` file.

25

Controlling the Browser Database With `.sbinit`

26

This chapter describes how to work with source files whose database information is stored in multiple directories. As a default, the database is built in the current working directory and searches that database when you issue a query.

The information in this chapter pertains mainly to the use of the command line to complete tasks also available from within WorkShop. For more conceptual information on using source browsing, see the online help.

This chapter is organized into the following sections:

Basic Concepts

The text file, `.sbinit`, is used by the Source Browsing mode of WorkShop, the compilers, and `sbtags` to obtain control information about the Source Browsing database structure. Use `.sbinit` if you want to work with source files whose database information is stored in multiple directories.

The `.sbinit` file should be placed in the directory from which Source Browsing, the compilers, and `sbtags` are run. These tools look in the current working directory for the `.sbinit` file.

Moving the `.sbinit` *File*

The default is to look in the current working directory for the `.sbinit` file. To instruct WorkShop and the compiler to search for the `.sbinit` file in another directory:

- **Set the environment variable** `SUNPRO_SB_INIT_FILE_NAME` **to** `/absolute/pathname/.sbinit`**.**

File Commands

To use `.sbinit` commands, add the command to the file. The `.sbinit` file is limited to the following commands:

- `import`—Reads databases in directories other than the current working directory.
- `export`—Writes database component files associated with specified source files to directories other than the current working directory of the compiler. This command also acts as an import command.
- `automount-prefix`—Enables you to browse source files on a machine other than the one on which you compiled your program.
- `replacepath`—Specifies how to modify paths to file names found in the database, allowing you to move a database.

Command Reference

`import`

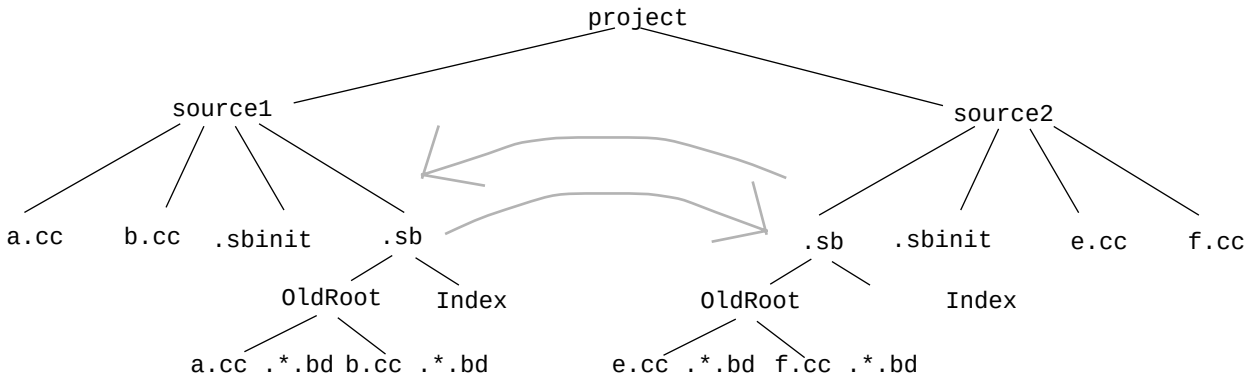
```
import /pathname to directory
```

Allows the Source Browsing mode of WorkShop to read databases in directories other than the current working directory. The `import` command enables you to retain separate databases for separate directories.

For example, you may want to set up administrative boundaries so that programmers working on Project A cannot write into directories for Project B and vice versa. In that case, Project A and Project B each need to maintain their own database, both of which can then be read but not written by programmers working on the other project.

In the following figure, the current working directory is `/project/source1`. The database in source2 is read by including either of these commands in the source1 `.sbinit` file:

```
import /project/source2
import ../source2
```



export

```
export prefix into path
```

Causes the compilers and `sbtags` to write database component files associated with source files to directories other than the current working directory used by the Source Browsing mode of WorkShop and the compiler.

Whenever the compiler processes a source file whose absolute path starts with *prefix*, the resulting browser database (`.bd`) file is stored in *path*`/.sb`.

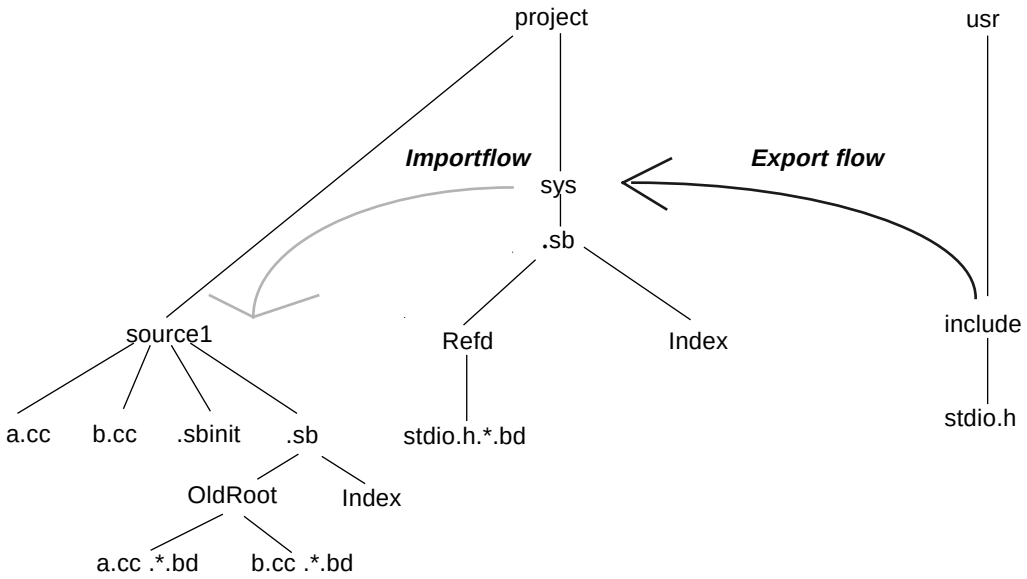
The export command contains an implied `import` command of *path*, so that exported database components are automatically read by the Source Browsing mode of WorkShop.

The `export` command enables you to save disk space by placing database files associated with identical files, such as `#include` files from `/usr/include`, in a single database, while still retaining distinct databases for individual projects.

If your `.sbinit` files include multiple `export` commands, then you must arrange them from the most specific to the least specific. The compiler scans `export` commands in the same order that it encounters them in the `.sbinit` file.

In this figure, to place the `.bd` file and index file created for files from `/usr/include` in a `.sb` subdirectory in the `sys` subdirectory, the user included this `export` command in the `.sbinit` file for `source1`:

```
export /usr/include into /project/sys
```



If the configuration had included a `source2` directory with a `.sbinit` file containing the same `export` command, then you would save disk space because you didn't create two identical database files. For the `stdio.h` files, the compiler would create a single database file for `stdio.h` in the `sys` subdirectory.

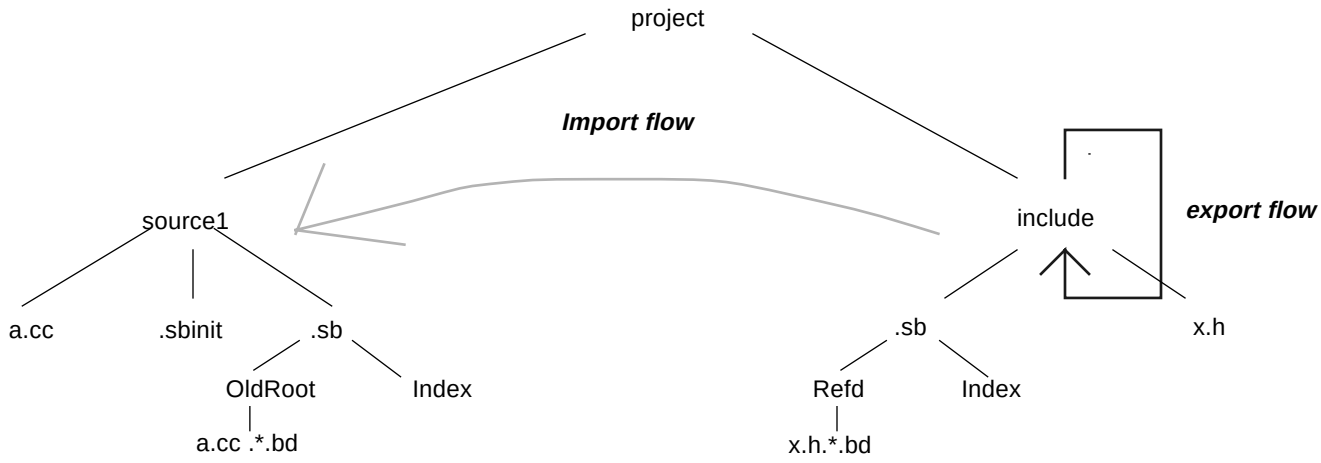
The `.sbinit` file contains an implied `export / into .` that instructs the compiler to put database files created for source files not explicitly mentioned by an `export` command in the current working directory. In the previous figure, the `.bd` files associated with `a.cc` and `b.cc` are placed in the `.sb` subdirectory in the `source1` directory.

When you include the `export` command in the `.sbinit` file, an implied `import` command causes the specified database to be read. Based on the configuration in the previous figure, the database in the `sys` subdirectory, as well as the database in the `source1` directory, is searched each time you issue a query.

As another example, the user included this export command in the `.sbinit` file for `source1`:

```
export /project/include into /project/include
```

This places the database files and index file created for files from `/project/include` in the `.sb` subdirectory in the `/project/include` subdirectory. An implied `import` command caused the database in `/project/include` to be read.



The database files are stored in a common subdirectory even though it located the `include` file in a nonstandard location. The `export` command saves disk space if the project includes multiple references from many different directories to the same `include` file.

`replacepath`

`replacepath` *from_prefix to_prefix*

This command specifies how to modify path names in the Source Browsing database.

In general, *from-prefix* corresponds to the automounter “mount point” (the path name where the automounter actually mounts the file system); and *to-prefix* corresponds to the automounter “trigger point” (the path name known and used by the developer).

There is considerable flexibility in how an automounter is used; the method can vary from host to host.

Path replacement rules are matched in the order that they are found and matching stops after a replacement is done.

The default `replacepath` command is used to strip away automounter artifacts:

```
replacepath /tmp_mnt
```

When used for this purpose, the command should be given as the mount point as the first argument and the trigger_point as the second argument.

automount-prefix

```
automount-prefix mount_point trigger_point
```

The `automount-prefix` command enables you to browse on a machine other than the one on which you compiled your program. This command is identical to the `replacepath` command except that `automount-prefix` path translations occur at compile time and are written into the database.

The `automount-prefix` command defines which automounter prefixes to remove from the source names stored in the database. The directory under which the automounter mounts the file systems is the *mount_point;* the *trigger_point* is the prefix you use to access the exported file system. The default is `/.`

The path translations from both commands (that is, `automount_prefix` and `replacepath`) are used to search for source files while browsing if the path in the database fails.

At first glance, this may not seem possible; the browser database that is created when you run the compiler contains the absolute path for each source file. If the absolute path is not uniform across machines, then WorkShop will not be able to display the source files when it responds to a query.

To get around this problem, you can do either of the following:

- Ensure that all source files are mounted at the same mount point on all machines.
- Compile your programs in an automounted path. A reference to such a path causes the *automounter* to automatically mount a file system from another machine.

There is a default `automount-prefix` command that is used to strip away automounter artifacts:

```
automount-prefix /tmp_mnt /
```

The default rule is generated only if no `automount-prefix` commands are specified.

For more information on using the automounter, see the `automount(1M)` man page.

`cleanup-delay`

This command limits the time elapsed between rebuilding the index and the associated database garbage collection. The compilers automatically invoke `sbcleanup` if the limit is exceeded. The default value is 12 hours.

Collecting Browsing Information With `sbtags`

27

The `sbtags` command provides a quick and convenient method for collecting browsing information from source files, enabling you to collect minimal browsing information for programs that do not compile completely.

The information contained in this chapter pertains mainly to the use of the command line to complete tasks also available from within WorkShop. For more conceptual information on using source browsing, see the online help.

This chapter is organized into the following sections:

Basic Concepts

The `sbtags` command collects a subset of the information available through compilation. The reduced information restricts some browsing functionality. A database generated by `sbtags` enables you to perform queries on functions and variables and to display the function call graph.

The follow restrictions on `sbtags` currently exist:

- Cannot issue queries about local variables
- Cannot browse classes
- Cannot graph class relationships
- Limited ability to issue complex queries
- Limited ability to focus queries

- Scanned database information may be less reliable than compiled information

Once a file has been changed, it often need not be scanned again to incorporate changes into the database.

An `sbtags` database is based on a lexical analysis of the source file. Though it does not always correctly identify all the language constructs, it will operate on files that will not compile and is faster than recompilation.

`sbtags` recognizes definitions for variables, types, and functions. It also collects information on function calls. No other information is collected (in particular, other semantic information for complex queries is not collected).

The functionality of `sbtags` is similar to `ctags` and `etags`, except for the Call Grapher information. You may mix direct queries to the database for definitions and graphing with pattern-matching queries.

With an `sbtags` generated database:

- Class Browser and Class Grapher features are not available.
- The database does not contain information on all symbols and strings. It contains information on functions, classes, types, and variable definitions, and calls to functions.
- Time is saved since the `sbtags` program runs faster than the compiler.
- The database size is much smaller than the size of your source code.
- The database content is not guaranteed to be semantically correct because the `sbtags` program performs only simple syntactic and semantic analysis and may sometimes be in error.
- A database is generated even if the source code cannot be compiled because the code is incomplete or semantically incorrect.

Generating an `sbtags` *Database*

To generate a browsing database using `sbtags`:

♦ **Type `sbtags`, followed by the name of the file (or files) for which you want to generate the database.**

```
sbtags file ...
```

Part 4 — Using Merging

Using twmerge 28

Merging lets you compare two text files, merge two files into a single new file, and compare two edited versions of a file against the original to create a new file which contains all new edits.

Starting twmerge from the command line enables you to:

- Determine which files (if any) are loaded into twmerge at startup
- Specify the name of the merged output file at startup
- Specify whether or not twmerge should consider leading white space (tabs and spaces) when identifying differences
- Specify whether the merged output file is writable or read-only
- Load lists of files from specified directories
- Specify input file display names for the left and right text panes in the Merging window (Useful when merging directories of files from a list.)

This chapter explains how to start Merging, load it with files, and save the output file. The chapter is organized into the following sections:

28

Understanding Merging

Merging loads and displays two text files for side-by-side comparison, each in a read-only text pane. Any differences between the two files are marked, and a merged version of the two files which you can edit to produce a final merged version.

When you load the two files to be merged, you can also specify a third file from which the two files were created for comparison. When you have specified this *ancestor* file, Merging marks lines in the *descendants* that are different from the ancestor and produces a merged file based on all three files

The merged version contains two types of lines:

- Lines common to both input files
- Resolved differences

Starting `twmerge`

To start `twmerge` without loading any input files:

♦ **Type** `twmerge.`

The directory you start `twmerge` from becomes the working directory for opening and saving files.

Loading Two Files at Startup

To load two files when starting `twmerge`, change to the directory in which the files are stored and type the file names on the command line.

To merge two files named `file_1` and `file_2`, type:

```
demo% twmerge file_1 file_2 &
```

The first file, `file_1`, opens in the left text pane; `file_2` opens in the right pane.

Loading Three Files at Startup

To merge two files and compare them to a common ancestor, change to the directory in which the files are stored and type:

```
demo% twmerge -a ancestor_file file_1 file_2 &
```

The ancestor file is not shown, but differences between the ancestor file and the two descendants are marked, with the output file being based on the ancestor file.

Loading Files from a List File

You can sequentially load files from a list of file names.

Suppose ancestor versions of a project's source files are stored in a directory named `/src`. You have been editing the files `file_1`, `file_2`, and `file_3` in your directory `/usr_1`, and another developer has been simultaneously editing the same files in the directory `/usr_2`. You want to merge the changes to both sets of files, and place the new merged versions in a directory named `/new_src`.

To merge the `/src`, `/usr_1`, and `/usr_2` directories, first create a list file that contains only the names of the three files to be merged, each name on a separate line, as follows:

```
file_1
file_2
file_3
```

Name the file `sourcelist` and place it in the directory where you plan to start `twmerge`. Change to that directory and start `twmerge` by typing:

```
demo% twmerge -a /src -1 sourcelist /usr_1/usr_2/new_src &
```

This command loads `/usr_1/file_1` into the left text pane, `/usr_2/file_1` into the right text pane, and sets the common ancestor as `/src/file_1`.

28

Working with Differences

Merging operates on *differences* between files. When Merging discovers a line that differs between the two files to be merged (or between either of the two files and an ancestor), it marks the lines in the two files with glyphs corresponding to how the lines differ. Together, these marked lines are called a *difference*. As you move through the files from one difference to the next, the lines that differ and their glyphs are highlighted.

Current, Next, and Previous Difference

The highlighted difference is called the *current* difference. The differences immediately before and immediately after are called the *previous* difference and the *next* difference.

Resolved and Remaining Difference

A difference is *resolved* if the changes to a line are accepted. A *remaining* difference is one that has not yet been resolved.

If the Auto Merge feature is on, Merging resolves differences automatically.

Moving Between Differences

You can move between differences using the buttons above the two panes, or the Navigate menu. Use the Previous and Next buttons to scroll through the differences without accepting them.

Resolving Differences

To resolve a difference, you accept the change in either the left or right pane.

To accept a difference:

- **Click the Accept button, or Click the Accept & Next button to accept the difference and move to the next difference.**

Understanding Glyphs

Glyphs help you understand the differences between files. There are three types of glyph:

Glyph type	Designates
Plus sign	New line
Minus sign	Deleted line
Vertical bar	Change in line
No glyph	No changes in line

Comparing Three Input Files

When an common ancestor file is designates, glyphs next to the lines in each file indicate when they differ from the corresponding lines in the ancestor:

- If a line is identical in all three files, there is no glyph.
- If a line was added to one or both of the descendants, then a plus sign (+) shows where the line was added, and the line is highlighted.
- If a line is present in the ancestor but was removed from one or both of the descendants, then a minus sign (-) shows where the line was removed, and the line is highlighted and in strikethrough font.
- If a line is in the ancestor but has been edited in one or both of the descendants, then a vertical bar (|) shows where the line was changed, and the characters that differ are highlighted.

When differences are resolved, the glyphs change to an outline font.

Merging Automatically

Merging can resolve differences automatically, based on the following rules:

- If a line has not changed in all three files, it is placed in the output file.
- If a line has changed in one of the descendants, the changed line is placed in the output file. Changes could be the addition or removal of an entire line, or an edit to a line.

- If identical changes have been made to a line in both descendants, the changed line is placed in the output file.
- If a line has been edited in both descendant files so that it is different in all three files, no line is placed in the output file. You must decide how to resolve the difference—either by choosing a line from a descendant, or by editing the merged file by hand.

When Merging automatically resolves a difference, it changes the glyphs to outline font. Merging lets you examine automatically resolved differences to be sure that it has made the correct choices.

You can disable Auto Merge by choosing Options ➤ Auto Merge. When automatic merging is disabled, the output file contains only the lines that are identical in all three files and relies on you to resolve the differences.

If you do not specify an ancestor file, Merging has no reference with which to compare a difference between the two input files. Consequently, Merging cannot determine which line in a difference is likely to represent the desired change. The result of an auto merge with no ancestor is the same as disabling automatic merging: Merging constructs a merged file using only lines that are identical in both input files and relies on you to resolve differences.

Saving the Output File

Save the output file by clicking on the Save button or choosing File ➤ Save. The name of the output file is the name you specify in the Output File field.

To change the name of the output file while saving, choose Save As and fill in the new file and directory names in the resulting pop-up window, as shown in the following figure.

Command Reference

The complete `twmerge` command is summarized below, with command options enclosed in square brackets.

Usage

```
twmerge [-b] [-r] [-a  ancestor] [-f1  name1] [-f2  name2]
[-l listfile] [  leftfile rightfile [outfile] ]
```

Options

`-b`
Ignores leading blanks and tabs when comparing lines.

`-r`
Starts in read-only mode. Only the input file text panes are displayed, and the output text pane is absent.

`-a` *ancestor*
Specifies a common ancestor file of the two files to be merged. The output file is based on this ancestor file and the changes to it made in the descendants.

When used with the `-l` *listfile* option, *ancestor* is a directory of files, which you can load in succession.

`-f1` *name1*
Sets the file name displayed for the left pane. Use when a list of files is being loaded with the `-l` option, and you want to display a reference name in the twmerge window.

If you are loading files from two directories that correspond to two different revisions of a product, you could use the `-f1` option to display the name `Rev1` above the left pane and the `-f2` option to display the name `Rev2` above the right pane.

`-f2` *name2*
Sets the file name displayed for the right file pane.

`-l` *listfile*
Specifies a file that contains a list of individual file names. Use when merging entire project directories.

twmerge uses the names in *listfile* to successively load files from directories you name with the *leftfile* and *rightfile* arguments, placing the output files in the directory you name with the *outfile* argument. The names in *listfile* must

match file names in the *leftfile* and *rightfile* directories. When used with the `-a` *ancestor* option, the *ancestor* argument must be a directory: twmerge looks in the *ancestor* directory for files that have the same names as those in *listfile* and use those with matching names as ancestor files for each merge.

If you start twmerge with the `-l` option, the Load item in the File menu changes to Load Next From List. To load successive files named in *listfile*, choose this menu item.

If you type the character "`-`" for *listfile*, twmerge reads the list of files from standard input.

leftfile—The name of the file to be loaded into the left pane for comparison. When used with the `-l` *listfile* option, *leftfile* is a directory of files, which you can load in succession.

rightfile—The name of the file to be loaded into the right pane for comparison. When used with the `-l` *listfile* option, *rightfile* is a directory of files, which you can load in succession.

Note – If you use the `-l` *listfile* option, then all three input file names (*ancestor*, *leftfile*, and *rightfile*) must be directories. If you do *not* use the `-l` *listfile* option, then any two input file names can be directories, but one of the three must be a simple file name. In this case, twmerge uses the file name to find a file with the same local name in each directory.

---

outfile—Specifies the name of the merged output file. If you do not specify an *outfile*, the output file is given the default name `twmerge.out`. If you want to specify a different name when you save the file, use File Save As.

When used with the `-l` *listfile* option, *outfile* names the directory to be used when each merged output file is saved. Individual file names in the *outfile* directory are the same as the names listed in *listfile*.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X