Debugging a Program With dbx

A Sun Microsystems, Inc.
Business
901 San Antonio Road
Palo Alto, , CA 94303-4900

Part No: 805-4948
Revision A, February 1999

USA 650 960-1300 fax 650 969-9131

Debugging a Program With dbx

Part No: 805-4948
Revision A, February 1999

Contents

Preface

This manual describes how to use `dbx` utility to accomplish debugging tasks from the command line.

Who Should Use This Book

This manual is intended for programmers with a working knowledge of Fortran, C, or C++, and some understanding of the Solaris™ operating environment and UNIX® commands, who want to debug an application uaing `dbx` commands rather than the Sun™ WorkShop™ Debugging graphical user interface (GUI). For a discussion of the main development features of Sun WorkShop, including the basics of the Debugging GUI, see *Using Sun WorkShop.*

How This Book Is Organized

Debugging a Program With dbx contains the following chapters:

Chapter 1" 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.

Chapter 2" describes how to adjust dbx environment 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.

Chapter 3" covers visiting code, visiting functions, locating symbols and looking up variables, members, types, and classes.

Chapter 4” describes how to run, attach to, continue, stop, and rerun a program in dbx, and how to single-step through program code.

Chapter 5” describes how to set, clear, and list breakpoints and traces, and how to use watchpoints.

Chapter 6” 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.

Chapter 7” describes how to examine the call stack, and how to debug a core file with the where command

Chapter 8” 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.”

Chapter 9” describes how to use runtime checking (RTC), which enables you to automatically detect runtime errors in an application during the development phase.

Chapter 10” Tells you how to to display your data graphically as you debug your program from Sun WorkShop.

Chapter 11” describes the fix and continue feature that allows you to modify and recompile a source file and continue executing without rebuilding the entire program.

Chapter 12” describes how to find information about threads by using the dbx thread commands.

Chapter 13” describes several dbx facilities to help you debug processes that create children.

Chapter 14” describes how to use dbx to work with signals.

Chapter 15” describes the dbx collector commands you can use to collect performance data.

Chapter 16” describes dbx’s support of C++ templates, and discusses the commands that are available for handling C++ exceptions, and how dbx handles these exceptions.

Chapter 17” introduces some dbx features to be used with Fortran.

Chapter 18” 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.

Chapter 19” explains differences between ksh-88 and dbx command language.

Chapter 20” describes dbx’s debugging support for programs that use dynamically-linked, shared libraries.

Appendix A” focuses on dbx usage and commands that change your program or change the behavior of your program as compared to running it without dbx.

Appendix B” describes incremental linking, `ild`-specific features, example messages, and `ild` options.

Appendix C”provides tips on using various facilities in `dbx` more effectively.

Multi-Platform Release

Note - The name of the latest Solaris operating environment release is Solaris 7 but code and path or package path names may use Solaris 2.7 or SunOS 5.7.

The Sun™ WorkShop™ documentation applies to Solaris 2.5.1, Solaris 2.6, and Solaris 7 operating environments on:

- The SPARC™ platform
- The x86 platform, where x86 refers to the Intel implementation of one of the following: Intel 80386, Intel 80486, Pentium, or the equivalent

Note - The term “x86” refers to the Intel 8086 family of microprocessor chips, including the Pentium, Pentium Pro, and Pentium II processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term “x86” refers to the overall platform architecture. Features described in this book that are particular to a specific platform are differentiated by the terms “SPARC” and “x86” in the text.

Related Books

The following Sun manuals and guides provide additional useful information.

Other Sun WorkShop Books

- *Sun WorkShop Quick Install* provides installation instructions.
- *Sun WorkShop Installation and Licensing Reference* provides supporting installation and licensing information.
- *Using Sun WorkShop* gives information on performing development operations through Sun WorkShop.

- *Analyzing Program Performance With Sun WorkShop* describes the profiling tools; the LoopTool, LoopReport, and LockLint utilities; and use of the Sampling Analyzer to enhance program performance.
- *Sun WorkShop TeamWare User's Guide* describes how to use the Sun WorkShop TeamWare code management tools.

Other Programming Books

- *C User's Guide* describes compiler options, pragmas, and more.
- *Sun Visual WorkShop C++ Overview* gives a high-level outline of the C++ package suite.
- *C++ User's Guide* provides information on command-line options and how to use the compiler.
- *C++ Programming Guide* discusses issues relating to the use of templates, exception handling, and interfacing with FORTRAN 77.
- *C++ Migration Guide* describes migrations between compiler releases.
- *C++ Library Reference* explains the `iostream` libraries.
- *Tools.h++ User's Guide* provides details on the `Tools.h++` class library.
- *Tools.h++ Class Library Reference* discusses use of the C++ classes for enhancing the efficiency of your programs.
- *Sun Performance WorkShop Fortran Overview* gives a high-level outline of the Fortran package suite.
- *Fortran User's Guide* provides information on command-line options and how to use the compilers.
- *Fortran Programming Guide* discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
- *Fortran Library Reference* gives details on the language and routines.
- *FORTRAN 77 Language Reference* provides a complete language reference.
- *Numerical Computation Guid*e details floating-point computation numerical accuracy issues.
- *Fortran Library Reference* gives detail on the language and routines.
- *FORTRAN 77 Language Reference Manual* provides a complete language reference.
- *Numerical Computation Guide* details floating-point computation numerical accuracy issues.
- *Standard C++ Library User's Guide* describes how to use the Standard C++ Library.
- *Standard C++ Class Library Reference* provides detail on the Standard C++ Library.

Solaris Books

The following Solaris manuals and guides provide additional useful information:

- The Solaris *Linker and Libraries Guide* gives information on linking and libraries.
- The Solaris *Programming Utilities Guide* provides information for developers about the special built-in programming tools available in the SunOS™ system.

Ordering Sun Documents

The SunDocsSM program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

Accessing Sun Documents Online

Sun WorkShop documentation is available online from several sources:

- The `docs.sun.com` Web site
- AnswerBook2™ collections
- HTML documents
- Online help and release notes

Using the `docs.sun.com` Web site

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Accessing AnswerBook2 Collections

The Sun WorkShop documentation is also available using AnswerBook2 software. To access the AnswerBook2 collections, your system administrator must have installed

the AnswerBook2 documents during the installation process (if the documents are not installed, see your system administrator or Chapter 3 of *Sun WorkShop Quick Install* for installation instructions). For information about accessing AnswerBook2 documents, see Chapter 6 of *Sun WorkShop Quick Install*, Solaris installation documentation, or your system administrator.

Note - To access AnswerBook2 documents, Solaris 2.5.1 users must first download AnswerBook2 documentation server software from a Sun Web page. For more information, see Chapter 6 of *Sun WorkShop Quick Install*.

Accessing HTML Documents

The following Sun Workshop documents are available online only in HTML format:

- Tools.h++ Class Library Reference
- Tools.h++ User's Guide
- *Numerical Computation Guide*
- Standard C++ Library User's Guide
- *Standard C++ Class Library Reference*
- *Sun WorkShop Performance Library Reference Manual*
- *Sun WorkShop Visual User's Guide*
- Sun WorkShop Memory Monitor User's Manual

To access these HTML documents:

1. **Open the following file through your HTML browser:**
   install-directory`/SUNWspro/DOC5.0/lib/locale/C/html/index.html`

   Replace *install-directory* with the name of the directory where your Sun WorkShop software is installed (the default is `/opt`).

   The browser displays an index of the HTML documents for the Sun WorkShop products that are installed.

2. **Open a document in the index by clicking the document's title.**

Accessing Sun WorkShop Online Help and Release Notes

This release of Sun WorkShop includes an online help system as well as online manuals. To find out more see:

- Online Help. A help system containing extensive task-oriented, context-sensitive help. To access the help, choose Help Help Contents. Help menus are available in all Sun WorkShop windows.
- Release Notes. The Release Notes contain general information about Sun WorkShop and specific information about software limitations and bugs. To access the Release Notes, choose Help Release Notes.

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.

TABLE P–2 System Prompts

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	`#`

CHAPTER 1

Starting dbx

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

This chapter contains the following sections:

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
```

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

```
$ dbx program_name
```

dbx Start-up Sequence

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

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, which is kalias, the alias command for the Korn shell. 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()). Typically, you want to set a breakpoint and then issue a run command, such as stop in main and run for a C program.

Note - By default, the loading of debugging information for modules compiled with -xs is delayed in the same way as the debugging information stored in .o files. The dbx environment variable delay_xs lets you turn off the delayed loading of debugging information for modules compiled with -xs and have this information loaded at dbx startup (see "Debugging Without the Presence of .o Files" on page 34).

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.

To debug a core file, type:

```
$ dbx program_name
 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 ] fromto
```

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 2” 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.

For example, to compile using C++:

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

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

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

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.

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:

A.		B.	
	debug		debug
	stop at line		stop at line
Saving a Complete Run	run next next stop at line continue next next step next save	Saving a Run Minus the Last Two Steps	run next next stop at line continue next next step next save-2

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 `save` portion of the command. By default, the value of *-number* is `-1`, so `replay` works as an undo command, restoring the last run up until but not including the last command issued.

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

```
(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

CHAPTER 2

Customizing dbx

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

This chapter is organized into the following sections:

Using the .dbxrc File

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`

Creating a `.dbxrc` File

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

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

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

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.

`dbx` Environment Variables and the Korn Shell

Each `dbx` environment variable is also accessible as a `ksh` variable. The name of the `ksh` variable is derived from the `dbx` environemnt varaible by prefixing it with `DBX_`. For example

```
dbxenv stack_verbose
```

and

```
echo $DBX_stack_verbose
```

yield the same output.

Customizing dbx in Sun WorkShop

If you use dbx through the Dbx Commands window in Sun WorkShop Debugging as well as from the command line in a shell, you can customize dbx most effectively by taking advantage of the customization features in Sun WorkShop Debugging.

Setting Debugging Options

If you use dbx primarily in the Dbx Commands window in Sun WorkShop Debugging, you can and should set most dbx environment variables using the Debugging Options dialog box (see the *Using Sun WorkShop* manual and the "Customizing Your Debugging Environment" section in the Sun WorkShop online help).

When you set debugging options using the Debugging Options dialog box, the dbxenv commands for the corresponding environment variables are stored in the Sun WorkShop configuration file `.workshoprc`. When you start to debug a program in the Sun WorkShop Debugging window, any settings in your `.dbxrc` file that conflict with those in your `.workshoprc` take precedence.

Maintaining a Unified Set of Options

If you use dbx both from the command line in a shell and from within Sun WorkShop, you can create a unified set of options that customizes dbx for both modes:

1. **In the Sun WorkShop Debugging window, choose Debug Æ Debugging Options.**
2. **Click Save as Defaults to make your current option settings your defaults.**
3. **Open your `.dbxrc` file in an editor window.**
4. **Near the top, add the line `source $HOME/.workshoprc`.**
5. **Move the relevant dbxenv commands to a location after the source line, or delete them, or comment them out.**
6. **Save the file.**

Any changes made through the Debugging Options dialog box are now available to dbx through both Sun WorkShop and when running in a shell.

Maintaining Two Sets of Options

To maintain different options settings for running dbx within Sun WorkShop and from the command line in a shell, you can use the havegui variable in your .dbxrc file to conditionalize your dbxenv commands. For example:

```
if $havegui
  then
     dbxenv follow_fork_mode ask
     dbxenv stack_verbose on
  else
      dbxenv follow_fork_mode parent
      dbxenv stack_verbose off
```

.

Storing Custom Buttons

Sun WorkShop Debugging includes the Button Editor for adding. removing, and editing buttons in the Custom Buttons window. You no longer need to use the button command to store buttons in your .dbxrc file. You cannot add buttons to, or remove buttons from, your .dbxrc file with the Button Editor.

You cannot permanently delete a button stored in your .dbxrc file with the Button Editor. The button will reappear in your next debugging session. You can remove such a button by editing your .dbxrc file.

Command Reference

To display the value of a specific variable::

dbxenv *variable*

To show all variables and their values:

dbxenv

To set the value of a variable:

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.

Table 2–1 shows all of the dbx environment variables that you can set:

TABLE 2–1 dbx Environment Variables

dbx **Environemnt Variable**	**What the Variable Does**
allow_critical_exclusion [on|off]	Normally loadobject -x disallows the exclusion of certain shared libraries critical to dbx functionality. By setting this variable to on that restriction is defeated. While this variable is on only corefiles can be debugged. Default: $DBX_allow_critical_exclusion
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. 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
delay_xs [on|off]	Governs whether debugging information for modules compiled with the -xs option is loaded at dbx startup or delayed. Default: on
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

TABLE 2–1 dbx Environment Variables *(continued)*

dbx **Environemnt Variable**	**What the Variable Does**
`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 (Sun 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 or Fortran 90 files, otherwise `true`. If `true`, case matters in variable and function names; otherwise, case is not significant. Default: autodetect
`language_mode [autodetect|main|c|ansic|c++|objc|fortran|fortran90|native_java]`	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`, `fortran`, `fortran90`, `native_java`: 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.

TABLE 2–1 dbx Environment Variables *(continued)*

dbx **Environemnt Variable**	**What the Variable Does**
`output_auto_flush [on|off]`	Automatically call fflush() after each `call`. Default: on
`output_base [8|10|16|automatic]`	Default base for printing integer constants. Default: automatic (pointers in hexadecimal characters, all else in decimal.
`output_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`ping 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

TABLE 2–1 dbx Environment Variables *(continued)*

dbx **Environemnt Variable**	**What the Variable Does**
`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
`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 Sun WorkShop. Default: on

TABLE 2–1 dbx Environment Variables *(continued)*

dbx **Environemnt Variable**	**What the Variable Does**
`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
`suppress_startup_message` n*um*	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

CHAPTER 3

Viewing and Visiting Code

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.

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

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:

```
pathmap [-c] 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 C++ only: class members of the current function's class and its base class.
3. The immediately enclosing "compilation unit," generally, the file containing the current function.
4. The LoadObject scope.
5. The global scope.
6. 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_look_aside.

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:`Blockŝblock_draw.cc̀table
function:`Blockŝblock.cc̀table::table(char*, int, int, const point&)
class:`Blockŝblock.cc̀table
class:`Blockŝmain.cc̀table
variable:       `libc.so.1ĥsearch.c̀table
```

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.cc̀wedge::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.

Looking Up Definitions of Variables, Members, and Functions

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. The debugging information is read the same way as for applications compiled with the `-xs` option. See “Debugging Without the Presence of `.o` Files” on page 34.”

Debugging Without the Presence of `.o` Files

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 this option 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; therefore the `.o` files do not have to be present in order to debug those modules.

In `dbx` 4.0, the debugging information for modules compiled with the `-xs` option is loaded during `dbx` startup. For a large program compiled with `-xs`, this may cause `dbx` to start slowly.

In `dbx` 5.0, the loading of debugging information for modules compiled with `-xs` is also delayed in the same way as the debugging information stored in `.o` files. However, you can instruct `dbx` to load the debugging information for modules compiled with `-xs` during startup. The `dbx` environment variable `delay_xs` lets you turn off the delayed loading of debugging information for modules compiled with `-xs`. To set the environment variable, add this line to your `.dbxrc` file:

```
dbxenv delay_xs off
```

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 minus 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.

CHAPTER 4

Controlling Program Execution

The commands used for running, stepping, and continuing (`run`, `rerun`, `next`, `step`, and `cont`) are called *process control* commands.

Used together with the event management commands described later in this manual, you can control the run-time behavior of a program as it executes under `dbx`.

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:

Running a Program

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). `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
```

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

```
(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. You might attach to a running process if:

- You want to debug a running server and you do not want to stop or kill it.
- You want to debug a running GUI program, and you don"t want to restart it.
- Your program is looping indefinitely, and you want to debug it without killing it.

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

Once you have debugged the program, you can then use the `detach` command to take the program out from 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` **by typingt**:

```
% 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 dbx From a Process

When you have finished debugging the program, use the detach command to detach dbx from the program. The program then resumes running independently of dbx.

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

```
(dbx) detach
```

Stepping Through 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 allows the call to be executed and stops at the following line (“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.

Single Stepping

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

```
(dbx) 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.

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 current function.

Using `cont at` *line_number* with `assign`, you can avoid executing a line of code that contains a call to a function that 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;}
```

Calling a Function

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

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

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

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

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

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()`.

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 an dproceeds with the call.

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

For C++, `dbx` handles the implicit `this` pointer, 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 a 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 this nesting. You can then re-issue the command. Alternatively, you can use the command `pop -c` to pop all frames up to the most recent call.

Using Ctrl+C to Stop a Process

You can stop a process running in `dbx` 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 Reference

`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 (lightweight process):

```
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.

Note - For information on lightweight processes (LWPs), see the Solaris *Multithreaded Programming Guide.*

---

cont

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

```
cont
```

To continue execution at line *lineid* is optional.

```
cont line id
```

To continue execution with the signal *sig*:

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

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 `undelete()` is performed.

```
debug -r
```

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

```
debug [options] – program
```

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
```

CHAPTER 5

Setting Breakpoints and Traces

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` breakpoints. If the program arrives at a breakpoint created with a `stop` command, the program halts. The program cannot resume until you issue another debugging command, such as `cont`, `step`, or `next`.
- `when` breakpoints. The program halts and `dbx` executes one or more debugging commands, then the program continues (unless one of the commands is `stop`).
- `trace` 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 file name 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 library loaded by dlopen()before that library is loaded by dlopen().
- You cannot set a breakpoint in a filter library loaded by dlopen() until the first function in it is called.
- 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 stop inmember.

To set a when breakpoint, use when inmember.

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

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

Setting Breakpoints in Member Functions of Same Class

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

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

To set a breakpoint in all member functions of the class draw:

```
(dbx) stop inclass 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 infunction` 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;}
```

Setting Breakpoints in Objects

Set an In Object breakpoint to check the operations applied to a specific object. An In Object breakpoint suspends program execution in all nonstatic member functions of the object"s class when called from the object.

To set a breakpoint in object `foo`:

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

Tracing Code

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

Setting a Trace

Set a trace by typing a `trace` command at the command line. Table 5–1 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.

TABLE 5–1 `trace` Command Syntax

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
```

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 parentheses, 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 is 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 has changed:

```
(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 in 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
]
```

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 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 `dbxenv 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.

CHAPTER 6

Event Management

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.

An example of the association of a program event with a dbx action is setting a breakpoint on a particular line.

The most generic form of creating a handler is 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.

Old	**New**
`when` *cond body*	`when step -if` *cond body*
`when` *cond* `in` *func body*	`when next -if` *cond* `-in` *func body*

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

when

When the event specified by the when command specified event occurs, the *cmds* are executed. Once the commands have all executed, the process is automatically continued.

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

stop

When the event specified by the stop command 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 event specified by the trace command oc

```
trace event-specification
```

curs, a trace message is printed:

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 “Command Reference” on page 80.

- `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` command 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.

Breakpoint Event Specifications

The following are event specifications, syntax and descriptions for breakpoint events.

`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 specified, then the designated line in the specified file is about to be executed. The file name can be the name of a source file or an object file. Although quote marks are not required, they may be necessary if the file name contains odd characters.

`at` *filename*`:`*lineno*

If the designated line is in template code, a breakpoint is placed on all instances of that template.

`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 specific object at the address denoted by *obj-expr* has been called.

Watchpoint Event Specifications

The following are event specifications, syntax, and descriptions for watchpoint events.

access *mode addr-exp*, [*byte-size-exp*]

The memory specified by *addr-exp* has been accessed.

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)
```

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

`r`	The memory has been read.
`w`	The memory has been written to.
`x`	The memory has been executed.

mode can also contain one of:

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

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

`access` is a replacement for `modify`. While both syntaxes work on Solaris 2.4, 2.5, 2.5.1, 2.6, and Solaris 7, on all of these operating environments except Solaris 2.6, `access` suffers the same limitations as `modify` and accepts only a mode of `wa`.

Limitations of access

No two matched regions may overlap.

change *variable*

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.

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

The specified address range has been modified. This is the older watchpoint facility.

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 on Solaris 2.5.1*

Addresses on the stack cannot be watched.

The event does not occur 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.

Addresses that do not exist at the time a handler for this event is created cannot be watched.

Note - Multithreaded applications are prone to deadlock so mt watchpoints are nominally disallowed. They can be turned on by setting the `dbx` environment variable `mt_watchpoints`.

System Event Specifications

The following are event specifications, syntax, and descriptions for system events.

`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 occur whenever there is any `dl`-activity. In this case, `$dlobj` is empty but `$dllist` is valid.

`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

Note - 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.

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.

lwp_exit

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

sig *sig*

This event occurs when the signal is first delivered to the debugee. *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 not 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.

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.

Execution Progress Event Specifiers

The following are event specifications, syntax, and descriptions for events pertaining to execution progress.

`next`

Similar to `step` except that functions are not stepped into.

`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 returns event is always `-temp` and can only be created in the presence of a live process.

`returns` *func*

This event executes each time the given function returns to its call site. This is not a temporary event. The return value is not provided, but you can find integral return values by accessing:

Sparc	`$o0`
Intel	`$eax`

It is another way of saying:

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

step

The `step` event occurs when the first instruction of a source line is executed. For example, you can get simple tracing with:

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

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"
```

Other Event Specifications

The following are event specificatons, syntax, and descriptions for other types of events.

attach

`dbx` has successfully attached to a process.

detach

The debugee has been detached from.

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.

proc_gone

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

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 occurs. For example, the following are equivalent:

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

sync

The process being debugged has just been executed with exec(). All memory specified in a.out is valid and present but 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 occurs after a `sync` (or `attach` if the process being debugged has not yet processed shared libraries). It executes after the dynamic linker startup code has executed and the symbol tables of all preloaded shared libraries have been loaded, but before any code in the `.init` section has run.

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

`throw`

This event occurs whenever any exception that is not unhandled or unexpected is thrown by the application.

`throw` *type*

If an exception type is specified, only exceptions of that type cause the `throw` event to occur.

`throw -unhandled`

`-unhandled` is a special exception type signifying an exception that is thrown but for which there is no handler.

`throw -unexpected`

`-unexpected` is a special exception type signifying an exception that does not satisfy the exception specification of the function that throw it.

`timer` *seconds*

Occurs when the debugee has been running for *seconds*. The timer used with this is shared with `collector` command. The resolution is in milliseconds, so a floating point value for seconds is acceptable.

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 dash (`-`), preceded by blanks. Modifiers consist of the following:

-if *cond*

The condition is evaluated when the event specified by the *event-spec* occurs. The side effect of the handler is allowed only if the condition evaluates to nonzero.

If `-if` is used with an event that has an associated singular source location, such as `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 active 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.

-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 executed only if the thread that caused it matches *tid*.

-lwp *lid*

The event is executed 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 variables 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`

Variable	Definition
`$dllist`	After `dlopen` or `dlclose` event, contains the list of load objects just dlopened or dlclosed. The first word of `dllist` is actually a “+” or a “-” depending on whether a `dlopen` or a `dlclose` has occurred.
`$newhandlerid`	ID of the most recently created handler
`$proc`	Process id of the current process being debugged
`$lwp`	Lwp id of the current LWP
`$thread`	Thread id of the current thread
`$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 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 run without the user's knowledge so that the list and location of shared libraries can be ascertained. The process is then killed. This sequence is termed booting.

While booting is occurring, all events are still available. Use this variable to distinguish the `sync` and the `syncrtld` events occurring during a `debug` and the ones occurring during a normal `run`.

Variables Valid for the Given Event

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

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.

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 the specified event occurs:

when *event-specification*
 { *command*(s);
}

stop

To stop execution at a given event:

stop *event-specification*

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
```

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 forwarding 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` 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

The `delete` command deletes breakpoints and other handlers.

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

`delete` *hid* deletes the specified handler.

`delete all, delete 0 (zero), or delete -all` deletes all handlers including temporary handlers.

`delete -temp` deletes only all temporary handlers.

`-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

A handler is created for each event that needs to be managed in a debugging session. The commands `trace`, `stop`, and `when` create handlers. Each of these commands

returns a number known as the *handler ID* (hid). The `handler`, `status`, and `delete` commands manipulate or provide information about handlers in a generic fashion.

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

`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 [ -count hid  [new-count-limit
] | -reset  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` *hid new-count-limit* assigns a new count limit to the given handler.

`handler -reset` *hid* resets the count of the handler to 0 (zero).

CHAPTER 7

Examining the Call Stack

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.

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 toward the caller's frame (and eventually `main()`) and *down* means going toward the callee (and eventually the current function). 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 in lower memory.

This chapter is organized into the following sections:

Finding Your Place on the Stack

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

`where [-f] [-h] [-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:

```
hide [ 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 | regexp
]
```

CHAPTER 8

Evaluating and Displaying Data

In dbx you can perform two types of data checking:

- Evaluate data (print) - spot-checks the value of an expression
- Display data (display) - monitors the value of an expression each time the program stops

This chapter 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:

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

To evaluate a variable or expression:

`print` *expression*

Printing C++

In C++ an object pointer has two types, its static type, what is defined in the source code, and its dynamic type. `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 that 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
```

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 Chapter 17.”

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 of 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)

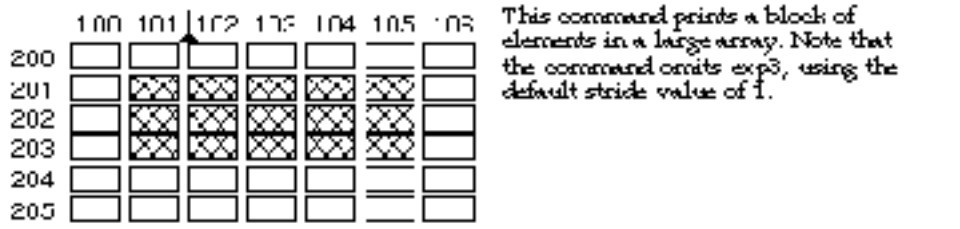
For an *n*-dimensional slice, separate the definition of each slice with a comma:

```
(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)
```



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.

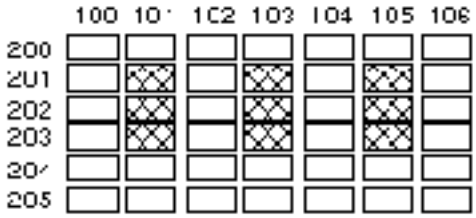
Strides

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.

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.

`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 one-dimensional array:

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*

CHAPTER 9

Using Runtime Checking

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:

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 compiler
- 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 and Continue With RTC” on page 119.

- 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 stack trace or `print` to examine variables. If the error is not a fatal error, you can continue execution of the program with the `cont` command. The program continues to the next error or breakpoint, whichever is detected first.

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 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 involves and then determines whether the operation is valid. The possible memory states are:

- Unallocated—initial state. Memory has not been allocated. It is illegal to read, write, or free this memory because it is not owned by the program.
- Allocated, but uninitialized. Memory has been allocated to the program but not initialized. It is legal to write to or free this memory, but is illegal to read it because it is uninitialized. For example, upon entering a function, stack memory for local variables is allocated, but uninitialized.
- Read-only. It is legal to read, but not write or free, read-only memory.
- Allocated and initialized. It is legal to read, write, or free allocated and initialized memory.

Using 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	Size of attempted access.
addr	Address of attempted access.
detail	More detailed information about addr. For example, if addr is in the vicinity of the stack, then its position relative to the current stack pointer is given. If addr is in the heap, then the address, size, and relative position of the nearest heap block is given.
stack	Call stack at time of error (with batch mode).
allocation	If *addr* is in the heap, then the allocation trace of the nearest heap block 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 0xefffee50
 which is 96 bytes above the current stack pointer
Variable is ‘j’
Current function is rui
   12           i = j;
```

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” on page 132.

Note - RTC does not do array bound checking, and therefore does not report array bound violations as access errors.

Using Memory Leak Checking

A memory leak is a dynamically allocated block of memory that has no pointers pointing to it anywhere in the data space of the program. Such blocks are orphaned memory. Because there are no pointers pointing to the blocks, programs cannot 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” on page 132.

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 “Runtime Checking Application Programming Interface” on page 120.

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.

Using Memory Use Checking

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 with the `showmemuse` command, which 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



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"
```

The following is the corresponding verbose memory use report:

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 load object, 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.

```
suppress mel in foo
```

Do not report memory leaks whose allocation occurs in function `foo`:

Suppress reporting blocks in use allocated from `libc.so.1`

```
suppress biu in libc.so.1
```

:

```
suppress rui in a.out
```

Suppress read from uninitialized in `a.out`:

```
suppress rua in main.cc
```

Do not report read from unallocated in file `main.cc`:

```
suppress duf at main.cc:10
```

Suppress duplicate free at line 10 of `main.cc`:

Suppress reporting of all errors in function

```
suppress all in bar
```

`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

RTC reports an error in the child

When the exec of program2 occurs, the RTC settings are inherited by program2 so access and memory use checking are enabled for that process

RTC reports an access error in the executed program, program2

```
(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
(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;
```

```
(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());
(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 and Continue With RTC

You can use RTC along with Fix and Continue to rapidly isolate and fix programming errors. Fix and Continue provides a powerful combination that can save you a lot of debugging time. Here is an example:

```
% cat -n bug.c
     1 #include stdio.h
     2 char *s = NULL;
     3
     4 void
     5 problem()
     6 {
     7      *s = "c";
     8 }
     9
    10 main()
    11 {
    12      problem();
    13      return 0;
    14 }
% cat -n bug-fixed.c
     1 #include stdio.h
     2 char *s = NULL;
     3
     4 void
     5 problem()
     6 {
     7
     8      s = (char *)malloc(1);
     9      *s = "c";
    10 }
 11
    12 main()
    13 {
```

```
    14      problem();
    15      return 0;
    16 }


yourmachine46: cc -g bug.c
yourmachine47: dbx -C a.out
Reading symbolic information for a.out
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for librtc.so
Reading symbolic information for libc.so.1
Reading symbolic information for libintl.so.1
Reading symbolic information for libdl.so.1
Reading symbolic information for libw.so.1
(dbx) check -access
access checking - ON
(dbx) run
Running: a.out
(process id 15052)
Enabling Error Checking... done
Write to unallocated (wua):
Attempting to write 1 byte through NULL pointer
Current function is problem
    7       *s = "c";
(dbx) pop
stopped in main at line 12 in file "bug.c"
   12       problem();
(dbx) #at this time we would edit the file; in this example just copy the correct version
(dbx) cp bug-fixed.c bug.c
(dbx) fix
fixing "bug.c" ......
pc moved to "bug.c":14
stopped in main at line 14 in file "bug.c"
   14       problem();
(dbx) cont

execution completed, exit code is 0
(dbx) quit
The following modules in`a.out" have been changed (fixed):
bug.c
Remember to remake program.
```

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 Sun 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`” on page 125.

RTC’s Eight 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 eight megabyte range. This means that if the debugged program has used up all the address space within eight 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 Eight Megabyte Limit

Note - On V9, you can work around the eight megabyte limit by using the setenv command to set the USE_FASTTRAPS environment variable to 1. This workaround makes dbx run more slowly and use more memory.

---

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 Sun WorkShop.

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 .
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 .
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 eight 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 ./
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 eight 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 eight megabyte 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 one megabyte patch area shared library:

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

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.

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

To turn on memory use checking:

`-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 load objects 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 load object 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 load object:

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

To change the defaults for all load objects:

```
{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 following are the error type location specifiers.

`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 load objects, 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 a
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 exis
Possible causes: All references to an allocated block are contained in registers. This can occur legitimat
  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
  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 do
  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 prog
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 h
  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
```

```
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 five access errors and five memory leaks are shown in both the leak report at the end of the run and for each `showleaks` command you issue. The default is 1000.

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`.

CHAPTER 10

Data Visualization

If you need a way to display your data graphically as you debug your program from Sun WorkShop, you can use Data Visualization.

Data visualization can be used during debugging to help you explore and comprehend large and complex datasets, simulate results, or interactively steer computations. The Data Graph window gives you the ability to "see" program data and analyze that data graphically. The graphs can be printed out or printed to a file.

This chapter contains the following sections:

Specifying Proper Array Expressions

To display your data you must specify the array, and how it should be displayed. You can invoke the Data Graph window from the Sun WorkShop Debugging window by typing an array name in the Expression text box. All scalar array types are supported except for complex (Fortran) array types.

Single-dimensional arrays are graphed (a vector graph) with the x-axis indicating the index and the y-axis indicating the array values. In the default graphic representation of a two-dimensional array (an area graph), the x-axis indicates the index of the first dimension, the y-axis indicates the index of the second dimension, while the z-axis

represents the array values. You can visualize arrays of *n* dimensions, but at most, only two of those dimensions can vary.

You do not have to examine an entire dataset. You can specify slices of an array, as shown in the following examples. Figure 10–1and Figure 10–2 show the `bf` array from the sample Fortran program given at the end of this chapter.

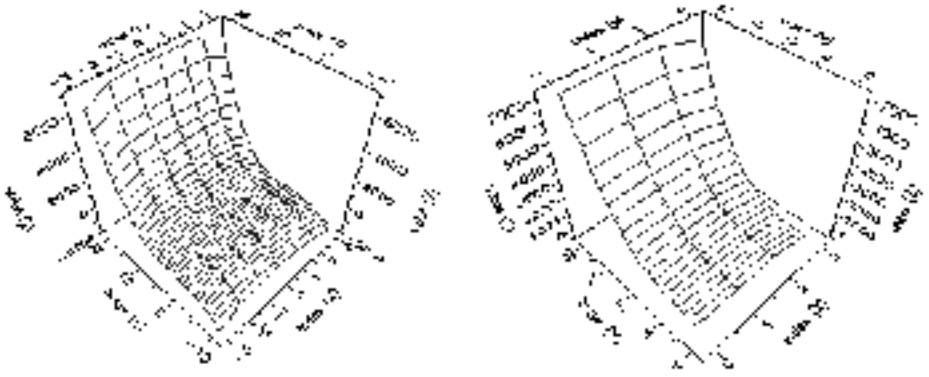


Figure 10–1 Graph of (left) bf array name only, (right) bf(0:3,1:20)

The next two figures show the array with a range of values:

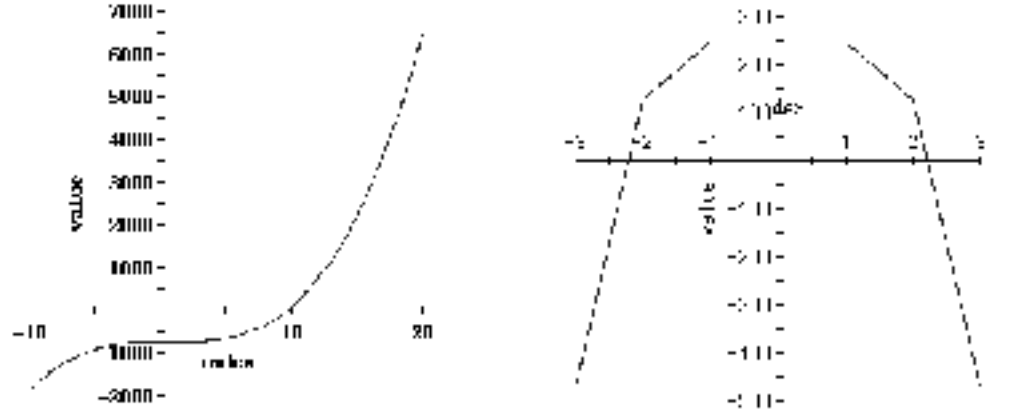


Figure 10–2 Graph of array `bf`: (left) `bf(-3:3,:)`, (right) `bf(:,-7:7)`

Graphing an Array

Before you can graph an array, you need to follow these preliminary steps:

1. **Load a program into the Debugging window.**

   Choose Debug New Program to load your program. If the program was previously loaded in the current Sun WorkShop session, choose the program from the program list in the Debug menu.

2. **Set at least one breakpoint in the program.**

   You can set a single breakpoint at the end of the program or you can set one or more at points of interest in your program.

3. **Run your program.**

   When the program stops at the breakpoint, decide which array you want to examine.

Now you can graph the array. Sun WorkShop provides multiple ways to graph an array through Sun WorkShop:

From the Debugging window. You can enter an array in the Expression text field and choose Data Graph Expression or you can select an array in a text editor and choose Data Graph Selected in the Debugging window.

From the Data Display window. You can choose the Graph command from the Data menu or from the identical pop-up menu (right-click to open the pop-up). If the array in the Data Display window can be graphed, the Graph command is active.

From the Data Graph window. You can choose Graph New, enter an array name in the Expression text field in the Data Graph: New window and click Graph.

If you click the Replace current graph button, the new graph replaces the current one. Otherwise, a new Data Graph window is opened.

From the dbx Commands window. You can display a Data Graph directly from the dbx command line with the `vitem` command (you must have opened the Dbx Commands window from Sun WorkShop):

`(dbx)` **`vitem -new`** *array-expr*

array-expr specifies the array expression to be displayed.

Automatic Updating of Array Displays

The value of an array expression can change as the program runs. You can choose whether to show the array expression"s new values at specific points within the program or at fixed time intervals with the Update at field in the graph options.

If you want the values of an array updated each time the program stops at a breakpoint, you must turn on the Update at: Program stops option. As you step through the program, you can observe the change in array value at each stop. Use this option when you want to view the data change at each breakpoint. The default setting for this feature is off.

If you have a long-running program, choose the Update at: Fixed time interval option to observe changes in the array value over time. With a fixed time update, a timer is automatically set for every *n*th period. The default time is set for one second intervals. To change the default setting, choose Graph Default Options and change the time interval in the Debugging Options dialog box (make sure you are in the Data Grapher category).

Note - Every time the timer reaches the nth period, WorkShop tries to update the graph display; however, the array could be out of scope at that particular time and no update can be made.

Since the timer is also used in collecting data and when checking for memory leaks and access checking, you cannot use the Update at Fixed time interval setting when you are running the Sampling Collector or the run-time checking feature.

Changing Your Display

Once your data is graphically displayed, you can adjust and customize the display using the controls in the Data Graph window. This section presents examples of some of graph displays that you can examine.

The Data Graph window opens with the Magnify and Shrink options present. when graphing any type of array. With an area graph, the window includes the Axis Rotation and Perspective Depth fields. These options enable you to change your view of the graph by rotating its axis or increasing or decreasing the depth perspective. To quickly rotate the graph, hold the right mouse button down with the cursor on the graph and drag to turn the graph. You can also enter the degree of rotation for each axis in the Axis Rotation field.

Click Show Options for more options. If the options are already shown, click Hide to hide the additional options.

Figure 10–3 shows two displays of the same array. The figure on the left shows the array with a Surface graph type with the Wire Mesh texture. The figure on the right shows the array with a Contour graph type delineated by range lines.

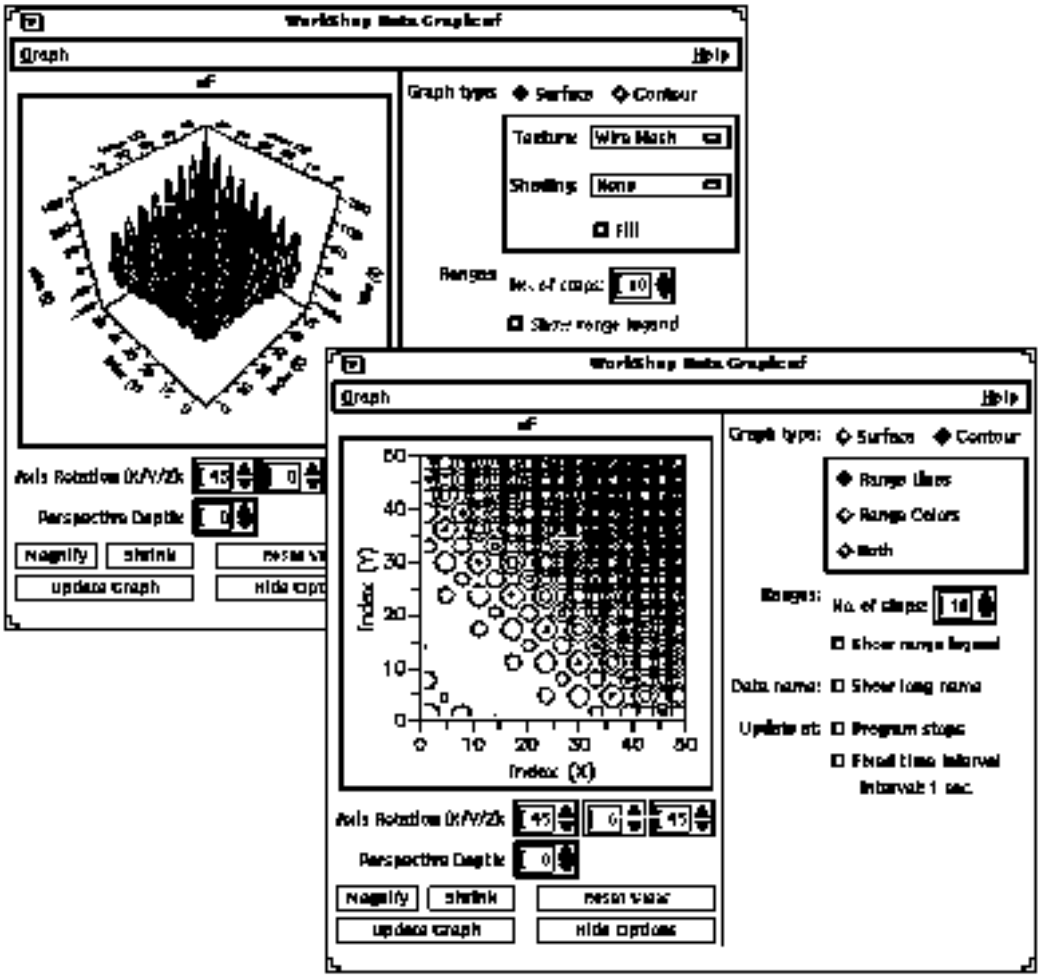


Figure 10–3 Two Displays of the Same Array

When you choose a Contour graph type, the range options enable you to see areas of change in the data values.

The display options for the Surface graph type are texture, shading, and fill. Texture choices include Wire Mesh or Range Lines as shown in Figure 10–4.

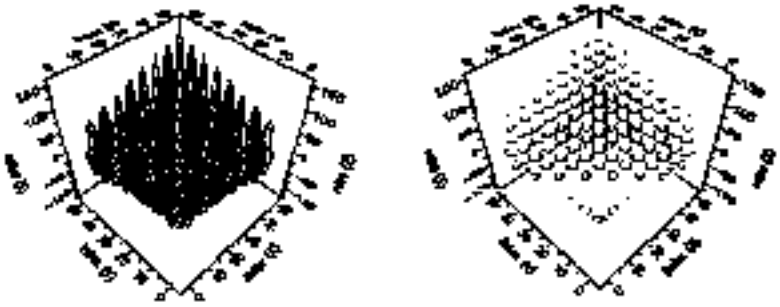


Figure 10–4 Surface graph with (left to right) Wire Mesh, Range Lines

Shading choices for the Surface graph type include Light Source or Range Colors:

Turn on Fill to shade in areas of a graph or to create a solid surface graph.

Choose Contour as the graph type to display an area graph using data range lines. With the Contour graph type, you have the additional options of displaying the graph in lines, in color, or both.

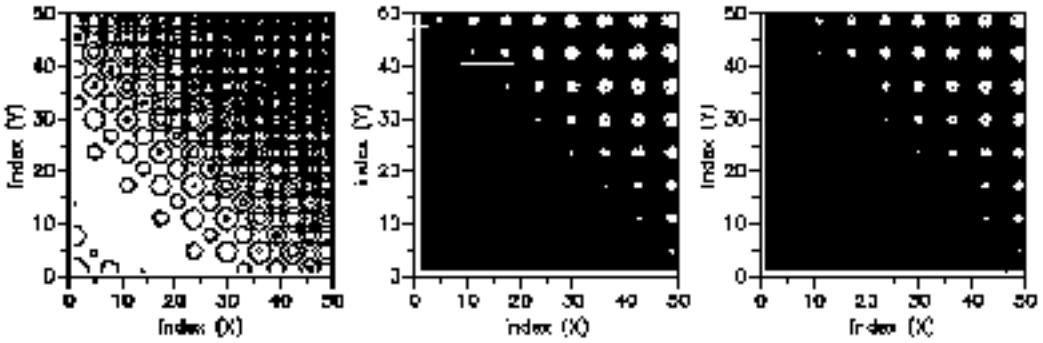


Figure 10–5 Contour graph with (left to right) Range Lines, Range Colors, Both

You can change the number of data value ranges being shown by changing the value in the Ranges: No. of steps.field.

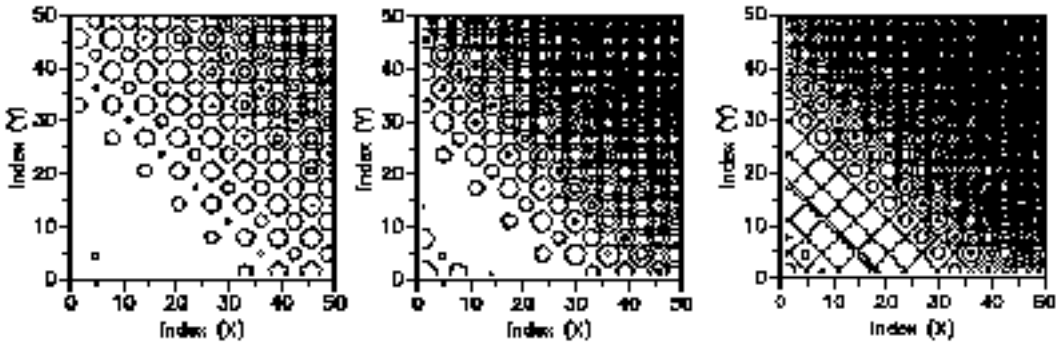


Figure 10–6 Number of steps (left to right): 5, 10, 15

Be aware that if you choose a large number of steps, you can adversely affect the color map.

Click on the Show range legend button if you want a legend to display your range intervals.

Analyzing Visualized Data

There are different ways to update the data being visualized, depending on what you are trying to accomplish. For example, you can update upon demand, at

breakpoints, or at specified time intervals. You can observe changes or analyze final results. This section provides several scenarios illustrating different situations.

Two sample programs, `dg_fexamp` (Fortran) and `dg_cexamp` (C), are included with WorkShop. These sample programs are used in the following scenarios to illustrate data visualization.

You can also use these programs for practice. They are located in *install-dir*`/SUNWspro/WS5.0/examples/datagrapher`, where the default *install-dir* is `/opt`. To use the programs, `change` to this directory and type `make`, the executable programs are created for you.

Scenario 1: Comparing Different Views of the Same Data

The same data can be graphed multiple times which allows different graph types and different segments of data to be compared.

1. **Load the C or Fortran sample program.**
2. **Set a breakpoint at the end of the program.**
3. **Start the program, running the program to that breakpoint.**
4. **Type `bf` into the Expressions text field in the Debugging window.**
5. **Choose Data Graph Expression, which brings up a Surface graph of both dimensions of the `bf` array.**
6. **Choose Data Graph Expressions again to bring up a duplicate graph.**
7. **Click Show Options and select Contour for the graph type. Now you can compare different views of the same data (Surface vs. Contour).**
8. **Type `bf(1,:)` for the Fortran or `bf[1][..]` for the C example program into the Expression text field.**
9. **Now Choose Data Graph Expression to bring up a graph of a section of the data contained in the `bf` array.**

All these different views of the data can now be compared.

Scenario 2: Updating Graphs of Data Automatically.

The updating of a graph can be controlled automatically by turning on the Update at: Program stops option. This feature enables you to make comparisons of data as it changes during the execution of the program.

1. **Load the C or Fortran sample program.**

2. **Set a breakpoint at the end of the outer loop of the `bf` function.**

3. **Start the program, running the program to that breakpoint.**

4. **Type `bf` into the Expressions text field.**

5. **Choose Data Graph Expression. A graph of the values in the `bf` array after the first loop iteration appear.**

6. **Click Show Options and select Update At: Program stops.**

7. **Choose the Go command to cause the execution of several other loop iterations of the program. Each time the program stops at the breakpoint, the graph is updated with the values set in the previous loop iteration.**

8. **Utilizing the automatic update feature can save time when an up to date view of the data is desired at each breakpoint.**

Scenario 3: Comparing Data Graphs at Different Points in Program

The updating of a graph can also be controlled manually.

1. **Load the C or Fortran example program.**

2. **Set a breakpoint at the end of the outer loop of the `af` function.**

3. **Start the program, running the program to that breakpoint.**

4. **Type `af` into the Expression: text field.**

5. **Choose Data Graph Expression. A graph of the values in the `af` array after the first loop iteration appears. Make sure automatic updating is turned off on this graph (the default setting).**

6. **Execute another loop iteration of the program using the Go command.**

7. **Bring up another graph of the `af` array choosing Data Graph Expression. This graph contains the data values set in the second iteration of the outer loop.**

8. **The data contained in the two loop iterations of the `af` array can now be compared. Any graph with automatic updating turned off can be used as a reference graph to a graph that is continually being updated automatically or manually.**

Scenario 4: Comparing Data Graphs from Different Runs of Same Program

Data graphs persist between different runs of the same program. Graphs from previous runs will not be overwritten unless they are manually updated or automatic updating is turned on.

1. **Load the C or Fortran example program.**

2. **Set a breakpoint at the end of the program.**

3. **Start the program, running the program to the breakpoint.**

4. **Type `vec` into the expressions text field, and choose Data Graph Expression.**

5. **A graph of the `vec` array appears (as a sine curve).**

6. **Now you can edit the program (for example, replace `sin` with `cos`). Use fix and continue to recompile the program and continue (click the Fix tool bar button).**

7. **Restart the program.**

8. **Because automatic updating is turned off, the previous graph does not get updated when the program reaches the breakpoint.**

9. **Choose Data Graph Expression (`vec` is still in the Expressions text field), a graph of the current `vec` values appear alongside the graph of the previous run.**

10. **The data from the two runs can now be compared. The graph of the previous run will only change if it is updated manually using the update button, or if automatic updating is turned on.**

Fortran Example Program

```
real vec(100)
real af(50,50)
real bf(-3:3,-10:20)
real cf(50, 100, 200)
int x,y,z

ct = 0

do x = 1,100
            ct = ct + 0.1
            vec(x) = sin(ct)
enddo

do x = 1,50
           do y = 1,50
              af(x,y) = (sin(x)+sin(y))*(20-abs(x+y))
           enddo
 enddo

do x = -3,3
          do y = -10,20
            bf(x,y) = y*(y-1)*(y-1.1)-10*x*x*(x*x-1)
          enddo
enddo

do x = 1,50
          do y = 1,100
            do z = 1,200
              cf(x,y,z) = 3*x*y*z - x*x*x - y*y*y - z*z*z
            enddo
          enddo
enddo

end
```

C Example Program

```
#include <math.h>
main()
{
    int x,y,z;
    float ct=0;
    float vec[100];
    float af[50][50];
    float bf[10][20];
    float cf[50][100][200];

    for (x=0; x<100; x++)
```

```
    {
          ct = ct + 0.1;
          vec[x] = sin(ct);
    }
    for (x=0; x<50; x++)
    {
          for (y=0; y<50; y++)
          {
          af[x][y] = (sin(x)+sin(y))*(20-abs(x+y));
          }
    }
    for (x=0; x<10; x++)
    {
          for (y=0; y<20; y++)
          {
            bf[x][y] = y*(y-1)*(y-1.1)-10*x*x*(x*x-1);
          }
    }
    for (x=0; x<50; x++)
    {
          for (y=0; y<100; y++)
          {
          for (z=0; z<200; z++)
          {
                    cf[x][y][z] = 3*x*y*z - x*x*x - y*y*y - z*z*z ;
          }
          }
    }
}
```

CHAPTER 11

Fixing and Continuing

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 in “Command Reference” on page 154 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

Sun 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.**
   Sun 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 segmentation fault because the zero-pointer is still pushed on the stack. Use the `pop` command to pop one frame of the stack:

```
(dbx) pop
stopped in main at line 14 in file "testfix.cc"
      14 copy(buf);
```

If the program is continued from here, it 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) contships
```

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 `SunWS_cache`. 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.

CHAPTER 12

Debugging Multithreaded Applications

With dbx, you can examine stack traces of each thread, resume 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`.

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

This chapter is organized into the following sections:

Understanding Multithreaded Debugging

When it detects a multithreaded program, dbx tries to `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.

Note - For information on multithreaded programming and LWPs, see the Solaris *Multithreaded Programming Guide*.

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 occurred.
- 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 that 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.

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 -resume 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

TABLE 12–1 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.
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.

TABLE 12–1 Thread and LWP States *(continued)*

Thread and LWP States	Description
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.

CHAPTER 13

Debugging Child Processes

This chapter describes how to debug a child process. `dbx` has several facilities to help you debug processes that create children 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 path name 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 Sun 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 6."

CHAPTER 14

Working With Signals

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.

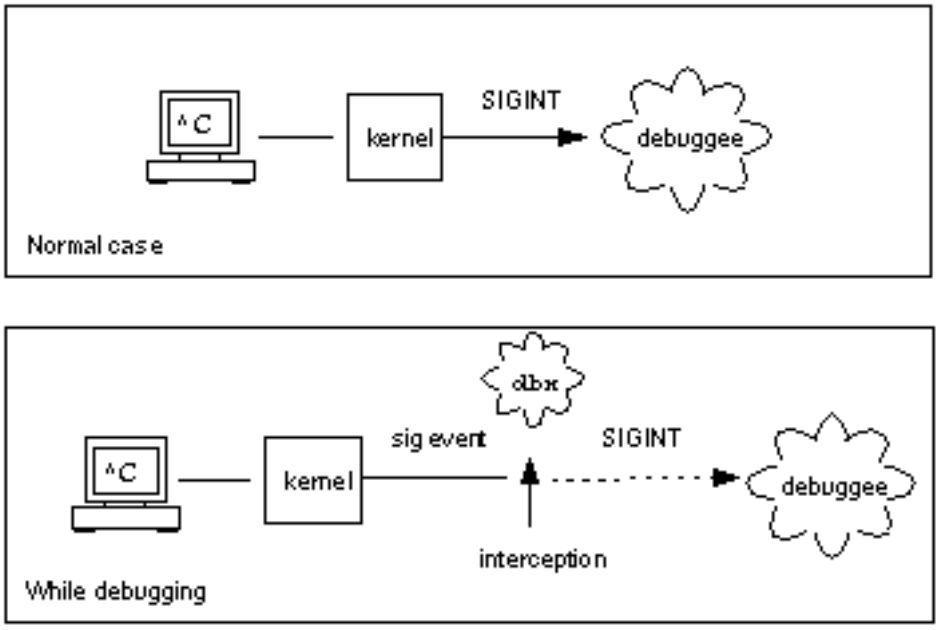
The dbx commands cont, step, and next support the -sig *signal_name* option, which allows you to resume execution of a program with the program behaving as if it had received the signal specified in the cont -sig command.

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) man page for an example of a trap handler.)

You can enable a trap using:

- `ieee_handler`

- `fpsetmask` (see the `fpsetmask`(3c) man page)
- `-ftrap` compiler flag (for FORTRAN 77 and Fortran 90, see the `f77`(1) and `f90`(1) man pages)

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.

Determining Where the Exception Occurred

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.

Determining the Cause of the Exception

To determine the cause of the exception, use the `regs -f` command to display the floating point state register (FSR). Look at the accrued exception (aexc) and current exception (cexc) fields of the register, which contain bits for the following floating-point exception conditions:

- Invalid operand
- Overflow
- Underflow
- Division by zero
- Inexact result

For more information on the floating point state register, see Version 8 (for V8) or Version 9 (for V9) of *The SPARC Architecture Manual*.For more discussion and examples, see the floating point manual, *Numerical Computation Guide.*

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 two commands have the same effect:

```
(dbx) stop sig signal
(dbx) catch signal
```

Having the signal event is more useful if you need to associate some pre-programmed action:

```
(dbx) when sig SIGCLD {echo Got $sig $signame;}
```

In this case make sure to first move `SIGCLD` to the ignore list:

```
(dbx) ignore SIGCLD
```

CHAPTER 15

Collecting Data

During the execution of an application, the Sampling 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 Sun WorkShop online help and the *Analyzing Program Performance With Sun WorkShop* manual.

This chapter is organized into the following sections:

Using the Sampling 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 turning 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 that 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
```

`collector show` *options*

To show settings of one or more categories

options can be:

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

For example:

```
collector show
```

To specify profile options:

```
collector profile options
```

options can be:

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

For example:

```
collector profile mode stack
```

To specify one or more sample options:

```
collector sample options
```

options can be:

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

For example:

```
collector sample timer 20
```

To inquire status about current experiment:

```
collector status
```

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

```
collector store options
```

options can be:

–directory *string*	Specifies directory where experiment is stored
–filename *string*	Specifies experiment file name

For example:

```
collector store directory ‘‘.test.z.er’’
```

To specify working_set options:

```
collector working_set options
```

options can be:

–mode on | off	Specifies data collection mode

For example:

```
collector working_set mode off
```

CHAPTER 16

Debugging C++

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:

For information on compiling C++ programs, see "Debugging Optimized Code" on page 4.

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.

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.

Examples of Exception Handling

This example demonstrates how exception handling is done in dbx using a sample program containing exceptions. An exception of type int is thrown in the function bar and is caught in the following catch block.

```
1  #include <stdio.h>
2
3  class c {
4      int x;
5    public:
6      c(int i) { x = i; }
7      ~c() {
8       printf("destructor for c(%d)\n", x);
9     }
10  };


11
12  void bar() {
13      c c1(3);
14      throw(99);
```

```
15  }
16
17  int main() {
18      try {
19          c c2(5);
20          bar();
21          return 0;
22      }
23      catch (int i) {
24          printf("caught exception %d\n", i);
25      }
26  }
```

The following transcript from the example program shows the exception handling features in dbx.

```
(dbx) intercept
(dbx) intercept int
int
(dbx) stop in bar
(2) stop in bar()
(dbx)run
Running: a.out
(process id 304)
Stopped in bar at line 13 in file ‘‘foo.cc’’
   13       c c1(3);
(dbx) whocatches int
int is caught at line 24, in function
main (frame number 2)
(dbx) whocatches c
dbx: no runtime type info for class c (never thrown or caught)
(dbx) contException of type int is caught at line 24,
in function main (frame number 4)
stopped in _exdbg_notify_of_throw at 0xef731494
0xef731494: _exdbg_notify_of_throw          :        jmp     %o7 + 0x8
Current function is bar
   14        throw(99);


(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8         printf("destructor
for c(%d)\n", x);
(dbx) step
destructor for c(3)
stopped in c::~c at line 9 in file "foo.cc"
    9       }
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8        printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(5)
stopped in c::~c at line 9 in file "foo.cc"
    9       )
(dbx) step
stopped in main at line 24 in file "foo.cc"
```

```
   24           printf("caught exception %d\n", i);
(dbx) step
caught exception 99
stopped in main at line 26 in file "foo.cc"
   26   }
```

Debugging With C++ Templates

dbx supports C++ templates. You can load programs containing class and function templates into dbx and invoke any of the dbx commands on a template that you would use on a class or function, such as:

- Setting breakpoints at class or function template instantiations
- 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  {
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 C++ Templates

Use these commands on templates and template instantiations. Once you know the class or type definitions, you can print values, display source listings, or set breakpoints.

`whereis` *name*

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` *function_name* (*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.

```
(dbx) call square(j,&i)
```

`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*)
```

CHAPTER 17

Debugging Fortran Using dbx

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`.

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. **Type 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, type 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, type the `next` command.**

   Advance execution to the next line:

```
(dbx) next
stopped in MAIN at line 4 in file "a1.f"
    4    print *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo =
    (1,1)       1.0
    (2,1)       0.0
    (1,2)       0.0
    (2,2)       1.0
(dbx) quit
demo%
```

   The `next` command executes the current source line and stops at the next line. It counts subprogram calls as single statements.

   Compare `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`, type the `quit` command.**

```
(dbx)quit
   demo%
```

Debugging Segmentation Faults

If a program gets a segmentation fault (`SIGSEGV`), it references a memory address outside of the memory available to it.

The most frequent causes for a segmentation fault are:

- An array index is outside the declared range.
- The name of an array index is misspelled.
- The calling routine has a `REAL` argument, which the called routine has as `INTEGER`.

- An array index is miscalculated.
- The calling routine has fewer arguments than required.
- A pointer is used before it is defined.

Using dbx to Locate Problems

Use 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 dbx segmentation 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.

```
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
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)
```

Show the sequence of calls, starting at where the execution stopped:

Working With Arrays

dbx recognizes arrays and can print them:

```
demo% dbx a.out
Reading symbolic information…
(dbx) list 1,25
    1           DIMENSION IARR(4,4)
    2           DO 90 I = 1,4
    3                   DO 20 J = 1,4
    4                           IARR(I,J) = (I*10) + J
    5   20              CONTINUE
    6   90      CONTINUE
    7           END
(dbx)  stop at 7
(1) stop at "Arraysdbx.f":7
(dbx) run
Running: a.out


stopped in MAIN at line 7 in file "Arraysdbx.f"
    7           END
(dbx) print IARR
iarr =
 (1,1) 11
 (2,1) 21
 (3,1) 31
 (4,1) 41
 (1,2) 12
 (2,2) 22
 (3,2) 32
 (4,2) 42
 (1,3) 13
 (2,3) 23
 (3,3) 33
 (4,3) 43
```

```
 (1,4) 14
 (2,4) 24
 (3,4) 34
 (4,4) 44
(dbx) print IARR(2,3)
 iarr(2, 3) = 23  - Order of user-specified subscripts ok
(dbx) quit
```

Fortran 90 Allocatable Arrays

The following example shows how to work with allocated arrays in dbx.

```
Alloc.f90
 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
```

Unknown size at line 6

Known size at line 9

```
buffer(1000) holds 1000
 (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 )
 (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)
 (dbx) cont
 stopped in main at line 9 in file "alloc.f90"
     9           PRINT *, buffer(n)
 (dbx) print n
 n = 1000
 (dbx) print buffer(n)
 buffer(n) = 1000
```

Unknown size at line 6

Known size at line 9

```
buffer(1000) holds 1000
 (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 )
(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)
(dbx) cont
stopped in main at line 9 in file "alloc.f90"
    9           PRINT *, buffer(n)
(dbx) print n
n = 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
)
```

TABLE 17–1 Variables for Array Slicing

Variable	Description	Default
arr-exp	Expression that should evaluate to an array type	
first-exp	First element in range, also first element printed	Lower bound
last-exp	Last element in range, but may not be last element printed if stride is greater than 1	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)
```

Showing 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)
```

Showing 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%
```

Showing 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%
```

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`.

DebStruc.f90

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.
 demo% f90 -o debstr -g DebStruc.f90
 demo% dbx debstr
 (dbx) stop in main
 (2) stop in main
 (dbx) list 1,99
     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



(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.

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) 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
```

CHAPTER 18

Debugging at the Machine-Instruction Level

This chapter describes how to use event management and process control commands at the machine-instruction level, how to display the contents of memory at specified addresses, and how to display source lines along with their corresponding machine instructions. The `next`, `step`, `stop` and `trace` commands each support a machine-instruction level variant: `nexti`, `stepi`, `stopi`, and `tracei`. 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 expression as an address:

```
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 16 bits (2 bytes) in decimal
D	Display as 32 bits (4 bytes) in decimal
o	Display as 16 bits (2 bytes) in octal.
O	Display as 32 bits (4 bytes) in octal.

x	Display as 16 bits (2 bytes) in hexadecimal.
X	Display as 32 bits (4 bytes) in hexadecimal. (default format)
b	Display as a byte in octal
c	Display as a character
w	Display as a wide character.
s	Display as a string of characters terminated by a null byte.
W	Display as a wide character.
f	Display as a single-precision floating point number.
F, g	Display as a double-precision floating point number.
E	Display as an extended-precision floating point number.
ld, lD	Display 32 bits (4 bytes) in decimal (same as D)
lo, lO	Display 32 bits (4 bytes) in octal (same as O)
lx, LX	Display 32 bits (4 bytes) in hexadecimal (same as X)
Ld, LD	Display 64 bits (8 bytes) in decimal
Lo, LO	Display 64 bits (8 bytes) in octal
Lx, LX	Display 64 bits (8 bytes) in hexadecimal

Count

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

Examples

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)
```

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".

For SPARC:

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x0001083c: main+0x0014:  ld      [%fp + 0x48], %l0
0x00010840: main+0x0018:  add     %l0, 0x4, %l0
0x00010844: main+0x001c:  ld      [%l0], %l0
0x00010848: main+0x0020:  or      %l0, %g0, %o0
0x0001084c: main+0x0024:  call    0x000209e8 [unresolved PLT 7: atoi]
0x00010850: main+0x0028:  nop
0x00010854: main+0x002c:  or      %o0, %g0, %l0
0x00010858: main+0x0030:  st      %l0, [%fp - 0x8]
   14       j = foo(i);
0x0001085c: main+0x0034:  ld      [%fp - 0x8], %l0
0x00010860: main+0x0038:  or      %l0, %g0, %o0
0x00010864: main+0x003c:  call    foo
0x00010868: main+0x0040:  nop
0x0001086c: main+0x0044:  or      %o0, %g0, %l0
0x00010870: main+0x0048:  st      %l0, [%fp - 0xc]
```

For Intel:

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x080488fd: main+0x000d:  movl   12(%ebp),%eax
0x08048900: main+0x0010:  movl   4(%eax),%eax
0x08048903: main+0x0013:  pushl  %eax
0x08048904: main+0x0014:  call   atoi <0x8048798>
0x08048909: main+0x0019:  addl   $4,%esp
0x0804890c: main+0x001c:  movl   %eax,-8(%ebp)
   14       j = foo(i);
0x0804890f: main+0x001f:  movl   -8(%ebp),%eax
0x08048912: main+0x0022:  pushl  %eax
0x08048913: main+0x0023:  call   foo <0x80488c0>
0x08048918: main+0x0028:  addl   $4,%esp
0x0804891b: main+0x002b:  movl   %eax,-12(%ebp)
```

Stepping and Tracing at Machine-Instruction Level

Machine-instruction level commands behave the same as their source level counterparts except that they operate at the level of single instructions instead of source lines.

Single-Stepping 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 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 5."

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) (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 and Intel that can be used in expressions.

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 $l7`	“local” registers
`$i0 through $i7`	“in” registers

Register	Description
`$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). These pairs can also be referred to as `$d0` ... `$d30`.

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

```
$g0g1 through $g6g7
$o0o1 through $o6o7
$xfsr $tstate $gsr
$f32f33 $f34f35 through $f62f63 ($d32 ... $$d62)
```

See the *SPARC Architecture Reference Manual* and the *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
`$es`	Alternate data segment register

Register	Description
`$uesp`	User stack pointer
`$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

CHAPTER 19

Using dbx With the Korn Shell

The `dbx` command language is based on the syntax of the Korn Shell (ksh 88), including I/O redirection, loops, built-in arithmetic, history, and command-line editing. This chapter lists the differences between ksh-88 and `dbx` command language.

If no `dbx` initialization file is located on startup, `dbx` assumes ksh mode.

This chapter is organized into the following sections:

ksh-88 Features Not Implemented

The following features of ksh-88 are not implemented in `dbx`:

- `set` –-`A` *name* for assigning values to array *name*
- `set` –-`o` particular options: –`allexport bgnice gmacs markdirs noclobber nolog privileged protected viraw`
- `typeset` –-`l` –-`u` –-`L` –-`R` –-`H` attributes
- backquote (‘‘) for command substitution (use `$(...)` instead)
- [[*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*.

CHAPTER 20

Debugging Shared Libraries

dbx provides full debugging support for programs that use dynamically-linked, shared libraries, provided that the libraries are compiled using the -g option.

This chapter is organized into the following sections:

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:

- Program startup. At program startup, rtld runs first and dynamically loads all shared objects specified at link time. These are *preloaded* shared objects and may include libc.so, libC.so, or libX.so. (Use ldd(1) to find out what shared objects a program will load.)
- Application requests. The application uses the function calls dlopen(3) and dlclose(3) to dynamically load and unload shared objects or executables.

dbx uses the term *load object* to refer to a shared object (.so) or executable (a.out).

Link Map

The dynamic linker maintains a list of all loaded objects in a list called a *link map*, which is maintained in the debugee's memory, and is indirectly accessed through `librtld_db.so`, a special system library for `rtld` debugging.

Startup Sequence and `.init` Sections

A `.init` section is a piece of code belonging to a shared object that is executed when the shared object is loaded. For example, the `.init` section is used by the C++ runtime system to call all static initializers in a `.so`.

The dynamic linker first maps in all the shared objects, putting them on the link map. Then, the dynamic linker traverses the link map and executes the `.init` section for each shared object. The `syncrtld` event occurs between these two phases.

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.

Debugging Support for Preloaded Shared Objects

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
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 the prompt. These dbx tasks are conducted 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.

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.

Setting a Breakpoint in a Dynamically Linked Library

dbx automatically detects that a dlopen or a dlclose has occurred and loads the symbol table of the loaded object. Once a shared object has been loaded with dlopen() you can place breakpoints in it and debug it like any part of your program.

If a shared object is unloaded via dlclose(), dbx remembers the breakpoints placed in it and re-places them if the shared object is again dlopened, even if the application is run again. (Versions of dbx prior to 5.0 would instead mark the breakpoint as '(defunct)', and it had to be deleted and re-placed by the user.)

However, you need not wait for the loading of a shared object with dlopen()in order to place a breakpoint in it, (or navigate its functions and source code for that matter). If you know the name of the shared object that the debugee will be dlopening you can arrange for dbx to preload its symbol table into dbx by using:

```
loadobjects -p /usr/java1.1/lib/libjava_g.so
```

You can now navigate the modules and functions in this loadobject and place breakpoints in it before it has ever been loaded with dlopen(). Once it is loaded dbx automatically places the breakpoints.

Setting a breakpoint in a dynamically linked library is subject to the following limitations:

1. You cannot set a breakpoint in a `dlopen` 'ed "filter" library until the first function in it is called.
2. 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`.

APPENDIX **A**

Modifying a Program State

This appendix focuses on `dbx` usage and commands that change your program or change the behavior of your program as compared to running it without `dbx`. It is important to understand which commands might make modifications to your program.

The chapter is divided into the following sections:

Impacts of Running a Program Under dbx

Your application might behave differently when run under `dbx`. Although `dbx` strives to minimize its impact on the program being debugged, you should be aware of the following:

- You might have forgotten to take out a `-C` or disable RTC. 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.
- `dbx` has to catch LWP creation and `dlopen` events, which might affect timing-sensitive multithreaded applications.
- `dbx` does context switching on signals, so if your application makes heavy use of signals, things might work differently.
- Your program might be expecting that `mmap()` always returns the same base address for mapped segments. Running under `dbx` perturbs the address space sufficiently to make it unlikely that `mmap()` returns the same address as when the program is run without `dbx`. To determine if this is a problem 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.

To minimize perturbations imposed by `dbx`, try attaching to the application while it is running in its natural environment.

Commands That Alter the State of the Program

assign

The `dbx assign` command assigns a value of the *exp* to *var*. Using it in `dbx` permanently alters the value of *var*.

`assign` *var* `=` *exp*

pop

The `dbx pop` command pops a frame or frames from the stack:

Pop current frame	`pop`
Pop num frames	`pop` *num*
Pop frames until specified frame number	`pop -f` *num*

Any calls popped are re-executed 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 11."

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.

APPENDIX **B**

Incremental Link Editor (`ild`)

This appendix describes incremental linking, `ild`-specific features, example messages, and `ild` options. This document is organized into the following sections:

Introduction

`ild` is an incremental version of the Link Editor `ld`, and replaces `ld` for linking programs. `ild` allows you to complete the development sequence (the edit, compile, link, and debug loop) efficiently and more quickly than by using a standard linker. You can avoid relinking entirely by using `fix` and `continue`. `fix` and `continue` (a part of `dbx`) allows you to work without relinking, but if you need to relink, the process can be faster if you use `ild`.

`ild` links incrementally so you can insert modified object code into an executable file that you created earlier, without relinking unmodified object files. The time required to relink depends upon the amount of code modified. Linking your application on

every build does not require the same amount of time; small changes in code can be relinked very quickly.

On the initial link, `ild` requires about the same amount of time that `ld` requires, but subsequent `ild` links can be much faster than an `ld` link. The cost of the reduced link time is an increase in the size of the executable.

Overview of Incremental Linking

When `ild` is used in place of `ld`, the initial link causes the various text, data, bss, exception table sections, etc., to be padded with additional space for future expansion (see Figure B–1). Additionally, all relocation records and the global symbol table are saved into a new persistent state region in the executable file. On subsequent incremental links, `ild` uses timestamps to discover which object files have changed and patches the changed object code into a previously built executable. That is, previous versions of the object files are invalidated and the new object files are loaded into the space vacated, or into the pad sections of the executable when needed. All references to symbols in invalidated object files are patched to point to the correct new object files.

`ild` does not support all `ld` command options. If `ild` is passed a command option that it does not support (see “Notes” on page 247), `ild` directly invokes `/usr/ccs/bin/ld` to perform the link.

How to Use `ild`

`ild` is invoked automatically by the compilation system in place of `ld` under certain conditions. When you invoke a compilation system, you are invoking a compiler driver. When you pass certain options to the driver, the driver uses `ild`. The compiler driver reads the options from the command line and executes various programs in the correct order and adds files from the list of arguments that are passed.

For example, `cc` first runs `acomp` (the front-end of the compiler), then `acomp` runs the optimizing code generator, then `cc` does the same thing for the other source files listed on the command line. The driver then generates a call to either `ild` or `ld`, depending on the options, passing it all of the files just compiled, plus other files and libraries needed to make the program complete

Figure B–1 shows an example of incremental linking.

.

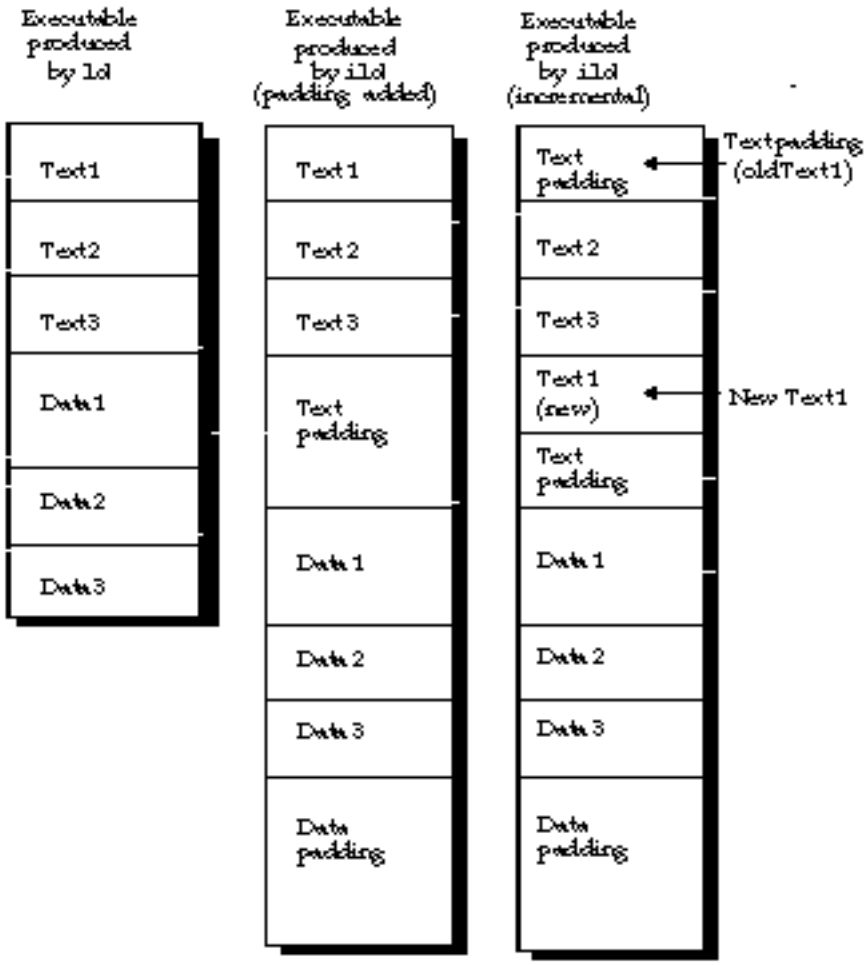


Figure B–1 Example of Incremental Linking

The compilation system options that control whether a link step is performed by `ild` or `ld` are listed here:

- `-xildon` Always use `ild`
- `-xildoff` Always use `ld`

Note - If `-xildon` and `-xildoff` are both present, the last governs.

- `-g` When neither `-xildoff` or `-G` are given, use `ild` for link-only invocations (no source files on the command line)
- `-G` Prevents the `-g` option from having any effect on linker selection

Note - Both `-g` and `-G` have other meanings which are documented as part of the compilation systems.

When you use the `-g` option to invoke debugging, and you have the default Makefile structure (which includes compile-time options such as `-g` on the link command line), you use `ild` automatically when doing development.

How `ild` Works

On an initial link, `ild` saves information about:

- All of the object files looked at.
- The symbol table for the executable produced
- All symbolic references not resolved at compile time

Initial `ild` links take about as much time as an `ld` link.

On incremental links, `ild`:

- Determines which files have changed
- Relinks the modified object files
- Uses stored information to modify changed symbolic references in the rest of the program

Incremental `ild` links are much faster than `ld` links.

In general, you do one initial link and all subsequent links are incremental.

For example, `ild` saves a list of all places where symbol `foo` is referenced in your code. If you do an incremental link that changes the value of `foo`, `ild` must change the value of all references to `foo`.

`ild` spreads out the components of the program and each section of the executable has padding added to it. Padding makes the executable modules larger than when they were linked by `ld`. As object files increase in size during successive incremental links, the padding can become exhausted. If this occurs, `ild` displays a message and does a complete full relink of the executable.

For example, as shown in Figure B–1, each of the three columns shows the sequence of text and data in a linked executable program. The left column shows text and data in an executable linked by `ld`. The center column shows the addition of text and data padding in an executable linked by `ild`. Assume that a change is made to the source file for Text 1 that causes the Text section to grow without affecting the size of the other sections. The right column shows that the original location of Text 1 has been replaced by Text padding (Text 1 has been invalidated). Text 1 has been moved to occupy a portion of the Text padding space.

To produce a smaller nonincremental executable, run the compiler driver (for example, `cc` or `CC`) with the `-xildoff` option, and `ld` is invoked to produce a more compact executable.

The resulting executable from `ild` can be debugged by `dbx` because `dbx`/Debugger understands the padding that `ild` inserts between programs.

For any command-line option that `ild` does not understand, `ild` invokes `ld`. `ild` is compatible with `ld` (in `/usr/ccs/bin/ld`). See “Options” on page 241, for details.

There are no special or extra files used by `ild`.

What `ild` Cannot Do

When `ild` is invoked to create shared objects, `ild` invokes `ld` to do the link.

Performance of `ild` may suffer greatly if you change a high percentage of object files. `ild` automatically does an full relink when it detects that a high percentage of files have been changed.

Do not use `ild` to produce the final production code for shipment. `ild` makes the file larger because parts of the program have been spread out due to padding. Because of the padding and additional time required to link, it is recommended that you do not use the `-xildon` option for production code. (Use `-xildoff` on the link line if `-g` is present.)

`ild` may not link small programs much faster, and the increase in size of the executable is greater than that for larger programs.

Third-party tools that work on executables may have unexpected results on `ild`-produced binaries.

Any program that modifies an executable, for example `strip` or `mcs`, might affect the ability of `ild` to perform an incremental link. When this happens, `ild` issues a message and performs a full relink. See “Reasons for Full Relinks” on page 237 and “Example 2: running `strip`” on page 239.

Reasons for Full Relinks

`ild` Deferred-Link Messages

The message ‘`ild: calling ld to finish link`’ . . . means that `ild` cannot complete the link, and is deferring the link request to `ld` for completion.

By default, these messages are displayed as needed. You can suppress these messages by using the `-z i_quiet` option.

`ild: calling ld to finish link --`
`cannot handle shared libraries in archive` *library name*

This message is suppressed if `ild` is implicitly requested (`-g`), but is displayed if `-xildon` is on the command line. This message is displayed in all cases if you use the `-z i_verbose` option, and never displayed if you use the `-z i_quiet` option.

`ild: calling ld to finish link -- cannot handle keyword` *Keyword*

`ild: calling ld to finish link -- cannot handle -d` *Keyword*

`ild: calling ld to finish link -- cannot handle -z` *keyword*

`ild: calling ld to finish link -- cannot handle argument` *keyword*

`ild` Relink Messages

The message '`ild: (Performing full relink)`' . . . means that for some reason `ild` cannot do an incremental link and must do a full relink. This is not an error. It is to inform you that this link will take longer than an incremental link (see "How `ild` Works" on page 236, for more details). `ild` messages can be controlled by `ild` options `-z i_quiet` and `-z i_verbose`. Some messages have a verbose mode with more descriptive text.

You can suppress all of these messages by using the `ild` option `-z i_quiet`. If the default message has a verbose mode, the message ends with an ellipsis (`[...]`) indicating more information is available. You can view the additional information by using the `-z i_verbose` option. Example messages are shown with the `-z i_verbose` option selected.

Example 1: internal free space exhausted

The most common of the full relink messages is the `internal free space exhausted` message:

```
 $ cat test1.c

 int main() { return 0; }

 $ rm a.out
# This creates test1.o
 $ cc -xildon -c -g test1.c
# This creates a.out with minimal debugging information.
 $ cc -xildon -z i_verbose -g test1.o
# A one-line compile and link puts all debugging information into a.out.
 $ cc -xildon -z i_verbose -g test1.c
```

```
 ild: (Performing full relink) internal free space in output file exhausted (sections)
$
```

These commands show that going from a one-line compile to a two-line compile causes debugging information to grow in the executable. This growth causes `ild` to run out of space and do an full relink.

Example 2: running `strip`

Another problem arises when you run `strip`. Continuing from Example 1:

# Strip a.out	$ **strip a.out**
# Try to do an incremental link	$ **cc -xildon -z i_verbose -g test1.c**
	ild: (Performing full relink) a.out has been a altered since the last incremental link – maybe you ran strip or mcs on it?
	$

Example 3: `ild` version

When a new version of `ild` is run on an executable created by an older version of `ild`, you see the following error message:

# Assume old_executable was created by an earlier version of ild	$ **cc -xildon -z i_verbose foo.o -o old_executable**
	`ild: (Performing full relink) an updated ild has been installed since a.out was last linked (2/16)`

Note - The numbers (2/16) are used only for internal reporting.

Example 4: too many files changed

Sometimes `ild` determines that it will be faster to do a full relink than an incremental link. For example:

```
$ rm a.out
$ cc -xildon -z i_verbose \
x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o test2.o
$ touch x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o
$ cc -xildon -z i_verbose \
x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o test2.o
ild: (Performing full relink) too many files changed
```

Here, use of the `touch` command causes `ild` to determine that files `x0.o` through `x8.o` have changed and that a full relink will be faster than incrementally relinking all nine object files.

Example 5: full relink

There are certain conditions that can cause a full relink on the next link, as compared to the previous examples that cause a full relink on this link.

The next time you try to link that program, you see the message:

 # ild detects previous error and does a full relink

```
$ cc -xildon -z i_verbose broken.o
ild: (Performing full relink) cannot do incremental relink due to problems in the previous link
```

A full relink occurs.

Example 6: new working directory

```
% cd /tmp
% cat y.c
int main(){ return 0;}
% cc -c y.c
```

```
                                % rm -f a.out
# initial link with cwd equal to /tmp
                                % cc -xildon -z i_verbose y.o -o a.out
                                % mkdir junk
                                % mv y.o y.c a.out junk
                                % cd junk
# incremental link, cwd is now /tmp/junk
                                % cc -xildon -z i_verbose y.o -o a.out

                                ild: (Performing full relink) current directory has changed from '/tmp' to '/tmp/junk'
                                %
```

Options

Linker control options directly accepted by the compilation system and linker options that may be passed through the compilation system to `ild` are described in this section.

Options Accepted by the Compilation System

These are linker control options accepted by the compilation system:

`-i`

Ignores `LD_LIBRARY_PATH` setting. When an `LD_LIBRARY_PATH` setting is in effect, this option is useful to influence the runtime library search, which interferes with the link editing being performed.

`-s`

Strips symbolic information from the output file. Any debugging information and associated relocation entries are removed. Except for relocatable files or shared objects, the symbol table and string table sections are also removed from the output object file.

`-V`

Output a message about the version of `ild` being used.

`-B` *dynamic* | *static*

Options governing library inclusion. Option `-B` *dynamic* is valid in dynamic mode only. These options can be specified any number of times on the command line as toggles: if the `-B` *static* option is given, no shared objects are accepted until `-B` *dynamic* is seen. See option `-l` *x*.

`-g`

The compilation systems invoke `ild` in place of `ld` when the `-g` option (output debugging information) is given, unless any of the following are true:

- The `-G` option (produce a shared library) is given
- The `-xildoff` option is present
- Any source files are named on the command line

`-d` *y* | *n*

When `-d`*y* (the default) is specified, `ild` uses dynamic linking; when `-d`*n* is specified, `ild` uses static linking. See option `-B dynamic | static`.

`-L` *path* `(space is optional)`

Adds *path* to the library search directories. `ild` searches for libraries first in any directories specified by the `-L` options, and then in the standard directories. This option is useful only if it precedes the `-l` options to which it applies on the command line. The environment variable `LD_LIBRARY_PATH` or `LD_LIBRARY_PATH_64` (in 64-bit link mode) can be used to supplement the library search path (see).

`-l` *x* `(space is optional)`

Searches a library `libx.so` or `lib.a`, the conventional names for shared object and archive libraries, respectively. In dynamic mode, unless the `-B`*static* option is in effect, `ild` searches each directory specified in the library search path for a file `libx.so` or `libx.a`. The directory search stops at the first directory containing either `ild` chooses the file ending in `.so` if `-l` expands to two files whose names are of the form `libx.so` and `libx.a`. If no `libx.so` is found, then `ild` accepts `libx.a`. In static mode, or when the `-B` *static* option is in effect, `ild` selects only the file ending in `.a`. A library is searched when its name is encountered, so the placement of `-l` is significant.

`-o` *outfile*

Produces an output object file named *outfile*. The name of the default object file is `a.out`.

`-Q` *y* | *n*

Under `-Qy`, an *ident* string is added to the `.comment` section of the output file to identify the version of the link editor used to create the file. This results in multiple *ld idents* when there have been multiple linking steps, such as when using `ld -r`. This is identical with the default action of the `cc` command. Option `-Qn` suppresses version identification.

path`-R (space is optional)`

This option gives a colon-separated list of directories that specifies library search directories to the runtime linker. If present and not null, *path* is recorded in the output object file and passed to the runtime linker. Multiple instances of this option are concatenated and separated by a colon.

`-xildoff`

Incremental linker off. Force the use of bundled `ld`. This is the default if `-g` is not being used, or `-G` is being used. You can override this default with `-xildon`.

`-xildon`

Incremental linker. Force the use of `ild` in incremental mode. This is the default if `-g` is being used. You can override this default with `-xildoff`.

`-Y P,`*dirlist* `(space is optional)`

(`cc` only) Changes the default directories used for finding libraries. Option *dirlist* is a colon-separated path list.

Note - The "`-z` *name*" form is used by `ild` for special options. The `i_` prefix to the `-z` options identifies those options peculiar to `ild`.

`-z defs`

Forces a fatal error if any undefined symbols remain at the end of the link. This is the default when building an executable. It is also useful when building a shared object to assure that the object is self-contained, that is, that all its symbolic references are resolved internally.

`-z i_dryrun`

(`ild` only.) Prints the list of files that would be linked by `ild` and exits.

`-z i_full`

(`ild` only.) Does a complete relink in incremental mode.

`-z i_noincr`

(`ild` only.) Runs `ild` in nonincremental mode (not recommended for customer use — used for testing only).

`-z i_quiet`

(`ild` only) Turns off all ild relink messages.

`-z i_verbose`

(ild only) Expands on default information on some `ild` relink messages.

`-z nodefs`

Allows undefined symbols. This is the default when building a shared object. When used with executables, the behavior of references to such “undefined symbols” is unspecified.

`-z allextract | defaultextract | weakextract`

Under `-z allextract`, all archive members are extracted from the archive. `-z defaultextract` provides a means of returning to the default following use of the former extract options. A weak reference to a symbol (`-z weakextract`) causes a file defining that symbol to be extracted from a static library.

Options Passed to `ild` from the Compilation System

The following options are accepted by ild, but you must use the form:

`-Wl,`*arg*`,`*arg* (for cc), or `-Qoption ld` *arg*`,`*arg* (for others),

to pass them to `ild` via the compilation system

`-a`

In static mode only, produces an executable object file; gives errors for undefined references. This is the default behavior for static mode. Option `-a` cannot be used with the `-r` option.

`-m`

Produces a memory map or listing of the input/output sections on the standard output.

`-t`

Turn off the warning about symbols that are defined more than once and that are not the same size.

`-e` *epsym*

Sets the entry point address for the output file to be that of the symbol *epsym*.

`-I` *name*

When building an executable, uses *name* as the path name of the interpreter to be written into the program header. The default in static mode is no interpreter; in dynamic mode, the default is the name of the runtime linker, `/usr/lib/ld.so.1`. Either case can be overridden by `-I` *name*. The `exec` system call loads this interpreter when it loads the `a.out` and passes control to the interpreter rather than to the `a.out` directly.

`-u` *symname*

Enters *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from an archive library because, initially, the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. The placement of this option on the command line is significant; it must be placed before the library that defines the symbol.

Environment

`LD_LIBRARY_PATH`

A list of directories in which to search for libraries specified with the `-l` option. Multiple directories are separated by a colon. In the most general case, it contains two directory lists separated by a semicolon: *dirlist1*; *dirlist2* If `ild` is called with any number of occurrences of `-L`, as in: `ild ...-L`*path1*`...-L`*pathn* ... then the search path ordering is: *dirlist1 path1 ... pathn dirlist2*

When the list of directories does not contain a semicolon, it is interpreted as *dirlist2*. `LD_LIBRARY_PATH` is also used to specify library search directories to the runtime linker. That is, if `LD_LIBRARY_PATH` exists in the environment, the runtime linker searches the directories named in it, before its default directory, for shared objects to be linked with the program at execution.

Note - When running a `set-user-ID` or `set-group-ID` program, the runtime linker searches only for libraries in `/usr/lib`. It also searches for any full pathname specified within the executable. A full pathname is the result of a runpath being specified when the executable was constructed. Any library dependencies specified as relative pathnames are silently ignored.

`LD_LIBRARY_PATH_64`

On Solaris 7, this environment variable is similar to LD_LIBRARY_PATH but overrrides it when searching for 64-bit dependencies.

When running Solaris 7 on a SPARC processor and linking in 32-bit mode, LD_LIBRARY_PATH_64 is ignored. If only LD_LIBRARY_PATH is defined, it is used for both 32-bit and 64-bit linking. If both LD_LIBRARY_PATH and LD_LIBRARY_PATH_64 are defined, the 32-bit linking is done using LD_LIBRARY_PATH and the 64-bit linking is done using LD_LIBRARY_PATH_64. See the Solaris *Linker and Libraries Guide* for more information on these environment variables.

LD_OPTIONS

A default set of options to ild. LD_OPTIONS is interpreted by ild as though its value had been placed on the command line immediately following the name used to invoke ild, as in: ild $LD_OPTIONS ... *other-arguments* ...

LD_PRELOAD

A list of shared objects that are to be interpreted by the runtime linker. The specified shared objects are linked in after the program being executed and before any other shared objects that the program references.

Note - When running a set-user-ID or set-group-ID program, this option is silently ignored.

LD_RUN_PATH

An alternative mechanism for specifying a runpath to the link editor (see -R option). If both LD_RUN_PATH and the -R option are specified, the -R is used.

LD_DEBUG

(not supported by ild) Provide a list of tokens that cause the runtime linker to print debugging information to the standard error. The special token help indicates the full list of tokens available.

Note - Environment variable names beginning with the characters ‘LD_ ‘are reserved for possible future enhancements to ld. Environment variable-names beginning with the characters ‘ILD_ ‘ are reserved for possible future enhancements to ild.

Notes

If `ild` determines that a command line option is not implemented, `ild` directly invokes `/usr/css/bin/ld` to perform the link.

`ld` Options Not Supported by `ild`

The following options, which may be given to the compilation system, are not supported by `ild`:

`-G`

In dynamic mode only, produces a shared object. Undefined symbols are allowed.

`-B` *symbolic*

In dynamic mode only, when building a shared object, bind references to global symbols to their definitions within the object, if definitions are available. Normally, references to global symbols within shared objects are not bound until runtime, even if definitions are available, so that definitions of the same symbol in an executable or other shared objects can override the object's own definition. `ld` issues warnings for undefined symbols unless `-z` *defs* overrides.

`-b`

In dynamic mode only, when creating an executable, does not do special processing for relocations that reference symbols in shared objects. Without the `-b` option, the link editor creates special position-independent relocations for references to functions defined in shared objects and arranges for data objects defined in shared objects to be copied into the memory image of the executable by the runtime linker. With the `-b` option, the output code can be more efficient, but it is less sharable.

`-h` *name*

In dynamic mode only, when building a shared object, records *name* in the object's dynamic section. Option *name* is recorded in executables that are linked with this object rather than the object's UNIX System file name. Accordingly, *name* is used by the runtime linker as the name of the shared object to search for at runtime.

`-z muldefs`

Allows multiple symbol definitions. By default, multiple symbol definitions occurring between relocatable objects result in a fatal error condition. This option suppresses the error condition, and allows the first symbol definition to be taken.

`-z text`

In dynamic mode only, forces a fatal error if any relocations against non-writable, allocatable sections remain.

In addition, the following options that may be passed directly to `ld`, are not supported by `ild`:

`-D` *token*`,`*token*`, ...`

Prints debugging information as specified by each token, to the standard error. The special token *help* indicates the full list of tokens available.

`-F` *name*

Useful only when building a shared object. Specifies that the symbol table of the shared object is used as a "filter" on the symbol table of the shared object specified by *name*.

`-M` *mapfile*

Reads *mapfile* as a text file of directives to `ld`. See the Solaris *Linker and Libraries Guide* for a description of mapfiles.

`-r`

Combines relocatable object files to produce one relocatable object file. `ld` does not complain about unresolved references. This option cannot be used in dynamic mode or with `-a`.

Files Used by `ild`

`libx.a`	libraries
`a.out`	output file
`LIBPATH`	usually `/usr/lib`

APPENDIX **C**

User Tips

This appendix includes the following tips:

Using dbx Equivalents for Common GDB Commands

Table C–1 lists approximate equivalent dbx commands for some common GNU Debugging (GDB) commands:

TABLE C–1 dbx Equivalents for Common 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*
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*

TABLE C–1 dbx Equivalents for Common GDB Commands *(continued)*

GDB	DBX
`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*
`define cmd`	`function cmd`
`handle` *sig*	`stop sig` *sig*
`info signals`	`status; catch`
`attach` *pid*	`debug -` *pid*
`attach` *pid*	`debug a.out` *pid*

TABLE C–1 dbx Equivalents for Common GDB Commands *(continued)*

GDB	DBX
`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*

Changes in dbx Since Release 2.0.1

A number of features have been added and changed in dbx since release 2.0.1. For lists of the changes from release to release, type `help changes` on the dbx command line or see the What's Changed in dbx section of the Sun WorkShop online help

Some of the differences between the current release and release 2.0.1 (or other vendors'variations of dbx) are described in the following sections.

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, see Chapter 2" 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

```
(dbx) /abc
```

now means to execute the file `abc` from the root directory.

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 in the shell from which `dbx` is started, 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 is 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 location. See Chapter 3” for details. The current location is usually where the program has stopped at a breakpoint

Locating Files

All files created by `dbx` are placed in the directory `/tmp` 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 1” 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++ members 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 static class member is not defined or allocated, dbx cannot determine its type so there is no type to print.

Runtime Checking Eight Megabyte Limit

Only access checking has this limit. Leak checking is not affected by this limit.

Note - On V9, you can work around the eight megabyte limit by using the setenv command to set the USE_FASTTRAPS environment variable to 1. This workaround makes dbx run more slowly and use more memory.

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 eight megabyte range. If the debugged program has used up all the address space within eight 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.

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." For more information on the eight megabyte limit, see "RTC's Eight Megabyte Limit" on page 123.

Using dbx to Locate Floating-Point Exceptions

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.
- The `syncs` command lists all synchronization objects known to `libthread`.
- The `sync` command provides information on a given synchronization object, such as which thread is blocked by it or which thread owns the locks.

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.

In the Solaris 2.5.1 operating environment, 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`.

Index